# XENIX® System V

## Development System

## Programmer's Guide

Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc. nor Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

Portions © 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988 Microsoft Corporation.
All rights reserved.
Portions © 1983, 1984, 1985, 1986, 1987, 1988 The Santa Cruz Operation, Inc.
All rights reserved.

ALL USE, DUPLICATION, OR DISCLOSURE WHATSOEVER BY THE GOVERNMENT SHALL BE EXPRESSLY SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBDIVISION (b) (3) (ii) FOR RESTRICTED RIGHTS IN COMPUTER SOFTWARE AND SUBDIVISION (b) (2) FOR LIMITED RIGHTS IN TECHNICAL DATA, BOTH AS SET FORTH IN FAR 52.227-7013.

Microsoft, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation.

SCO Document Number: XG-10-10-88-5.0/2.3

Replace this Page
with Tab Marked:

# PROGRAMMER'S GUIDE

# Contents


## 1 Introduction

1.1 Introduction 1-1
1.2 Creating Programs 1-1
1.3 Creating and Maintaining Libraries 1-2
1.4 Maintaining Program Source Files 1-2
1.5 Creating Programs with Shell Commands 1-3
1.6 Using This Guide 1-3
1.7 Notational Conventions 1-4
1.8 Referencing Commands 1-5

## 2 make: A Program Maintainer

2.1 Introduction 2-1
2.2 Creating a Makefile 2-1
2.3 Invoking make 2-3
2.4 Using Pseudo-Target Names 2-6
2.5 Using Macros 2-6
2.6 Using Shell Environment Variables 2-9
2.7 Using the Built-in Rules 2-10
2.8 Changing the Built-in Rules 2-12
2.9 Using Libraries 2-14
2.10 Troubleshooting 2-15
2.11 Using make: An Example 2-16

## 3 SCCS: A Source Code Control System

3.1 Introduction 3-1
3.2 Basic Information 3-1
3.3 Creating and Using s-files 3-5
3.4 Using Identification Keywords 3-15
3.5 Using s-file Flags 3-17
3.6 Modifying s-file Information 3-19
3.7 Printing from an s-file 3-22
3.8 Editing by Several Users 3-24
3.9 Protecting s-files 3-25
3.10 Repairing SCCS Files 3-28
3.11 Using Other Command Options 3-30

## 4 lint: A C Program Checker

4.1 Introduction 4-1
4.2 Invoking lint 4-1
4.3 Options 4-2
4.4 Checking for Unused Variables and Functions 4-3
4.5 Checking Local Variables 4-4
4.6 Checking for Unreachable Statements 4-5
4.7 Checking for Infinite Loops 4-6
4.8 Checking Function Return Values 4-7
4.9 Checking for Unused Return Values 4-7
4.10 Checking Types 4-8
4.11 Checking Type Casts 4-9
4.12 Checking for Nonportable Character Use 4-9
4.13 Checking for Assignment of longs to ints 4-10
4.14 Checking for Strange Constructions 4-10
4.15 Checking for Use of Older C Syntax 4-11
4.16 Checking Pointer Alignment 4-12
4.17 Checking Expression Evaluation Order 4-13
4.18 Embedding Directives 4-13
4.19 Checking For Library Compatibility 4-15

## 5 lex: A Lexical Analyzer

5.1 Introduction 5-1
5.2 An Overview of lex Programming 5-2
5.3 How to Format lex Programs 5-3
5.4 Specifying lex Regular Expressions 5-4
5.5 Invoking lex 5-6
5.6 Specifying Character Classes 5-6
5.7 Specifying an Arbitrary Character 5-7
5.8 Specifying Optional Expressions 5-7
5.9 Specifying Repeated Expressions 5-8
5.10 Specifying Alternation and Grouping 5-8
5.11 Specifying Context Sensitivity 5-9
5.12 Specifying Definitions 5-9
5.13 Specifying Expression Repetition 5-10
5.14 Specifying Actions 5-10
5.15 Handling Ambiguous Source Rules 5-14
5.16 Specifying Left Context Sensitivity 5-17
5.17 Specifying Source Definitions 5-19
5.18 Using lex and yacc Together 5-21
5.19 Specifying Character Sets 5-25
5.20 Source Format 5-26

**6 yacc: A Compiler-Compiler**

6.1 Introduction 6-1
6.2 Basic yacc Specifications 6-4
6.3 How the Parser Works 6-11
6.4 Ambiguity and Conflicts 6-15
6.5 How to Handle Operator Precedences 6-20
6.6 Error Handling and Recovery 6-23
6.7 The yacc Environment 6-25
6.8 Preparing Specifications 6-26
6.9 Advanced Topics 6-30
6.10 Examples 6-35
6.11 Old Features Supported but Not Encouraged 6-43

**7 Using Signals**

7.1 Introduction 7-1
7.2 Using the Signal System Call 7-1
7.3 Catching Several Signals 7-8
7.4 Controlling Execution with Signals 7-8
7.5 Using Signals in Multiple Processes 7-12

**8 adb: A Program Debugger**

8.1 Introduction 8-1
8.2 Starting and Stopping adb 8-1
8.3 Displaying Instructions and Data 8-4
8.4 Debugging Program Execution 8-16
8.5 Using the adb Memory Maps 8-32
8.6 Miscellaneous Features 8-37
8.7 Patching Binary Files 8-43

**9 ld: the XENIX Link Editor**

9.1 Introduction 9-1
9.2 Using the Link Editor 9-1
9.3 Link Editor Options 9-1
9.4 The Executable Object File 9-4
9.5 Communal Variable Allocation 9-5
9.6 Pointer and Integer Sizes 9-6
9.7 Segment and Register Sizes 9-8

## 10 m4: A Macro Processor

10.1 Introduction 10-1
10.2 Invoking m4 10-2
10.3 Defining Macros 10-2
10.4 Quoting 10-3
10.5 Using Arguments 10-6
10.6 Using Built-in Arithmetic Values 10-7
10.7 Manipulating Files 10-8
10.8 Using System Commands 10-9
10.9 Using Conditionals 10-9
10.10 Manipulating Strings 10-10
10.11 Printing 10-11

## 11 sdb: The Symbolic Debugger

11.1 Introduction 11-1

## A XENIX System Calls

A.1 Introduction A-1
A.2 Executable File Format A-1
A.3 Revised System Calls A-2
A.4 Version 7 Additions A-4
A.5 Changes to the ioctl Function A-4
A.6 Pathname Resolution A-4
A.7 Using the mount() and chown() Functions A-5
A.8 Super-Block Format A-5
A.9 Separate Version Libraries A-5

## B Kernel Error Messages

B.1 Introduction B-1
B.2 Informational Messages B-1
B.3 Warning Messages B-2
B.4 Panic Messages B-5

# Chapter 1

# Introduction

1.1 Introduction 1-1

1.2 Creating Programs 1-1

1.3 Creating and Maintaining Libraries 1-2

1.4 Maintaining Program Source Files 1-2

1.5 Creating Programs with Shell Commands 1-3

1.6 Using This Guide 1-3

1.7 Notational Conventions 1-4

1.8 Referencing Commands 1-5

## 1.1 Introduction

This guide explains how to use the XENIX Software Development System to create and maintain C language and assembly language programs. The system provides a broad spectrum of programs and commands to help you design and develop applications and system software. These programs and commands enable you to:

- create C and assembly language programs for execution on the XENIX system
- debug programs
- automatically create C and assembly language
- maintain different versions of the programs that you develop

The following sections introduce the programs and commands of the XENIX Software Development System. Some commands mentioned here are part of the XENIX Timesharing System. These are explained in the *XENIX User's Guide* and *XENIX Operations Guide.*

## 1.2 Creating Programs

The C programming language can meet the needs of most programming projects. A complete description of how to write, compile, link, and run C programs under the XENIX operating system is provided in the following guides:

- *XENIX C User's Guide*
- *XENIX C Language Reference*
- *XENIX C Library Guide*

You can also create assembly language programs using the XENIX macro assembler **masm**. It assembles source files and produces relocatable object files that can be linked to your C language programs with **ld**, the XENIX linker. **ld** links relocatable object files created by the C compiler or assembler to produce executable programs. Note that the **cc** command invokes the linker and the assembler automatically, so use of either **masm** or **ld** is optional. For a complete description of how to write, compile, link, and run assembly programs under the XENIX operating system, see the *XENIX Macro Assembler*.

You can create source files for lexical analyzers and parsers using the program generators **lex** and **yacc**. You use lexical analyzers in programs to pick patterns out of complex input and convert these patterns into meaningful values or tokens. You use parsers in programs to convert meaningful sequences of tokens and values into actions. The XENIX **lex** program generates lexical analyzers, written in C program statements, from given specification files. The XENIX **yacc** program generates parsers, written in C program statements, from given specification files. You can use **lex** and **yacc** together to make complete programs.

Special project programmers who need a convenient way to produce lexical analyzers and parsers should read "lex: A Lexical Analyzer" and "yacc: A Compiler-Compiler," for explanations of the **lex** and **yacc** program generators.

You can preprocess C and assembly language source files, or even **lex** and **yacc** source files, using the **m4** macro processor. The **m4** program performs several preprocessing functions, such as converting macros to their defined values and including the contents of files into a source file. For more information, see "m4: A Macro Processor."

## 1.3 Creating and Maintaining Libraries

You can create libraries of useful C and assembly language functions and programs using the **ar** and **ranlib** programs. **ar**, the XENIX archiver, creates libraries of relocatable object files. The XENIX random library generator **ranlib**, converts archive libraries to random libraries and places a table of contents at the front of each library. For more information on **ar**, see the *XENIX C User's Guide*. For more information on **ranlib**, see the *XENIX C Library Guide*.

## 1.4 Maintaining Program Source Files

You can automate the creation of executable programs from C and assembly language source files and maintain your source files using the **make** program and the SCCS (Source Code Control) commands. The **make** program is described in "make: A Program Maintainer," and the SCCS commands are described in "SCCS: A Source Code Control System."

The **make** program is the XENIX program maintainer. It automates the steps required to create executable programs and provides a mechanism for ensuring that programs are up-to-date. You use **make** with medium-scale programming projects.

The Source Code Control (SCCS) commands let you maintain different versions of a single program. The commands compress all versions of a source file into a single file containing a list of differences. These commands also restore compressed files to their original size and content.



Many XENIX commands let you carefully examine a program's source files. The **ctags** command creates a *tags* file so that C functions can be quickly found in a set of related C source files. The **mkstr** command creates an error message file by examining a C source file.

## 1.5 Creating Programs with Shell Commands

In some cases, it is easier to write a program as a series of XENIX shell commands than it is to create a C language program. Shell commands provide much of the same control capability as the C language, and give direct access to all the commands and programs normally available to the XENIX user.

The **csh** command invokes the C-shell, a XENIX command interpreter. The C-shell interprets and executes commands taken from the keyboard or from a command file. It has a C-like syntax which makes programming in this command language easy. It also has a facility for creating aliases, and a command history feature. For more information, see ''The C-Shell.''

## 1.6 Using This Guide

This guide is intended for programmers who are familiar with the C programming language, the assembly programming language, and with the XENIX system. The following list briefly describes each chapter.

Chapter 1, ''Introduction,'' introduces the XENIX Software Development programs provided with this package.

Chapter 2, ''make: A Program Maintainer,'' explains how to automate the development of a program or project using the **make** program.

Chapter 3, ''SCCS: A Source Code Control System,'' explains how to control and maintain all versions of a project's source files using the SCCS commands.

Chapter 4, ''lint: A C Program Checker,'' describes the XENIX program checker, **lint**, and describes the available options.

Chapter 5, "lex: A Lexical Analyzer," explains how to create lexical analyzers using the program generator **lex**.

Chapter 6, "yacc: A Compiler-Compiler," explains how to create parsers using the program generator **yacc**.

Chapter 7, "Using Signals," describes the signal functions. These functions let a program process signals that are normally processed by the system.

Chapter 8, "adb: A Program Debugger," explains how to debug C and assembly language programs using the XENIX debugger **adb**.

Chapter 9, "ld: the Link Editor" describes the design and function of the XENIX link editor, **ld**. The available options are explained in detail.

Chapter 10, "m4: A Macro Processor," explains how to use, create, and process macros using the **m4** macro processor.

Chapter 11, "sdb: The Symbolic Debugger," explains how to debug C, assembly language and Fortran programs using the XENIX debugger **sdb**.

Appendix A, "XENIX System Calls," explains how to create and use new XENIX system calls.

Appendix B, "XENIX System V Error Messages," lists and describes the system error messages produced by the XENIX kernel.

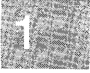
C language programmers should read the *XENIX C User's Guide* for an explanation of how to compile and debug C language programs.

Assembly language programmers should read the *XENIX Macro Assembler User's Guide* for an explanation of how to compile and debug **masm** programs.

## 1.7 Notational Conventions

This guide uses a number of special symbols to describe the syntax of XENIX commands. The following is a list of these symbols and their meaning.

| | |
|---|---|
| Examples | Examples of program fragments or commands are indented and set in monospace type. |



| | |
|---|---|
| SMALL | Small capitals indicate keynames, constants, or error conditions. |
| **bold** | Boldface characters indicate a command or program name, any command option or flag, and any function, routine, or subroutine. |
| *italics* | Italic characters indicate a filename (for example, */etc/ttys*) or a placeholder for a command argument. When typing a command, replace a placeholder with an appropriate filename, number, or option. Italics are also used to give emphasis in the text, and are used to identify the first use of a technical term. |
| `monospace` | Monospace type is used for sample command-lines, program code and examples, and sample sessions. |
| " " | Quotation marks are used in the text to set off examples of characters you actually type. |

## 1.8 Referencing Commands

Within the *XENIX Programmer's Guide*, a command may end with one of the following letters in parentheses:

(S), (F), (M), (CP), (C), or (ADM)

These notations mark which section of the *XENIX Reference* you will find a command in:

| | |
|---|---|
| (S) | System calls |
| (F) | Files and Formats |
| (M) | Miscellaneous |
| (CP) | Programming Commands |
| (C) | Commands |
| (ADM) | System Administration |

# Chapter 2

# make: A Program Maintainer


2.1 Introduction 2-1

2.2 Creating a Makefile 2-1

2.3 Invoking make 2-3

2.4 Using Pseudo-Target Names 2-6

2.5 Using Macros 2-6

2.6 Using Shell Environment Variables 2-9

2.7 Using the Built-in Rules 2-10

2.8 Changing the Built-in Rules 2-12

2.9 Using Libraries 2-14

2.10 Troubleshooting 2-15

2.11 Using make: An Example 2-16

## 2.1 Introduction

The **make** program provides an easy way to automate the creation of large programs. It reads commands from a user-defined *makefile* that lists the files to be created, the commands that create them, and the files from which they are created. When you invoke **make** to create a program, it verifies that each file on which the program depends is current, then creates the program by executing the given commands. If a file is not current, **make** updates it before creating the program. Then, **make** updates a program by executing explicitly given commands or one of the many built-in commands.



This chapter explains how to use **make** to compile automatically medium-sized programs. It explains how to create *makefiles* for each project, and how to invoke **make** for creating programs and updating files. For technical details about the program, see **make**(CP) in the *XENIX Programmer's Reference.*

## 2.2 Creating a Makefile

A *makefile* contains one or more lines of text called *dependency lines.* A dependency line shows how a given file depends on other files and what commands are required to bring a file up to date. A dependency line has the following form:

> *targets* : [ *dependents*] [ ; *commands*]

where:

- *targets* are the filenames of the files to be updated,
- *dependents* are the filenames of the files on which the target depends, and
- *commands* are the XENIX commands needed to create the target file.

Each dependency line must have at least one command associated with it, even if it is only the null command (;).

You can give more than one target filename or dependent filename, but you must separate each filename from the next by at least one space. Separate the target filenames from the dependent filenames with a colon (:). Remember to spell filenames correctly. You can also use shell metacharacters, such as asterisk (*) and question mark (?).

You can give a sequence of commands on the same line as the target and dependent filenames if you precede each command with a semicolon (;). You can give additional commands on following lines by beginning each line with a TAB character. You must type commands exactly as they would appear on a shell command line, and you can place the at sign (@) in front of a command to prevent **make** from displaying the command before executing it. Shell commands, such as **cd**(C), must appear on single lines; they must not contain the backslash (\) and Return character combination.

You can add a comment to a *makefile* by starting the comment with a number sign (#) and ending it with a Return. All characters after the number sign are ignored. If you place comments in a dependency line, they must go at the end of the line. If a command contains a number sign, you must enclose it in double quotation marks (" ").

If a dependency line is too long, you can continue it by typing a backslash (\) and a Return.

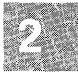
The *makefile* should be kept in the same directory as the given source files. For convenience, the filenames *makefile*, *Makefile*, *s.makefile*, and *s.Makefile*, are provided as default filenames. The **make** program uses these names if you don't supply an explicit name when the program is invoked. You can use one of these names for your *makefile*, or choose one of your own. If the filename begins with the *s.* prefix, **make** assumes that it is an SCCS file and invokes the appropriate SCCS command to retrieve the latest version of the file.

To illustrate dependency lines, consider the following example. A program named **prog** is made by linking three object files, *x.o*, *y.o*, and *z.o*. These object files are created by compiling the C language source files, *x.c*, *y.c*, and *z.c* respectively. Furthermore, the *x.c* and *y.c* files contain the line:

```
#include "defs"
```

This means that **prog** depends on the three object files, the object files depend on the C source files, and two of the source files depend on the include file *defs*. You can represent these relationships in a *makefile* with the following lines:

```
prog:  x.o  y.o  z.o
       cc  x.o  y.o  z.o  -o  prog
x.o:   x.c  defs
       cc  -c  x.c
y.o:   y.c  defs
       cc  -c  y.c
z.o:   z.c
       cc  -c  z.c
```




where:

- In the first dependency line, **prog** is the target file and *x.o*, *y.o*, and *z.o* are its dependents. The command sequence on the next line tells how to create **prog** if it is out of date:

  ```
  cc x.o y.o z.o -o prog
  ```

  The program is out of date if you have modified any one of its dependents since you last created **prog**.

- The second, third, and fourth dependency lines have the same form, with the *x.o*, *y.o*, and *z.o* files as targets and *x.c*, *y.c*, *z.c*, and *defs* files as dependents. Each dependency line has one command sequence which defines how to update the given target file.

## 2.3 Invoking make

Once you have a *makefile* and wish to update and modify one or more target files in the file, you can start **make**. The **make** command has the following syntax:

**make** [ *options*] [ *macdefs*] [ *targets*]

where:

- *options* are program options used to modify program operation.
- *macdefs* are macro definitions used to give a macro a value.
- *targets* are the filenames of the files to be updated. They must correspond to one of the target names in the *makefile*.

All arguments are optional. If you give more than one argument, you must separate them with spaces.

You can direct **make** to update the first target file in the *makefile* by typing just the program name. In this case, **make** searches for the files *makefile*, *Makefile*, *s.makefile*, and *s.Makefile* in the current directory, and uses the first one it finds as the *makefile*. For example, assume that the current *makefile* contains the dependency lines given in the previous section. Typing the following command compares the current date of the **prog** program with the current date of each of the object files *x.o*, *y.o*, and *z.o*:

```
make
```

It recreates **prog** if you have made any changes to any object file since you last created **prog**. It also compares the current dates of the object files with the dates of the four source files, *x.c*, *y.c*, *z.c*, and *defs*, and recreates the object files if the source files have changed. It does this before recreating **prog** so that you can use the recreated object files to recreate **prog**. If none of the source or object files has been altered since the last time **prog** was created, **make** announces this fact and stops. No files are changed.

You can direct **make** to update a given target file by giving the filename of the target. For example, typing the following causes **make** to recompile the *x.o* file if the *x.c* or *defs* files have changed since the object file was last created:

```
make x.o
```

Similarly, the following command causes **make** to recompile *x.o* and *z.o* if the corresponding dependents have been modified:

```
make x.o z.o
```

The **make** program processes target names from the command line in a left to right order.

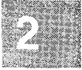
You can specify the name of the *makefile* you wish **make** to use by giving the **-f** option in the invocation. The option has the following form:

-f *makefile*

You must supply a full pathname if the file is not in the current directory. For example, the following command reads the dependency lines of the *makefile* **makeprog** found in the current directory:

```
make -f makeprog
```

You can direct **make** to read dependency lines from the standard input by entering a hyphen (-) as the *filename*. The **make** program reads the standard input until the end-of-file is encountered.



You can use the program options to modify the operation of the **make** program. The following list describes some of the options:

| Option | Description |
|---|---|
| -p | Prints the complete set of macro definitions and dependency lines in a *makefile*. |
| -i | Ignores errors returned by XENIX commands. |
| -k | Abandons work on the current entry, but continues on other branches that do not depend on that entry. |
| -s | Executes commands without displaying them. |
| -r | Ignores the built-in rules. |
| -n | Displays commands but does not execute them. **make** even displays lines beginning with the "at" sign (@). |
| -e | Ignores any macro definitions that attempt to assign new values to the shell's environment variables. |
| -t | Changes the modification date of each target file without recreating the files. |

Note that **make** executes each command in the *makefile* by passing it to a separate invocation of a shell. Because of this, take care with certain commands. For example, **cd** and shell control commands have meaning only within a single shell process; the results are forgotten before the next line is executed. If an error occurs, **make** normally stops the command.

## 2.4 Using Pseudo-Target Names

Often, you may want to include dependency lines with pseudo-target names, that is, names for which no files actually exist or are produced. Pseudo-target names allow **make** to perform tasks not directly connected with the creation of a program, such as deleting old files or printing copies of source files. For example, the following dependency line removes old copies of the given object files when the pseudo-target name *cleanup* is given in invoking **make**:

```
cleanup :
        rm  x.o y.o z.o
```

Since no file exists for a given pseudo-target name, the target is considered out-of-date. Thus, the associated command is always executed.

The **make** program also has built-in pseudo-target names that modify its operation. The pseudo-target name .Ignore causes **make** to ignore errors during execution of commands and continue after an error. This is the same as the **-i** option. Also, fBmake ignores errors for a given command if the command string begins with a hyphen (-).

The pseudo-target name .Default defines the commands to be executed when no built-in rule or user-defined dependency line exists for the given target. You can give any number of commands with this name. If you do not use .Default and you give an undefined target, **make** prints a message and stops.

The pseudo-target name .Precious prevents dependents of the current target from being deleted when **make** is terminated using the Delete (Quit) key. The pseudo-target name .Silent has the same effect as the **-s** option.

## 2.5 Using Macros

An important feature of a *makefile* is that it can contain macros. A macro is a short name that represents a filename or command option. You can define the macros when you invoke **make** or in the *makefile* itself.

A macro definition is a line containing a name, an equal sign (=), and a value. You must not precede the equal sign with a colon (:) or a TAB. The name (string of letters and digits) to the left of the equal sign is assigned the string of characters following the equal sign. Trailing blanks

and tabs on the name and leading blanks and tabs on the string are stripped. The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as its value.



You invoke a macro by preceding the macro name with a dollar sign ($). You must put macro names longer than one character in parentheses. The name of the macro is either the single character after the $ or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical.

Typically, you use macros as placeholders for values that may change from time to time. For example, the following *makefile* uses a macro for the names of the object files to be linked and one for the names of the library:

```
OBJECTS = x.o y.o z.o
LIBES = -lln
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  -o prog
```

If you invoke this *makefile* by typing the following command, it will load the three object files with the *lex* library specified with the **-lln** option:

```
make
```

You can include a macro definition in a command line. This has the same form as a macro definition in a *makefile*. If you use spaces in the definition, you should use double quotation marks (" ") to enclose the definition. Macros in a command line override corresponding definitions found in the *makefile*. For example, the following command assigns the library options **-lln** and **-lm** to LIBES:

```
make  "LIBES=-lln -lm"
```

You can modify all or part of the value generated from a macro invocation without changing the macro itself using the *substitution sequence*. The sequence has the following form:

*name* : *st1* =[ *st2* ]

where:

- *name* is the name of the macro whose value is to be modified,
- *st1* is the character or characters to be modified, and
- *st2* is the character or characters replacing the modified characters. If you do not specify *st2*, then *st1* is replaced by a null character.

You can use shell metacharacters in the substitution sequence. For example, suppose that you want to use *.x* as a metacharacter for a prefix and suppose that your *makefile* contains the following definition:

```
FILES = prog1.x prog2.x prog3.x
```

Then the macro invocation:

```
$(FILES : .x=.o)
```

will generate the value:

```
prog1.o prog2.o prog3.o
```

The actual value of FILES remains unchanged.

The **make** program has five built-in macros that can be used when writing dependency lines. The following is a list of these macros:

| Macro | Description |
|---|---|
| **$*** | Contains the name of the current target with the suffix removed. Thus if the current target is *prog.o*, **$*** contains *prog*. Use in dependency lines that redefine the built-in rules. |
| **$@** | Contains the full pathname of the current target. Use in dependency lines with user-defined target names. |
| **$<** | Contains the filename of the dependent that is more recent than the given target. Use in dependency lines with built-in target names or the .Default pseudo-target name. |
| **$?** | Contains the filenames of the dependents that are more recent than the given target. Use in dependency lines with user-defined target names. |

**$%** Contains the filename of a library member. Use with target library names. (For more information, see "Using Libraries.") In this case, **$@** contains the name of the library and **$%** contains the name of the library member.

You can change the meaning of a built-in macro by appending the D or F descriptor to its name. A built-in macro with the D descriptor contains the name of the directory containing the given file. If the file is in the current directory, the macro contains a dot (.). A macro with the F descriptor contains the name of the given file with the directory name part removed. Do not use the D and F descriptors with the **$?** macro.



## 2.6 Using Shell Environment Variables

The **make** program provides access to current values of the shell's environment variables such as Home, Path, and Login . It automatically assigns the value of each shell variable in your environment to a macro of the same name. You can access a variable's value in the same way that you access the value of explicitly defined macros. For example, in the following dependency line, $(HOME) has the same value as the user's HOME variable:

```
prog :
        cc   $(HOME)/x.o $(HOME)/y.o /usr/pub/z.o
```

**make** assigns the shell variable values after it assigns values to the built-in macros, but before it assigns values to user-specified macros. Thus, you can override the value of a shell variable by explicitly assigning a value to the corresponding macro. For example, the following macro definition causes **make** to ignore the current value of the HOME variable and use */usr/pub* instead:

```
HOME = /usr/pub
```

If a *makefile* contains macro definitions that override the current values of the shell variables, you can direct **make** to ignore these definitions using the **-e** option.

**make** has two shell variables, make and Makeflags, that correspond to two special-purpose macros.

The make macro provides a way to override the **-n** option and execute selected commands in a *makefile*. When you use make in a command, **make** will always execute that command, even if **-n** has been given in the invocation. You can set the variable to any value or command sequence.

The Makeflags macro contains one or more **make** options, and can be used in invocations of **make** from within a *makefile*. You may assign any **make** options to Makeflags except **-f**, **-p**, and **-d**. If you do not assign a value to the macro, **make** automatically assigns the current options to it, that is, the options given in the current invocation.

You can use the Make and Makeflags variables with the **-n** option to debug *makefiles* that generate entire software systems. For example, in the following *makefile*, setting MAKE to "make" and invoking this file with the **-n** options displays all the commands used to generate the programs **prog1**, **prog2**, and **prog3** without actually executing them:

```
system : prog1 prog2 prog3
     @echo  System complete.

prog1 : prog1.c
     $(MAKE) $(MAKEFLAGS) prog1

prog2 : prog2.c
     $(MAKE) $(MAKEFLAGS) prog2

prog3 : prog3.c
     $(MAKE) $(MAKEFLAGS) prog3
```

## 2.7 Using the Built-in Rules

The **make** program provides a set of built-in dependency lines, called *built-in rules*, that automatically check the targets and dependents given in a *makefile* and create up-to-date versions of these files if necessary. The built-in rules are identical to user-defined dependency lines except that they use the suffix of the filename as the target or dependent, instead of the filename itself. For example, **make** assumes automatically that all files with the *.o* suffix have dependent files with the suffixes *.c* and *.s*.

When you do not specify an explicit dependency line for a given file in a *makefile*, **make** checks the default dependents of the file automatically. It then forms the name of the dependents by removing the suffix of the given file and appending the predefined dependent suffixes. If the given file is out-of-date with respect to these default dependents, **make** searches for a

built-in rule that defines how to create an up-to-date version of the file, then executes it. There are built-in rules for the following files:

| Built-in Rule | File |
|---|---|
| *.o* | Object file |
| *.c* | C source file |
| *.r* | Ratfor source file |
| *.f* | Fortran source file |
| *.s* | Assembler source file |
| *.y* | Yacc-C source grammar |
| *.yr* | Yacc-Ratfor source grammar |
| *.l* | Lex source grammar |

For example, if you need the *x.o* file and there is an *x.c* in the description or directory, it is compiled. If there also is an *x.l*, you could run that grammar through **lex** before compiling the result.

The built-in rules are designed to reduce the size of your *makefiles*. They provide the rules for creating common files from typical dependents, for example:

```
prog:  x.o  y.o  z.o
       cc  x.o  y.o  z.o  -o  prog
x.o:  x.c  defs
       cc  -c  x.c
y.o:  y.c  defs
       cc  -c  y.c
z.o:  z.c
       cc  -c  z.c
```

In this example, the **prog** program depends on three object files, *x.o*, *y.o*, and *z.o*. These files, in turn, depend on the C language source files, *x.c*, *y.c*, and *z.c*. The *x.c* and *y.c* files also depend on the *defs* include file. In this example, each dependency and corresponding command sequence is explicitly given. Many of these dependency lines are unnecessary, since the built-in rules can be used instead. The following example is all you need to show the relationships between the files:

```
prog:  x.o  y.o  z.o
       cc  x.o  y.o  z.o  -o  prog

x.o y.o: defs
```

In this *makefile*, **prog** depends on three object files, and an explicit command is given showing how to update **prog**. However, the second line merely shows that two object files depend on the *defs* include file. No explicit command sequence is given on how to update these files if necessary. Instead, **make** uses the built-in rules to locate the desired C source files, compile these files, and create the necessary object files.

## 2.8 Changing the Built-in Rules

You can change the built-in rules by redefining the macros used in these lines or by redefining the commands associated with the rules. You can display a complete list of the built-in rules and the macros used in the rules by typing:

```
make -fp - 2>/dev/null </dev/null
```

The macros of the built-in dependency lines define the names and options of the compilers, program generators, and other programs invoked by the built-in commands. The **make** program automatically assigns a default value to these macros when you start the program. You can change the values by redefining the macro in your *makefile*. For example, the following built-in rule contains three macros, CC, Cflags, and Loadlibe

```
.c :
        $(CC) $(CFLAGS) $< $(LOADLIBES) -o $@
```

You can redefine any of these macros by placing the appropriate macro definition at the beginning of the *makefile*.

You can redefine the action of a built-in rule by giving a new rule in your *makefile*. A built-in rule has the following form:

> *suffix-rule* :
>         *command*

where:

- *suffix-rule* is a combination of suffixes showing the relationship of the implied target and dependent, and
- *command* is the XENIX command required to carry out the rule.

If you need more than one command, put each one on a separate line. The new rule must begin with an appropriate *suffix-rule*. The following *suffix-rules* are available:

| | | | |
|---|---|---|---|
| *.c* | *.c~* | *.l.o* | *.l~.o* |
| *.sh* | *.sh~* | *.y.c* | *.y~.c* |
| *.c.o* | *.c~.o* | *.l.c* | |
| | *.c~.c* | *.c.a* | *.c~.a* |
| *.s.o* | *.s~.o* | | *.s~.a* |
| *.y.o* | *.y~.o* | | *.h~.h* |

A tilde (~) indicates an SCCS file. A single suffix indicates a rule that makes an executable file from the given file. For example, the suffix rule *.c* is for the built-in rule that creates an executable file from a C source file. A pair of suffixes indicates a rule that makes one file from the other. For example, *.c.o* is for the rule that creates an object file (*.o*) from a corresponding C source file (*.c*).

Any commands in the rule may use the built-in macros provided by **make**. For example, the following dependency line redefines the action of the *.c.o* rule:

```
.c.o :
      cc68 $< -c $*.o
```

If necessary, you can also create new *suffix-rules* by adding a list of new suffixes to a *makefile* with .Suffixes. This pseudo-target name defines the suffixes that may be used to make *suffix-rules* for the built-in rules. The line has the following form:

.Suffixes: *suffix1*, *suffix2*

where *suffix* is usually a lowercase letter preceded by a dot. If you use more than one suffix in a line, you must use spaces to separate them.

The order of the suffixes is significant. Each suffix is a dependent of the suffix preceding it. For example, the following suffix list causes *prog.c* to be a dependent of *prog.o*, and *prog.y* to be a dependent of *prog.c*:

```
.SUFFIXES: .o .c .y .l .s
```

You can create new *suffix-rules* by combining a dependent suffix with the suffix of the intended target. The dependent suffix must appear first.

If you use a list of .Suffixes more than once in a *makefile*, the suffixes are combined into a single list. If you give .Suffixes that have no list, all suffixes are ignored.

## 2.9 Using Libraries

You can direct **make** to use a file contained in an archive library as a target or dependent. To do this you must explicitly name the file that you wish to access using a library name. A library name has the following form:

*lib(member-name)*

where:

- *lib* is the name of the library containing the file.
- *member-name* is the name of the file.

For example, the following library name refers to the *print.o* object file in the archive library *libtemp.a*:

```
libtemp.a(print.o)
```

You can create your own built-in rules for archive libraries by adding the *.a* suffix to the suffix list and creating new suffix combinations. For example, you can use the combination *.c.a* for a rule that defines how to create a library member from a C source file. Note that the dependent suffix in the new combination must be different from the suffix of the ultimate file. For example, you can use the *.c.a* combination for a rule that creates *.o* files, but not for one that creates *.c* files.

The most common use of the library-naming convention is to create a *makefile* that automatically maintains an archive library. For example, the following dependency lines define the commands required to create a library, named *lib*, containing up-to-date versions of the files *file1.o*, *file2.o*, and *file3.o*:

```
lib:
     lib(file1.o) lib(file2.o) lib(file3.o)
     @echo lib is now up to date
.c.a:
     $(CC) -c $(CFLAGS) $<
     ar rv $@ $*.o
     rm -f $*.o
```

The *.c.a* rule shows how to redefine a built-in rule for a library. In the following example, the built-in rule is disabled, allowing the first dependency to create the library:

```
lib:
        lib(file1.o) lib(file2.o) lib(file3.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        ar rv lib $?
        rm $?
        @echo lib is now up to date
.c.a:;
```



In this example, a substitution sequence is used to change the value of the **$?** macro from the names of the object files *file1.o*, *file2.o*, and *file3.o* to *file1.c*, *file2.c*, and *file3.c*.

## 2.10 Troubleshooting

Most difficulties in using **make** arise from **make**'s specific meaning of dependency. If the *x.c* file has the line:

```
#include "defs"
```

then the *x.o* object file depends on *defs*; the *x.c* source file does not. (If *defs* is changed, it is not necessary to do anything to the *x.c* file, while it is necessary to recreate *x.o*.)

To determine which commands **make** executes, without actually executing them, use the **-n** option. For example, the following command prints out the commands **make** normally executes without actually executing them:

```
make -n
```

The debugging option **-d** causes **make** to print out a detailed description of what **make** is doing, including the file times. Note that the output is verbose and recommended only as a last resort.

If a change to a file is certain to be benign (such as, adding a new definition to an include file), the touch (**-t**) option can save you a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the following command, which stands for touch silently, causes the relevant files to appear up-to-date:

```
make -ts
```

## 2.11 Using make: An Example

As an example of the use of **make**, examine the *makefile* in Figure 2-1, used to maintain the **make** itself. The code for **make** is spread over a number of C source files and a **yacc** grammar.

The **make** program usually prints out each command before issuing it. The following output results from giving **make** in a directory containing only the source and *makefile:*

```
cc  -c vers.c
cc  -c main.c
cc  -c doname.c
cc  -c misc.c
cc  -c files.c
cc  -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  -c gram.c
cc  vers.o main.o ... dosys.o gram.o -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars was mentioned by name in the *makefile*, **make** found them by using its suffix rules and issued the needed commands. The string of digits results from the **size make** command.

The last few targets in the *makefile* are useful maintenance sequences. The **print** target prints only the files that have been changed since the last **make print** command. A zero-length file, **print**, keeps track of the time of the printing; the **$?** macro in the command line then picks up only the names of the files changed since **print** was touched. The printed output can be sent to a different printer or to a file by changing the definition of the **P** macro.

## Example

```
# Description file for the make command

# Macro definitions below
P = lpr
FILES = Makefile vers.c defs main.c doname.c misc.c files.c dosys.c\
        gram.y lex.c
OBJECTS = vers.o main.o ... dosys.o gram.o
LIBES=
LINT = lint -p
CFLAGS = -O

#targets: dependents
#<TAB>actions

make:   $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        size make

$(OBJECTS):  defs
gram.o: lex.c

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make

print:  $(FILES)      # print recently changed files
        pr $? | $P
        touch print

test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

lint :  dosys.c doname.c files.c main.c misc.c vers.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c vers.c gram.c
        rm gram.c

arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

# Chapter 3

# SCCS: A Source Code Control System


3.1 Introduction 3-1

3.2 Basic Information 3-1
- 3.2.1 Files and Directories 3-2
- 3.2.2 Deltas and SIDs 3-2
- 3.2.3 SCCS Working Files 3-3
- 3.2.4 SCCS Command Arguments 3-4
- 3.2.5 File Administrator 3-5

3.3 Creating and Using s-files 3-5
- 3.3.1 Creating an s-file 3-6
- 3.3.2 Retrieving a File for Reading 3-7
- 3.3.3 Retrieving a File for Editing 3-8
- 3.3.4 Saving a New Version of a File 3-9
- 3.3.5 Retrieving a Specific Version 3-10
- 3.3.6 Changing the Release Number of a File 3-11
- 3.3.7 Creating a Branch Version 3-12
- 3.3.8 Retrieving a Branch Version 3-12
- 3.3.9 Retrieving the Most Recent Version 3-13
- 3.3.10 Displaying a Version 3-13
- 3.3.11 Saving a Copy of a New Version 3-14
- 3.3.12 Displaying Helpful Information 3-14

3.4 Using Identification Keywords 3-15
- 3.4.1 Inserting a Keyword into a File 3-15
- 3.4.2 Assigning Values to Keywords 3-16
- 3.4.3 Forcing Keywords 3-16

3.5 Using s-file Flags 3-17
- 3.5.1 Setting s-file Flags 3-17
- 3.5.2 Using the i Flag 3-17
- 3.5.3 Using the d Flag 3-18
- 3.5.4 Using the v Flag 3-18

3.5.5 Removing an s-file Flag 3-18

3.6 Modifying s-file Information 3-19
3.6.1 Adding Comments 3-19
3.6.2 Changing Comments 3-20
3.6.3 Adding Modification Requests 3-20
3.6.4 Changing Modification Requests 3-21
3.6.5 Adding Descriptive Text 3-21

3.7 Printing from an s-file 3-22
3.7.1 Using a Data Specification 3-22
3.7.2 Printing a Specific Version 3-23
3.7.3 Printing Later and Earlier Versions 3-24

3.8 Editing by Several Users 3-24
3.8.1 Editing Different Versions 3-24
3.8.2 Editing a Single Version 3-24
3.8.3 Saving a Specific Version 3-25

3.9 Protecting s-files 3-25
3.9.1 Adding a User to the User List 3-26
3.9.2 Removing a User from a User List 3-26
3.9.3 Setting the Floor Flag 3-27
3.9.4 Setting the Ceiling Flag 3-27
3.9.5 Locking a Version 3-27

3.10 Repairing SCCS Files 3-28
3.10.1 Checking an s-file 3-28
3.10.2 Editing an s-file 3-29
3.10.3 Changing an s-file's Checksum 3-29
3.10.4 Regenerating a g-file for Editing 3-29
3.10.5 Restoring a Damaged p-file 3-29

3.11 Using Other Command Options 3-30
3.11.1 Getting Help With SCCS Commands 3-30
3.11.2 Creating a File With the Standard Input 3-30
3.11.3 Starting at a Specific Release 3-30
3.11.4 Adding a Comment to the First Version 3-31
3.11.5 Suppressing Normal Output 3-31
3.11.6 Including and Excluding Deltas 3-32
3.11.7 Listing the Deltas of a Version 3-33
3.11.8 Mapping Lines to Deltas 3-33
3.11.9 Naming Lines 3-34
3.11.10 Displaying a List of Differences 3-34
3.11.11 Comparing SCCS Files 3-34
3.11.12 Checking a File Version 3-35

3.11.13 Removing a Delta 3-35
3.11.14 Searching for Strings 3-36

## 3.1 Introduction

The Source Code Control System (SCCS) is a collection of XENIX commands that create, maintain, and control special files called SCCS files. The SCCS commands let you create and store multiple versions of a program or document in a single file, instead of one file for each version. The commands let you retrieve any version you wish at any time, make changes to a version, and save the changes as a new version of the file in the SCCS file.

The SCCS system is useful wherever you require a compact way to store multiple versions of the same file. The SCCS system provides an easy way to update any given version of a file and explicitly record the changes made. Typically, you use the commands to control changes to multiple versions of source programs, but you can also use them to control multiple versions of guides, specifications, and other documentation.



This chapter explains how to:

- make SCCS files
- update the files contained in SCCS files
- maintain the SCCS files once they are created

The following sections describe the basic information you need to start using the SCCS commands.

## 3.2 Basic Information

This section provides some basic information about the SCCS system. In particular, it describes:

- files and directories
- deltas and SIDs
- SCCS working files
- SCCS command arguments
- file administration

### 3.2.1 Files and Directories

All SCCS files (also called *s-files*) are originally text files containing documents or programs created by a user. The text files must have been created using a XENIX text editor such as **vi**. You can use special characters in the files only if they are allowed by the given editor.

To simplify *s-file* storage, you should keep all logically related files (such as, files belonging to the same project) in the same directory. Such directories should contain *s-files* only, and should have read and examine permission for everyone and write permission for the user only.

Note that you must not use the XENIX **link** command to create multiple copies of an *s-file*.

### 3.2.2 Deltas and SIDs

Unlike an ordinary text file, an SCCS file contains nothing more than lists of changes. Each list corresponds to the changes needed to construct exactly one version of the file. You can then combine the lists to create the desired version from the original.

Each list of changes is called a *delta*. Each delta has an identification string called an *SID*. The SID is a string of at least two, and at most four, numbers separated by periods. The numbers name the version and define how it is related to other versions. For example, the first delta is usually numbered 1.1 and the second, 1.2.

The first number in any SID is called the *release number*. The release number usually indicates a group of versions that are similar and generally compatible. The second number in the SID is the *level number*. It indicates major differences between files in the same release.

An SID may also have two optional numbers. The *branch number*, the optional third number, indicates changes at a particular level; and the *sequence number*, the fourth number, indicates changes at a particular branch. For example, the SIDs 1.1.1.1 and 1.1.1.2 indicate two new versions that contain slight changes to the original delta 1.1.

An *s-file* can contain several different releases, levels, branches, and sequences of the same file. In general, the maximum number of releases an *s-file* can contain is 9999; that is, release numbers can range from 1 to 9999. The same limit applies to level, branch, and sequence numbers.

When you create a new version, the SCCS system usually creates a new SID by incrementing the level number of the original version. If you wish

to create a new release, you must instruct the system to do so explicitly. A change to a release number indicates a major new version of the file. How to create a new version of a file and change release numbers is described later.

The SCCS system creates a branch and sequence number for the SID of a new version, if the next higher level number already exists. For example, if you change version 1.3 to create a version 1.4 and then change 1.3 again, the SCCS system creates a new version named 1.3.1.1.

Version numbers can become quite complicated. You should keep the numbers as simple as possible by carefully planning the creation of each new version.



### 3.2.3 SCCS Working Files

The SCCS system uses several different kinds of files to complete its tasks. In general, these files contain either actual text or information about the commands in progress. For convenience, the SCCS system names these files by placing a prefix before the name of the original file from which all versions were made. The following is a list of the working files.

| File | Description |
|---|---|
| *s-file* | A permanent file that contains all versions of the given text file. The versions are stored as deltas, that is, lists of changes to be applied to the original file to create the given version. You form the name of an *s-file* by placing the file prefix *s.* at the beginning of the original filename. |
| *x-file* | A temporary copy of the *s-file*. You create *x-files* with SCCS commands that change the *s-file*. You use *x-files* instead of the *s-file* to carry out the changes. When all changes are complete, the SCCS system removes the original *s-file* and gives the *x-file* the name of the original *s-file*. You form the name of the *x-file* by placing the prefix *x.* at the beginning of the original file. |
| *g-file* | An ordinary text file created by applying the deltas in a given *s-file* to the original file. The *g-file* represents a copy of the given version of the original file, and as such receives the same filename as the original. When you create a *g-file*, it is placed in your current working directory. |

*p-file* A special file containing information about the versions of an *s-file* currently being edited. You create a *p-file* when you retrieve a *g-file* from the *s-file*. The *p-file* exists until you have saved all currently retrieved files in the *s-file*; it is then deleted. The *p-file* contains one or more entries describing the SID of the retrieved *g-file*, the proposed SID of the new, edited *g-file*, and the log-in name of the user who retrieved the *g-file*. You form the *p-file* name by placing the prefix *p.* at the beginning of the original filename.

*z-file* A lock file used by SCCS commands to prevent two users from updating a single SCCS file at the same time. Before a command modifies an SCCS file, it creates a *z-file* and copies its own process ID to it. Any other command that attempts to access the file while the *z-file* is present displays an error message and stops. When the original command has finished its tasks, it deletes the *z-file* before stopping. You form the *z-file* name by placing the prefix *z.* at the beginning of the original filename.

*l-file* A special file containing a list of the deltas required to create a given version of a file. You form the *l-file* name by placing the prefix *l.* at the beginning of the original filename.

*d-file* A temporary copy of the *g-file* used to generate a new delta.

*q-file* A temporary file used by the **delta** command when updating the *p-file*. The *q-file* is not directly accessible.

In general, you never directly access *x-files*, *z-files*, *d-files*, or *q-files*. If a system crash or similar situation abnormally terminates a command, you may wish to delete these files to ensure proper operation of subsequent SCCS commands.

### 3.2.4 SCCS Command Arguments

Almost all SCCS commands accept options and filenames as arguments. These appear in the SCCS command line immediately after the command name.

An option indicates a special action to be taken by the given SCCS command. An option is usually a lowercase letter preceded by a minus sign (-). Some options require an additional name or value.

A filename indicates the file to be acted on. The syntax for SCCS filenames is like other XENIX filename syntax. Appropriate pathnames must be given if required. Some commands also allow directory names. In this case, all files in the directory are acted on. If the directory contains non-SCCS and unreadable files, these are ignored. A filename must not begin with a minus sign (-).

You can use the special symbol (-) to make the given command read a list of filenames from the standard input. Then, when you process files, you can use these filenames. The list must terminate with an end-of-file character.

Any options that you give with a command apply to all files. The SCCS commands process the options before any filenames, so the options may appear anywhere on the command line.



Filenames are processed left to right. If a command encounters a fatal error, it stops processing the current file; and, if you have given any other filenames, it begins processing the next one.

### 3.2.5 File Administrator

Every SCCS file requires an administrator to maintain and keep the file in order. The administrator is usually the user who created the file and therefore owns it. Before other users can access the file, the administrator must ensure that they have adequate access. Several SCCS commands let the administrator define who has access to the versions in a given *s-file*. These are described in ''Protecting s-files.''

## 3.3 Creating and Using s-files

The *s-file* is the key element in the SCCS system. It provides compact storage for all versions of a given file and automatic maintenance of the relationships between the versions.

This section explains how to use the **admin**, **get**, and **delta** commands to create and use *s-files*. In particular, it describes how to:

- create the first version of a file
- retrieve versions for reading and editing
- save new versions

### 3.3.1 Creating an s-file

You can create an *s-file* from an existing text file using the **-i** (initialize) option of the **admin** command. The command has the following form:

**admin -i***filename* *s.filename*

where:

- **-i** *filename* is the name of the text file from which the *s-file* is to be created, and
- *s.filename* is the name of the new *s-file*.

The name must begin with *s.* and must be unique; no other *s-file* in the same directory can have the same name. For example, suppose the file *demo.c* contains the short C language program:

```
#include <stdio.h>

main ()
{
printf("This is version 1.1 \n");
}
```

To create an *s-file*, type:

```
admin  -idemo.c   s.demo.c
```

This command creates *s.demo.c* and copies the first delta describing the contents of *demo.c* to this new file. The first delta is numbered 1.1.

After you have created an *s-file*, you should remove the original text file using the **rm** command, since that file is no longer needed. If you wish to view the text file or make changes to it, you can retrieve the file using the **get** command described in "Retrieving a File for Editing."

When first creating an *s-file*, the **admin** command may display the warning message:

```
No id keywords(cm7)
```

You can ignore this message unless you have specifically included keywords in your file. For more information, see "Using Identification Keywords."

Note that only a user with write permission in the directory containing the *s-file* can use the **admin** command on that file. This protects the file from administration by unauthorized users.

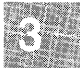
### 3.3.2 Retrieving a File for Reading

You can retrieve a file for reading from a given *s-file* using the **get** command. The command has the following form:

**get** *s.filename ...*

where *s.filename* is the name of an *s-file* containing the text file.

The command retrieves the latest version of the text file and copies it to a regular file. The file has the same name as the *s-file* but with the *s.* removed. It also has read-only file permission. For example, suppose *s.demo.c* contains the first version of the short C program shown in the previous section. To retrieve this program, type:




```
get s.demo.c
```

The command retrieves the program and copies it to the file named *demo.c*. You can then display the file just as you would any other text file.

The command also displays a message which describes the SID of the retrieved file and its size in lines. For example, after retrieving the short C program from *s.demo.c*, the command displays the following message:

```
1.1
6 lines
No id keywords(cm7)
```

You can also retrieve more than one file at a time by giving multiple *s-file* names in the command line. For example, the following command retrieves the contents of *s.demo.c* and *s.def.h* and copies them to the text files *demo.c* and *def.h*:

```
get  s.demo.c  s.def.h
```

When giving multiple *s-file* names in a command, you must separate each with at least one space. When the **get** command displays information about the files, it places the corresponding filename before the relevant information.

### 3.3.3 Retrieving a File for Editing

You can retrieve a file for editing from a given *s-file* using the **-e** (editing) option of the **get** command. The command has the following form:

**get -e** *s.filename ...*

where *s.filename* is the name of an *s-file* containing the text file. To give more than one filename, separate each name with a space.

The command retrieves the latest version of the text file and copies it to an ordinary text file. The file has the same name as the *s-file* but with the *s.* removed. It has read and write file permissions. For example, suppose *s.demo.c* contains the first version of a C program. To retrieve this program, type:

```
get -e s.demo.c
```

The command retrieves the program and copies it to the *demo.c* file. You can now edit the file just as you would any other text file.

If you give more than one filename, the command creates files for each corresponding *s-file*. Since the **-e** option applies to all the files, you can edit each one.

After retrieving a text file, the command displays a message giving the SID of the file and its size in lines. The message also displays a proposed SID, that is, the SID for the new version after editing. For example, after retrieving the six-line C program in *s.demo.c*, the command displays the following message:

```
1.1
new delta  1.2
6 lines
```

The proposed SID is 1.2. If more than one file is retrieved, the corresponding filename precedes the relevant information.

Note that any changes made to the text file are not immediately copied to the corresponding *s-file*. To save these changes, you must use the **delta** command described in the next section. To keep track of the current file version, the **get** command creates another file, called a *p-file*, that contains information about the text file. This file is used by a subsequent **delta** command when saving the new version. The *p-file* has the same name as the *s-file* but begins with a *p.*. The user must not access the *p-file* directly.

### 3.3.4 Saving a New Version of a File

You can save a new version of a text file using the **delta** command. The command has the form:

**delta** *s.filename*

where *s.filename* is the name of the *s-file* from which the modified text file was retrieved. For example, to save changes made to a C program in the file *demo.c* (which was retrieved from the file *s.demo.c*), type:

```
delta s.demo.c
```



Before saving the new version, the **delta** command asks for comments explaining the nature of the changes. It displays the prompt:

```
comments?
```

You can type any appropriate text, up to 512 characters. The comment must end with a Return. If necessary, you can start a new line by typing a backslash (\) followed by a Return. If you do not wish to include a comment, just type a Return.

Once you have given a comment, the command uses the information in the corresponding *p-file* to compare the original version with the new version. It then copies a list of all the changes to the *s-file*. This is the new delta.

After a command has copied the new delta to the *s-file*, it displays a message showing the new SID and the number of lines inserted, deleted, or left unchanged in the new version. For example, if the C program has been changed to the following:

```
#include <stdio.h>

main ()
{
int i = 2;

printf("This is version 1.%d 0, i);
}
```

the command displays the following message:

```
1.2
3 inserted
1 deleted
5 unchanged
```

Once you save a new version, the next **get** command retrieves it. The command ignores previous versions. If you wish to retrieve a previous version, use the **-r** option of the **get** command as described in the next section.

### 3.3.5 Retrieving a Specific Version

You can retrieve any version you wish from an *s-file* using the **-r** (retrieve) option of the **get** command. The command has the following form:

**get** [ **-e** ] **-r***SID* *s.filename* ...

where:

- **-e** is the edit option.
- **-r** *SID* gives the SID of the version to be retrieved.
- *s.filename* is the name of an *s-file* containing the file to be retrieved. You can give more than one filename, if you separate each name with a space.

The command retrieves the given version and copies it to the file having the same name as the *s-file* but with the *s.* removed. The file has read-only permission unless you also give the **-e** option. If you give multiple filenames, the command retrieves one text file of the given version from each. For example, to retrieve version 1.1 from *s.demo.c*, type:

```
get -r1.1 s.demo.c
```

To retrieve a version 1.1 from both *s.demo.c* and *s.def.h*, type:

```
get -e -r1.1 s.demo.c  s.def.h
```

If you give the number of a version that does not exist, the command displays an error message.

You can omit the level number of a version if you wish; that is, just give a release number. The command automatically retrieves the most recent version having the same release number. For example, if the most recent version in the *s.demo.c* file is numbered 1.4, the following command retrieves version 1.4:

```
get -r1 s.demo.c
```

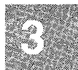
If there is no version with the given release number, the command retrieves the most recent version in the previous release.

### 3.3.6 Changing the Release Number of a File

You can direct the **delta** command to change the release number of a new version of a file using the **-r** option of the **get** command. In this case, the **get** command has the following form:

**get -e -r***rel-num s.filename ...*

where:



- **-e** is the required edit option,
- **-r** *rel-num* is the new release number of the file, and
- *s-filename* is the name of an *s-file* containing the file to be retrieved.

The new release number must be an entirely new number; that is, no existing version can have this number. You can give more than one filename.

The command retrieves the most recent version from the *s-file*, then copies the new release number to the *p-file*. The subsequent **delta** command saves the new version using the new release number and level number 1.

For example, if the most recent version of *s.demo.c* is 1.4, typing the following command causes the subsequent **delta** to save a new version 2.1, not 1.5:

```
get  -e  -r2  s.demo.c
```

The new release number applies to the new version only; the release numbers of previous versions are not affected. Therefore, if you edit version 1.4 (from which 2.1 was derived) and save the changes, you create a new version, 1.5. Similarly, if you edit version 2.1, you create a new version, 2.2.

As before, the **get** command also displays a message showing the current version number, the proposed version number, and the size of the file in lines. Similarly, the subsequent **delta** command displays the new version number and the number of lines inserted, deleted, and unchanged in the new file.

### 3.3.7 Creating a Branch Version

You can create a branch version of a file by editing a version that has been previously edited. A branch version is simply a version whose SID contains a branch and a sequence number.

For example, if version 1.4 already exists, typing the following command retrieves version 1.3 for editing and assigns 1.3.1.1 as the proposed SID:

```
get -e -r1.3 s.demo.c
```

In general, whenever **get** discovers that you wish to edit a version that already has a succeeding version, **get** uses the first available branch and sequence numbers for the proposed SID. For example, if you edit version 1.3 a third time, **get** assigns 1.3.2.1 as the proposed SID.

You can save a branch version just like any other version using the **delta** command.

### 3.3.8 Retrieving a Branch Version

You can retrieve a branch version of a file using the **-r** option of the **get** command. For example, typing the following command retrieves branch version 1.3.1.1:

```
get  -r1.3.1.1  s.demo.c
```

You can retrieve a branch version for editing using the **-e** option of the **get** command. When you are retrieving for editing, **get** creates the proposed SID by incrementing the sequence number by one. For example, if you retrieve branch version 1.3.1.1 for editing, **get** assigns 1.3.1.2 as the proposed SID.

As always, the command displays the version number and file size. If the given branch version does not exist, the command displays an error message.

You can omit the sequence number if you wish. The command retrieves the most recent branch version with the given branch number. For example, if the most recent branch version in *s.def.h* is 1.3.1.4, the following command retrieves version 1.3.1.4:

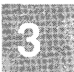
```
get  -r1.3.1  s.def.h
```

### 3.3.9 Retrieving the Most Recent Version

You can always retrieve the most recent version of a file using the **-t** option with the **get** command. For example, type the following to retrieve the most recent version from the file *s.demo.c*:

```
get -t s.demo.c
```

You can combine the **-r** and **-t** options to retrieve the most recent version of a given release number. For example, if the most recent version with release number 3 is 3.5, type the following to retrieve version 3.5:




```
get  -r3  -t  s.demo.c
```

If a branch version exists that is more recent than version 3.5 (such as, 3.2.1.5), then the above command retrieves the branch version and ignores version 3.5.

### 3.3.10 Displaying a Version

You can display the contents of a version at the standard output using the **-p** option of the **get** command. For example, typing the following command displays the most recent version in the file, *s.demo.c*:

```
get -p s.demo.c
```

Similarly, the following command displays version 2.1:

```
get -p -r2.1 s.demo.c
```

The **-p** option is useful for creating *g-files* with user-supplied names. This option also directs all output normally sent to the standard output, such as the SID of the retrieved file, to the standard error file. Thus, the resulting file contains only the contents of the given version. For example, the following command copies the most recent version in *s.demo.c* to the *version.c* file:

```
get  -p s.demo.c  >version.c
```

The command also copies the SID of the file and its size to the standard error file.

### 3.3.11 Saving a Copy of a New Version

The **delta** command normally removes the edited file after saving it in the *s-file*. You can save a copy of this file using the **-n** option of the **delta** command. For example, the following command first saves a new version of *s.demo.c* and then saves a copy of this version in the file *demo.c*:

```
delta  -n  s.demo.c
```

You can display or edit the *demo.c* file as desired, but you cannot edit this file through SCCS.

### 3.3.12 Displaying Helpful Information

An SCCS command displays an error message in the following form whenever it encounters an error in a file:

ERROR [*filename* ]: *message* ( *code* )

where:

- *filename* is the name of the file being processed,
- *message* is a short description of the error, and
- *code* is the error code.

You can use the error code as an argument to the **help** command to display additional information about the error. Type:

help *code*

where *code* is the error code given in an error message.

The command displays one or more lines of text that explain the error and suggest a possible remedy. For example, typing:

```
help col
```

displays the message:

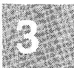
```
col:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

You can use the **help** command at any time.

## 3.4 Using Identification Keywords

The SCCS system provides several special symbols called *identification keywords* that you can use in the text of a program or document to represent a predefined value. Keywords represent a wide range of values, from the creation date and time of a given file to the name of the module containing the keyword. When you retrieve the file for reading, the SCCS system automatically replaces any keywords it finds in a given version of a file with the keyword's value.

This section explains how keywords are treated by the various SCCS commands and how you can use the keywords in your own files. Only a few keywords are described in this section. For a complete list of the keywords, see *get*(CP) in the *XENIX Programmer's Reference*.




### 3.4.1 Inserting a Keyword into a File

You can insert a keyword into any text file. A keyword is simply an uppercase letter enclosed in percent signs (%); it requires no special characters. For example, **%I%** is the keyword representing the SID of the current version, and **%H%** is the keyword representing the current date.

When you retrieve a program for reading using the **get** command, the command replaces the keywords with their current values. For example, the **%M%**, **%I%**, and **%H%** keywords are used in place of the module name, the SID, and the current date in a program statement:

```
char header[] = {" %M%  %I%  %H% "};
```

The **get** command expands these keywords in the retrieved version of the program:

```
char header[] = {" MODNAME  2.3  07/07/77 "};
```

The **get** command does not replace keywords when retrieving a version for editing. The system assumes that you want to keep the keywords (and not their values) when you save the new version of the file.

To indicate that a file has no keywords, the **get**, **delta**, and **admin** commands display the following message:

```
No id keywords (cm7)
```

This message is normally a warning, letting you know that no keywords are present. However, you can change the operation of the system to make this a fatal error, as explained later in this chapter.

### 3.4.2 Assigning Values to Keywords

The system predefines values of most keywords, but you can define some keywords explicitly, such as the value for the %M% keyword. To assign a value to a keyword, you must set the corresponding *s-file* flag to the desired value using the **-f** option of the **admin** command.

For example, to set the %M% keyword to **cdemo**, set the **m** flag as in the following command:

```
admin -fmcdemo s.demo.c
```

This command records **cdemo** as the current value of the %M% keyword. Note that if you do not set the **m** flag, the SCCS system uses the name of the original text file for %M% by default.

The **t** and **q** flags are also associated with keywords. You will find a description of these flags and their corresponding keywords in *get*(CP) in the *XENIX Programmer's Reference*. You can change keyword values at any time.

### 3.4.3 Forcing Keywords

If a version contains no keywords, you can force a fatal error by setting the **i** flag in the given *s-file*. The flag causes the **delta** and **admin** commands to stop processing of the given version and report an error. You can use the flag to ensure that keywords are used properly in a given file.

To set the **i** flag, use the **-f** option of the **admin** command. For example, typing the following command sets the **i** flag in *s.demo.c*:

```
admin -fi s.demo.c
```

If the given version does not contain keywords, subsequent **delta** or **admin** commands that access this file print an error message.

Note that if you attempt to set the **i** flag at the same time as you create an *s-file* and if the initial text file contains no keywords, then the **admin** command displays a fatal error message and stops without creating the *s-file*.

## 3.5 Using s-file Flags

An *s-file* flag is a special value that defines how a given SCCS command will operate on the corresponding *s-file*. The *s-file* flags are stored in the *s-file* and each SCCS command reads these flags before it operates on the file. The *s-file* flags affect operations such as keyword checking, keyword replacement values, and default values for commands.

This section explains how to set and use *s-file* flags. It also describes the action of commonly used flags. For a complete description of all flags, see **admin**(CP) in the *XENIX Programmer's Reference*.



### 3.5.1 Setting s-file Flags

You can set the flags in a given *s-file* using the **-f** option of the **admin** command. The command has the following form:

**admin -f***flag s.filename*

where:

- **-f***flag* is the flag to be set, and
- *s.filename* is the name of the *s-file* in which the flag is to be set.

For example, typing the following command sets the **i** flag in *s.demo.c*:

```
admin  -fi  s.demo.c
```

Note that some *s-file* flags take values when they are set. For example, the **m** flag requires that you give a module name. When a value is required, it must immediately follow the flag name, as in this command which sets the **m** flag to the module name *dmod*:

```
admin -f mdmod  s.demo.c
```

### 3.5.2 Using the i Flag

If no keywords are found in the given text file, the **i** flag causes the **admin** and **delta** commands to print a fatal error message and stop. The flag prevents a version of a file containing expanded keywords from being saved as a new version. (Saving an expanded version destroys the keywords for all subsequent versions.)

When you set the **i** flag, each new version of a file must contain at least one keyword. Otherwise, you cannot save the version.

### 3.5.3 Using the d Flag

The **d** flag gives the default SID for versions retrieved by the **get** command. The flag takes an SID as its value. For example, the following command sets the default SID to 1.1:

```
admin -fd1.1 s.demo.c
```

A subsequent **get** command which does not use the **-r** option will retrieve version 1.1.

### 3.5.4 Using the v Flag

The **v** flag lets you include modification requests in an *s-file*. Modification requests are names or numbers that you use as a shorthand means of indicating the reason for each new version.

When you set the **v** flag, the **delta** command asks for the modification requests just before asking for comments. The **v** flag also lets you use the **-m** option in the **delta** and **admin** commands.

### 3.5.5 Removing an s-file Flag

You can remove an *s-file* flag from an *s-file* using the **-d** option of the **admin** command:

**admin -d***flag* *s.filename*

where:

- **-d***flag* is the name of the flag to be removed, and
- *s.filename* is the name of the *s-file* from which the flag is to be removed.

For example, the following command removes the **i** flag from *s.demo.c*:

```
admin -di s.demo.c
```

When you are removing a flag that takes a value, only the flag name is required. For example, the following command removes the **m** flag from the *s-file*:

```
admin -dm s.demo.c
```

You must not use the **-d** and **-i** options at the same time.

## 3.6 Modifying s-file Information

Every *s-file* contains information about the deltas it contains. Normally, the SCCS commands maintain this information, so it is not directly accessible to you. Some information, however, is specific to the user who creates the *s-file* and can be changed to meet the user's requirements. This information is in two special parts of the *s-file* called the *delta table* and the *description field.*



The delta table contains information about each delta, such as the SID and the date and time of creation. It also contains user-supplied information, such as comments and modification requests.

The description field contains a user-supplied description of the *s-file* and its contents. You can change or delete both parts at any time to reflect changes to the *s-file* contents.

### 3.6.1 Adding Comments

You can add comments to an *s-file* using the **-y** option of the **delta** and **admin** commands. This option copies the given text to the *s-file* as the comment for the new version. The comment can be any combination of letters, digits, and punctuation symbols, but no embedded Returns are allowed. If you use spaces, enclose the comment in double quotes. The complete command must fit on one line. For example, the following command saves the comment ''George Wheeler'' in *s.demo.c*:

```
delta -y"George Wheeler"  s.demo.c
```

Typically, you use the **-y** option in shell procedures as part of an automated approach to maintaining files. When you do use the **-y** option, the **delta** command does not print the corresponding comment prompt, so no interaction is required. If you give more than one *s-file* in the command line, the given comment applies to all of them.

### 3.6.2 Changing Comments

You can change the comments in a given *s-file* using the **cdc** command. The command has the following form:

**cdc -r***SID s.filename*

where:

- **-r***SID* is the SID of the version whose comment is to be changed.
- *s.filename* is the name of the *s-file* containing the version.

The command asks for a new comment by displaying the following prompt:

```
comments?
```

You can type any sequence of characters up to 512. The sequence can contain embedded Returns if they are preceded by a backslash (\). You must terminate the sequence with a Return. For example, the following command prompts for a new comment for version 3.4:

```
cdc  -r3.4  s.demo.c
```

Although the command does not delete the old comment, it is no longer directly accessible to you. The new comment contains the log-in name of the user who invoked the **cdc** command and the time the comment was changed.

### 3.6.3 Adding Modification Requests

You can add modification requests to an *s-file* when the **v** flag is set, using the **-m** option of the **delta** and **admin** commands. A *modification request* is a shorthand method of describing the reason for a particular version. Modification requests are usually names or numbers that you choose to represent specific requests.

The **-m** option causes the given command to save the requests following the option. A request can be any combination of letters, digits, and punctuation symbols. If you give more than one request, you must separate each with a space and enclose the request in double quotes. For example, the following command copies the requests ''error35'' and ''optimize10'' to *s.demo.c*, while saving the new version:

```
delta -m"error35 optimize10" s.demo.c
```

The **delta** command does not prompt for modification requests if you use the **-m** option.

When you use the **-m** option with the **admin** command, you must combine it with the **-i** option. Also, you must set the **v** flag with the **-f** option. For example, the following command inserts the modification request "error0" in the new *s.def.h*: file

```
admin  -idef.h  -m"error0"  -fv  s.def.h
```

### 3.6.4 Changing Modification Requests



You can change modification requests when the **v** flag is set, using the **cdc** command. The command asks for a list of modification requests by displaying the prompt:

```
MRs?
```

You can type any number of requests. Each request can have any combination of letters, digits, or punctuation symbols, but no more than 512 characters are allowed; you must terminate the last request with a Return. To remove a request, you must precede the request with an exclamation point (!). For example, the following command asks for changes to the modification requests:

```
cdc -r1.4  s.demo.c
```

The following response adds the request "error36" and removes "error35":

```
MRs? error36 !error35
```

### 3.6.5 Adding Descriptive Text

You can add descriptive text to an *s-file* using the **-t** option of the **admin** command. *Descriptive text* is any text that describes the reason for the given *s-file*. Descriptive text is independent of the contents of the *s-file* and you can display it only by using the **prs** command.

The **-t** option directs **admin** to copy the contents of a given file into the description field of the *s-file*. The command has the following form:

**admin -t***filename s.filename*

where:

- **-t** *filename* gives the name of the file containing the descriptive text, and
- *s.filename* is the name of the *s-file* to receive the descriptive text.

The file to be inserted can contain any amount of text. For example, the following command inserts the contents of the *cdemo* file into the description field of *s.demo.c*:

```
admin -tcdemo s.demo.c
```

You can also use the **-t** option to initialize the description field when creating the *s-file*. For example, the following command inserts the contents of the *cdemo* file into *s.demo.c*:

```
admin -idemo.c -tcdemo s.demo.c
```

If you do not use **-t**, the description field of the new *s-file* is left empty.

You can remove the current descriptive text in an *s-file* using the **-t** option without a filename. For example, the following command removes the descriptive text from the *s-file*, *s.demo.c*:

```
admin  -t  s.demo.c
```

## 3.7 Printing from an s-file

This section explains how to use the **prs** command to display information contained in an *s-file*. The **prs** command has a variety of options that control the display format and content.

### 3.7.1 Using a Data Specification

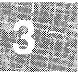
You can define explicitly the information to be printed from an *s-file* using the **-d** option of the **prs** command. The command copies user-specified information to the standard output. The command has the following form:

**prs -d***spec* *s.filename*

where:

- **-d***spec* is the data specification.
- *s.filename* is the name of the *s-file* from which the information is to be taken.

The data specification is a string of data keywords and text. A data keyword is an uppercase letter enclosed in colons (:). It represents a value contained in the given *s-file*. For example, the **:I:** keyword represents the SID of a given version, **:F:** represents the filename of the given *s-file*, and **:C:** represents the comment line associated with a given version. These values replace data keywords when the information is printed.




For example, typing:

```
prs  -d" version:  :I:   filename:  :F: "  s.demo.c
```

may produce the line:

```
version:  2.1   filename:  s.demo.c
```

For a complete list of data keywords, see *prs*(CP) in the *XENIX Programmer's Reference*.

### 3.7.2 Printing a Specific Version

You can print information about a specific version in a given *s-file* using the **-r** option of the **prs** command. The command has the following form:

**prs -r***SID  s.filename*

where:

- **-r** *SID* is the SID of the desired version, and
- *s.filename* is the name of the s-file containing the version.

For example, the following command prints information about version 2.1 in *s.demo.c*:

```
prs -r2.1 s.demo.c
```

If you do not specify the **-r** option, the command prints information about the most recently created delta.

### 3.7.3 Printing Later and Earlier Versions

You can print information about a group of versions using the **-l** and **-e** options of the **prs** command. The **-l** option causes the command to print information about all versions immediately after the given version. The **-e** option causes the command to print information about all versions immediately preceding the given version. For example, the following command prints information about all versions that precede version 1.4 (such as, 1.3, 1.2, and 1.1):

```
prs  -r1.4  -e  s.demo.c
```

The following command prints information about versions that succeed version 1.4 (e.g., 1.5, 1.6, and 2.1):

```
prs  -r1.4  -l  s.abc
```

If you specify both options, **prs** prints information about all versions.

## 3.8 Editing by Several Users

The following sections explain how to perform concurrent editing and how to save edited versions when you have retrieved more than one version for editing.

### 3.8.1 Editing Different Versions

The SCCS system lets several different versions of a file be edited at the same time. This means that you can edit version 2.1 while another user edits version 1.1. There is no limit to the number of versions that can be edited at any given time.

When several users edit different versions concurrently, all users must begin work in their own directories If users attempt to share a directory and work on versions from the same *s-file* at the same time, the **get** command will refuse to retrieve a version.

### 3.8.2 Editing a Single Version

You can allow a single version of a file to be edited by more than one user by setting the **j** flag in the given *s-file*. The flag causes the **get** command to check the *p-file* and create a new proposed SID if the given version is already being edited.

You can set the flag using the **-f** option of the **admin** command. For example, the following command sets the flag for *s.demo.c*:

```
admin -fj s.demo.c
```

When you set the **-f** flag, the **get** command uses the next available branch SID for each new proposed SID. For example, suppose a user retrieves version 1.4 in the *s.demo.c* file, and the proposed version is 1.5. If another user retrieves version 1.4 for editing before the first user has saved changes, the proposed version for the new user will be 1.4.1.1, since version 1.5 is already proposed and likely to be taken. In no case will a version edited by two separate users result in a single new version.

### 3.8.3 Saving a Specific Version

When editing two or more versions of a file, you can direct the **delta** command to save a specific version, using the **-r** option to give the SID of that version. The command has the following form:

**delta -r***SID* *s.filename*

where:

- **-r** *SID* is the SID of the version being saved, and
- *s.filename* is the name of the *s-file* to receive the new version.

The *SID* can be the SID of the version you have just edited or the proposed SID for the new version. For example, if you have retrieved version 1.4 for editing (and no version 1.5 exists), then either of the following commands saves version 1.5:

```
delta -r1.5 s.demo.c
```

or:

```
delta -r1.4 s.demo.c
```

## 3.9 Protecting s-files

The SCCS system uses the normal XENIX system file permissions to protect *s-files* from changes by unauthorized users. In addition to the XENIX system protections, the SCCS system provides two ways to protect the *s-files*: the *user list* and the *protection flags*. The user list is a list of log-in names and group IDs of users who are allowed to access the *s-file* and create new versions of the file. The protection flags are three special *s-file*

flags that define which versions are currently accessible to otherwise authorized users. The following sections explain how to set and use the user list and protection flags.

### 3.9.1 Adding a User to the User List

You can add a user or a group of users to the user list of a given *s-file* using the **-a** option of the **admin** command. The option adds the given name to the user list. The user list defines who may access and edit the versions in the *s-file*. The command has the following form:

**admin -a***name s.filename*

where:

- **-a***name* gives the log-in name of the user or the group name of a group of users to be added to the list.
- *s.filename* gives the name of the *s-file* to receive the new users.

For example, the following command adds the users *johnd* and *suex* and the group *marketing* to the user list of *s.demo.c*:

```
admin -ajohnd -asuex -amarketing s.demo.c
```

If you create an *s-file* without giving the **-a** option, the user list remains empty, and all users may access and edit the files. When you explicitly give a user name or names, only those users can access the files.

### 3.9.2 Removing a User from a User List

You can remove a user or a group of users from the user list of a given *s-file* using the **-e** option of the **admin** command. The command has the following form:

**admin -e***name s.filename*

where:

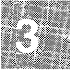
- **-e***name* is the log-in name of a user or the group name of a group of users, to be removed from the list.
- *s.filename* is the name of the *s-file* from which the names are to be removed.

For example, the following command removes the user *johnd* and the group *marketing* from the user list of *s.demo.c*:

```
admin -ejohnd -emarketing s.demo.c
```

### 3.9.3 Setting the Floor Flag

The floor flag **f** defines the release number of the lowest version that you can edit in a given *s-file*. You can set the flag using the **-f** option of the **admin** command. For example, the following command sets the floor to release number 2:



```
admin -ff2 s.demo.c
```

If you attempt to retrieve any versions with a release number less than 2, an error will result.

### 3.9.4 Setting the Ceiling Flag

The ceiling flag **c** defines the release number of the highest version that you can edit in a given *s-file*. You can set the flag using the **-f** option of the **admin** command. For example, the following command sets the ceiling to release number 5:

```
admin -fc5 s.demo.c
```

If you attempt to retrieve any versions with a release number greater than 5, an error will result.

### 3.9.5 Locking a Version

The lock flag **l** lists, by release number, all versions in a given *s-file* that are locked against further editing. You can set the flag using the **-f** flag of the **admin** command, followed by one or more release numbers. You must separate multiple release numbers with commas. For example, the following command locks all versions with release number 3 against further editing:

```
admin -fl3 s.demo.c
```

The following command locks all versions with release numbers 4, 5, and 9:

```
admin -fl4,5,9 s.def.h
```

Note that the special symbol **a** can be used to specify all release numbers. The following command locks all versions in the *s.demo.c* file:

```
admin -fla  s.demo.c
```

## 3.10 Repairing SCCS Files

The SCCS system carefully maintains all SCCS files. However, damage can result from hardware malfunctions copying incorrect information to the file. The following sections explain how to check for damage to SCCS files and how to repair the damage or regenerate the file.

### 3.10.1 Checking an s-file

You can check a file for damage using the **-h** option of the **admin** command. This option computes the checksum of the given *s-file* and compares it with the existing sum. An *s-file* checksum is an internal value computed from the sum of all bytes in the file. If the new and existing checksums are not equal, the command displays the following message which indicates damage to the file:

```
corrupted file (co6)
```

For example, the following command checks the file *s.demo.c* for damage by generating a new checksum for the file, and comparing the new sum with the existing sum:

```
admin  -h  s.demo.c
```

You can give more than one filename. The command checks each file in turn. You may also give the name of a directory, in which case, the command checks all files in the directory.

Since failure to repair a damaged *s-file* can destroy the file's contents or make the file inaccessible, it is a good idea to check all *s-files* regularly for damage.

### 3.10.2 Editing an s-file

When you find a damaged *s-file*, you should restore a backup copy of the file from a backup disk rather than attempting to repair the file. (Restoring a backup copy of a file is described in the *XENIX Operations Guide.*) If this is not possible, you can edit the file with a XENIX text editor.

To repair a damaged *s-file*, use the description of an *s-file* given in **sccsfile**(F) in the *XENIX User's Reference* to locate the part of the file that is damaged. Use extreme care when making changes; small errors often cause unwanted results.

3

### 3.10.3 Changing an s-file's Checksum

After repairing a damaged *s-file*, you must change the file's checksum by using the **-z** option of the **admin** command. For example, to restore the checksum of the repaired file *s.demo.c*, type:

```
admin  -z  s.demo.c
```

The command computes and saves the new checksum, replacing the old sum.

### 3.10.4 Regenerating a g-file for Editing

You can create a *g-file* for editing without affecting the current contents of the *p-file* using the **-k** option of the **get** command. The option has the same affect as the **-e** option, except that the current contents of the *p-file* remain unchanged. Typically, you use this option to regenerate a *g-file* that has been removed accidentally or destroyed before it has been saved by the **delta** command.

### 3.10.5 Restoring a Damaged p-file

You can use the **-g** option of the **get** command to generate a new copy of a *p-file* that has been removed accidentally. For example, the following command creates a new *p-file* entry for the most recent version in *s.demo.c*:

```
get  -e  -g  s.demo.c
```

If *demo.c* already exists, it will not be changed by this command.

## 3.11 Using Other Command Options

Many of the SCCS commands provide options that control their operation in useful ways. This section describes these options and explains how you can use them to perform useful work.

### 3.11.1 Getting Help With SCCS Commands

You can display helpful information about an SCCS command by giving the name of the command as an argument to the **help** command. The **help** command displays a short explanation of the command and its syntax. For example, typing:

```
help rmdel
```

displays the following message:

```
rmdel:
     rmdel  -rSID name
```

### 3.11.2 Creating a File With the Standard Input

You can direct **admin** to use the standard input as the source for a new *s-file* using the **-i** option without a filename. For example, the following command causes **admin** to create *s.demo.c*, using the *demo.c* text file as its first version:

```
admin -i s.demo.c <demo.c
```

This method of creating a new *s-file* is used typically to connect **admin** to a pipe. For example, the following command creates a new *s.mod.c*, which contains the first version of the concatenated files *mod1.c* and *mod2.c*:

```
cat mod1.c mod2.c | admin -i s.mod.c
```

### 3.11.3 Starting at a Specific Release

Normally, the **admin** command starts numbering versions with release number 1. You can direct the command to start with any given release number using the **-r** option. The command has the following form:

**admin -r***rel-num s.filename*

where:

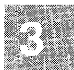
- **-r** *rel-num* is the value of the starting release number.
- *s.filename* is the name of the *s-file* to be created.

For example, the following command starts with release number 3:

```
admin  -idemo.c  -r3  s.demo.c
```

The first version is 3.1.

### 3.11.4 Adding a Comment to the First Version



You can add a comment to the first version of a file using the **-y** option of the **admin** command when creating the *s-file*. For example, the following command inserts the comment *George Wheeler* in *s.demo.c*:

```
admin -idemo.c -y"George Wheeler" s.demo.c
```

The comment can be any combination of letters, digits, and punctuation symbols. If you use spaces, enclose the comment in double quotes. The complete command must fit on one line.

If you do not use the **-y** option when creating an *s-file*, a comment of the following form is inserted automatically:

```
date  and  time  created  YY/MM/DD HH:MM:SS  by  username
```

where:

- *YY/MM/DD HH:MM:SS* are the date and time the file was created, and
- *username* is the login name of the user who created the file.

### 3.11.5 Suppressing Normal Output

You can suppress the normal display of messages created by the **get** command using the **-s** option. This option prevents information, such as the SID of the retrieved file, from being copied to the standard output. The option does not suppress error messages.

The **-s** option is often used with the **-p** option to pipe the output of the **get** command to other commands. For example, the following command copies the most recent version of *s.demo.c* to the lineprinter:

```
get  -p  -s  s.demo.c  | lpr
```

You can also suppress the normal output of the **delta** command using the **-s** option. This option suppresses all output normally directed to the standard output, except for the comment prompt.

### 3.11.6 Including and Excluding Deltas

When creating a *g-file*, you can define explicitly which deltas you want to include and exclude by using the **-i** and **-x** options of the **get** command:

**-i** causes the command to apply the given deltas when constructing a version.

**-x** causes the command to ignore the given deltas when constructing a version.

Both options must be followed by one or more SIDs. If you supply multiple SIDs, you must separate them with commas. For example, the following command constructs the *g-file* using deltas 1.2 and 1.3:

```
get -i1.2,1.3 s.demo.c
```

The **-i** option is useful if you wish to apply changes automatically to a version while retrieving it for editing. For example, the following command retrieves version 3.3 for editing:

```
get -e -i4.1 -r3.3  s.demo.c
```

When **get** retrieves the file, the changes in delta 4.1 are applied to it automatically, making the *g-file* the same as if version 3.3 had been edited by hand using the changes in delta 4.1. You can save these changes immediately by issuing a **delta** command. No editing is required.

The **-x** option is useful for removing changes performed on a given version. For example, the following command retrieves version 1.6 for editing:

```
get -e -x1.5 -r1.6 s.demo.c
```

When **get** retrieves the file, the changes in delta 1.5 are left out of it automatically, making the *g-file* the same as if version 1.4 had been changed according to delta 1.6 (with no intervening delta 1.5). You can save these changes immediately by issuing a **delta** command. No editing is required.

When you include or exclude deltas using the **-i** and **-x** options, **get** compares them with the deltas normally used in constructing the given version. If two deltas attempt to change the same line of the retrieved file, the command displays a warning message. The message shows the range of lines in which the problem may exist. Corrective action, if required, is your responsibility.



### 3.11.7 Listing the Deltas of a Version

Using the **-l** option, you can create a table showing the deltas required to create a given version. This option causes the **get** command to create an *l-file* containing the SIDs of all deltas used to create the given version. Typically, you use this option to create a history of a given version's development. For example, the following command creates a file *l.demo.c* containing the deltas required to create the most recent version of *demo.c*:

```
get -l s.demo.c
```

You can display the list of deltas required to create a version using the **-lp** option. The option performs the same function as the **-l** option except it copies the list to the standard output file. For example, the following command copies the list of deltas required to create version 2.3 of *demo.c* to the standard output:

```
get  -lp  -r2.3  s.demo.c
```

Note that you can combine the **-l** option with the **-g** option to create a list of deltas without retrieving the actual version.

### 3.11.8 Mapping Lines to Deltas

You can map each line in a given version to its corresponding delta by using the **-m** option of the **get** command. This option precedes each line in a *g-file* with the SID of the delta that caused that line to be inserted. A TAB character separates the SID from the beginning of the line. The **-m** option is typically used to review the history of each line in a given version.

### 3.11.9 Naming Lines

You can name each line in a given version with the current module name (that is, the value of the **%M%** keyword) using the **-n** option of the **get** command. This option precedes each line of the retrieved file with the value of the **%M%** keyword and a TAB character.

Often you will use the **-n** option to indicate that a given line is from the given file. When you specify both the **-m** and the **-n** options, each line begins with the **%M%** keyword.

### 3.11.10 Displaying a List of Differences

You can display a detailed list of the differences between a new version of a file and the previous version using the **-p** option of the **delta** command. This option causes **delta** to display the differences in a format similar to the output of **diff**(C).

### 3.11.11 Comparing SCCS Files

You can compare two versions from a given *s-file* using the **sccsdiff** command. This command prints on the standard output the differences between two versions of the s-file. The command has the following form:

**sccsdiff -r***SID1* **-r***SID2* *s.filename*

where:

- **-r** *SID1* and **-r** *SID2* are the SIDs of the versions to be compared, and
- *s.filename* is the name of the *s-file* containing the versions.

The SID versions must be given in the order in which they were created. For example, the following command displays the differences between versions 3.4 and 5.6:

```
sccsdiff -r3.4 -r5.6  s.demo.c
```

The differences are displayed in a form similar to the **diff**(C) command.

### 3.11.12 Checking a File Version

You can display information about a given version using the **-g** option of the **get** command. This option suppresses the actual retrieval of a version and displays only the most recent version's SID.

You can also use the **-g** with the **-r** option to check for the existence of a given version. For example, the following command displays the SID for this version of *s.demo.c*, if it exists:

```
get -g -r4.3 s.demo.c
```

If the version does not exist, the command displays an error message.



### 3.11.13 Removing a Delta

You can remove a delta from an *s-file* using the **rmdel** command. The command has the following form:

**rmdel -r***SID s.filename*

where:

- **-r** *SID* is the SID of the delta to be removed, and
- *s.filename* is the name of the *s-file* from which the delta is to be removed.

The delta must be the most recently created delta in the *s-file*. Furthermore, you must have write permission in the directory containing the *s-file,* and must own the *s-file* or be the user who created the delta.

For example, the following command removes delta 2.3 from *s.demo.c*:

```
rmdel  -r2.3  s.demo.c
```

The **rmdel** command will not remove a protected delta, that is, a delta whose release number is below the current floor value, above the current ceiling value, or equal to a current locked value. (For more information, see ''Protecting s-files.'') The command will also refuse to remove a delta which is currently being edited.

Reserve the **rmdel** command for those cases in which incorrect global changes were made to an *s-file*.

Note that **rmdel** changes the type indicator of the given delta from *D* (existing) to *R* (removed). A type indicator defines the type of delta. Type indicators are described in more detail in *delta*(CP) in the *XENIX Programmer's Reference*.

### 3.11.14 Searching for Strings

You can search for strings in files created from an *s-file* using the **what**(C) command. This command searches for the @(#) symbol (the current value of the **%Z%** keyword) in the given file. It then prints, on the standard output, all text immediately following the symbol, up to the next double quote ("), greater than (>), backslash (\), Return, or non-printing character. For example, if *s.demo.c* contains the line:

```
char  id[] = "%Z%%M%:%I%";
```

and the following command is executed:

```
get  -r3.4  s.prog.c
```

then the following command:

```
what  prog.c
```

displays:

```
prog.c:
       prog.c:3.4
```

You can also use the **what** command to search files that have not been created by SCCS commands.

# Chapter 4

# lint: A C Program Checker


4.1 Introduction 4-1

4.2 Invoking lint 4-1

4.3 Options 4-2

4.4 Checking for Unused Variables and Functions 4-3

4.5 Checking Local Variables 4-4

4.6 Checking for Unreachable Statements 4-5

4.7 Checking for Infinite Loops 4-6

4.8 Checking Function Return Values 4-7

4.9 Checking for Unused Return Values 4-7

4.10 Checking Types 4-8

4.11 Checking Type Casts 4-9

4.12 Checking for Nonportable Character Use 4-9

4.13 Checking for Assignment of longs to ints 4-10

4.14 Checking for Strange Constructions 4-10

4.15 Checking for Use of Older C Syntax 4-11

4.16 Checking Pointer Alignment 4-12

4.17 Checking Expression Evaluation Order 4-13

4.18 Embedding Directives 4-13

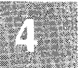
4.19 Checking For Library Compatibility 4-15

## 4.1 Introduction

This chapter explains how to use the C program checker **lint**(CP). The program examines C source files and warns of errors or misconstructions that may cause errors during compilation of the file or during execution of the compiled file.

In particular, **lint** checks for:

- unused functions and variables
- unknown values in local variables
- unreachable statements and infinite loops
- unused and misused return values
- inconsistent types and type casts
- mismatched types in assignments
- nonportable and old-fashioned syntax
- strange constructions
- inconsistent pointer alignment and expression evaluation order




The **lint** program and the C compiler are generally used together to check and compile C language programs. Although the C compiler rapidly and efficiently compiles C language source files, it does not perform the sophisticated type- and error- checking required by many programs. The **lint** program, on the other hand, provides thorough checking of source files without compiling.

## 4.2 Invoking lint

You can invoke **lint** by typing its name at the shell command line. The command has the form:

lint [*option...*] *filename* ... *lib* ...

where *option* is a command option that defines how the checker should operate, *filename* is the name of the C language source file to be checked, and *lib* is the name of a library to check. You can give more than one option, filename, or library name in the command as long as you use spaces to separate them. If you give two or more filenames, **lint** assumes that the files form a complete program and checks the files accordingly. For example, the command:

```
lint main.c add.c
```

treats *main.c* and *add.c* as two parts of a complete program.

If **lint** discovers errors or inconsistencies in a source file, it produces messages describing the problem. The messages have the form:

*filename* (*num*): *description*

where *filename* is the name of the source file containing the problem, *num* is the number of the line in the source containing the problem, and *description* is a description of the problem. For example, the message:

```
main.c (3): warning: x unused in function main
```

shows that the variable *x*, defined in line three of the source file *main.c*, is not used anywhere in the file.

## 4.3 Options

The options available to you may be classed into two categories: those that instruct **lint** to suppress certain kinds of complaints, and those that alter the behavior of **lint**. The following list summarizes both kinds.

### Suppressive Options

- a Suppresses complaints about assignments of long values to variables that are not long.

- b Suppresses complaints about **break** statements that cannot be reached. (Programs produced by **lex** or **yacc** will often result in a large number of such complaints.)

- c Suppresses complaints about casts that have questionable portability.

- h Does not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

- u Suppresses complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running **lint** on a subset of files of a larger program.)

- v Suppresses complaints about unused arguments in functions.

- x Does not report variables referred to by external declarations but never used.

### Other Options

4

- n Does not check compatibility against either the standard or the portable lint library.

- p Attempts to check portability to other dialects of C.

- l*library* Checks function definitions in the specified lint *library*. For example, **-lm** causes the library *llibm.ln* to be searched.

## 4.4 Checking for Unused Variables and Functions

The **lint** program checks for unused variables and functions by seeing if each declared variable and function is used at least once in the source file. The program considers a variable or function used if the name appears in at least one statement. It is not considered used if it only appears on the left side of an assignment. For example, in the following program fragment:

```
main ()
{
        int x,y,z;

        x=1; y=2; z=x+y;
```

the variables *x* and *y* are considered used, but variable *z* is not.

Unused variables and functions often occur during the development of large programs. It is not uncommon for a programmer to remove all references to a variable or function from a source file, but forget to

remove its declaration. Such unused variables and functions rarely cause working programs to fail, but do make programs harder to understand and change. Checking for unused variables and functions can also help you find variables or functions that you intended to use but have accidentally left out of the program.

Note that the **lint** program does not report a variable or function unused if it is explicitly declared with the **extern** storage class. Such a variable or function is assumed to be used in another source file.

You can direct **lint** to ignore all the external declarations in a source file by using the **-x** (for "external") option. This option causes the program checker to skip any line that begins with the **extern** storage class. The **-x** option is typically used to save time when checking a program, especially if all external declarations are known to be valid.

Some programming styles require functions that perform closely related tasks to have the same number and type of arguments, regardless of whether these arguments are used. Under normal operation, **lint** reports any argument not used as an unused variable. You can direct **lint** to ignore unused arguments by using the **-v** option.

The **-v** option causes **lint** to ignore all unused function arguments except for those declared with **register** storage class. The program considers unused arguments of this class to be a preventable waste of the register resources of the computer.

You can direct **lint** to ignore all unused variables and functions by using the **-u** (for "unused") option. This option prevents **lint** from reporting variables and functions it considers unused.

The **-u** option is typically used when checking a source file that contains just a portion of a large program. Such source files usually contain declarations of variables and functions that are intended to be used in other source files and are not explicitly used within the file. Since **lint** can only check the given file, it assumes that such variables or functions are unused and and reports them as errors whenever the **-u** option is not given.

## 4.5 Checking Local Variables

The **lint** program checks all local variables to ensure that they are set to a value before being used. Since local variables have either automatic or register storage class, their values at the start of the program or function cannot be known. Using such a variable before assigning a value to it is an error.

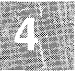
The **lint** program checks the local variables by searching for the first assignment in which the variable receives a value, and for the first statement or expression in which the variable is used. If the first assignment appears later than the first use, **lint** considers the variable inappropriately used. For example, in the program fragment:

```
char c;

if ( c != EOF )
     c = getchar();
```

**lint** warns that the the variable *c* is used before it is assigned.

If a variable is used in the same statement in which it is assigned for the first time, **lint** determines the order of evaluation of the statement and displays an appropriate message. For example, in the program fragment

```
int i,total;

scanf("%d", &i);
total = total + i;
```



**lint** warns that the variable *total* is used before it is set, since it appears on the right side of the same statement that assigns its first value.

Static and external variables are always initialized to zero before program execution begins, so **lint** does not report such variables if they are used before being set to a value.

## 4.6 Checking for Unreachable Statements

The **lint** program checks for unreachable statements. Unreachable statements are unlabeled statements that immediately follow a **goto**, **break**, **continue**, or **return** statement. During execution of a program, the unreachable statements never receive execution control and therefore are considered wasteful. For example, in the program fragment:

```
int x,y;

return (x+y);
exit (1);
```

the function call **exit** after the return statement is unreachable.

Unreachable statements are common when you are developing programs containing large case constructions, or loops containing break and continue statements. Such statements are wasteful and should be removed when convenient.

During normal operation, **lint** reports all unreachable break statements. These are relatively common (in fact, some programs created by the **yacc** and **lex** programs contain hundreds), so it may be desirable to suppress these reports. You can direct **lint** to suppress the reports by using the **-b** option.

Note that **lint** assumes that all functions eventually return control, so it does not report as unreachable any statement that follows a function that takes control and never returns it. For example, in the program fragment:

```
exit (1);
return;
```

the call to **exit** causes the **return** statement to become an unreachable statement, but **lint** does not report it as such.

## 4.7 Checking for Infinite Loops

The **lint** program checks for infinite loops and for loops that are never executed. For example, the statements:

```
while (1) { }
```

and:

```
for (;;) {}
```

are both considered infinite loops. The statements:

```
while (0) { }
```

and:

```
for (0;0;) { }
```

will be reported as never executed.

Although some valid programs have such loops, they are generally considered errors.

## 4.8 Checking Function Return Values

The **lint** program checks to ensure that a function returns a meaningful value if a return value is expected. Some functions return values that are never used. Some programs incorrectly use function values that have never been returned. So **lint** addresses these problems in a number of ways.

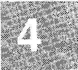
Within a function definition, the appearance of both:

```
return (expr);
```

and:

```
return ;
```

statements is cause for alarm. In this case, **lint** produces the following error message:




`warning: function` *filename* has return(e); and return;

It is difficult to detect when a function return is implied by the flow of control reaching the end of the given function. This is demonstrated with a simple example:

```
f(a)
{
        if(a)
                return (3);
        g ();
}
```

If *a* is false, then *f*() will call the function *g*() and then return with no defined return value. This will trigger a report from **lint**. If *g*(), like *exit*( ), never returns, the message will still be produced when in fact nothing is wrong. In practice, potentially serious bugs can be discovered with this feature. It also accounts for a substantial fraction of the undeserved error messages produced by **lint**.

## 4.9 Checking for Unused Return Values

The **lint** program checks for cases where a function returns a value, but the value is rarely, if ever, used. The program considers functions that return unused values to be inefficient, and functions that return rarely-used values to be a result of bad programming style.

In addition, **lint** checks for cases where a function does not return a value but the value is used anyway. This is considered a serious error.

## 4.10 Checking Types

The **lint** program enforces the type-checking rules of C more strictly than does the C compiler. The additional checking occurs in four major areas:

- across certain binary operators and implied assignments
- at the structure-selection operators
- between the definitions and uses of functions
- in the use of enumerations

There are a number of operators that have an implied balancing between types of operands. The assignment, conditional, and relational operators have this property. The argument of a **return** statement and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of x's can be intermixed with pointers to x's.

The type-checking rules also require that, in structure references, the left operand of a pointer-arrow symbol (->) must be a pointer to a structure, the left operand of a period (.) must be a structure, and the right operand of these operators must be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return-value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Pointers can also be matched with the associated arrays. Aside from these relaxations in type-checking, all actual arguments must agree in type with their declared counterparts.

The **lint** program checks to ensure that enumeration variables or members are not mixed with other types or other enumerations. It also ensures that the only operations applied to enumerated variables are assignment (=), initialization, equals (==), and not-equals (!=). Enumerations may also be function arguments and return values.

## 4.11 Checking Type Casts

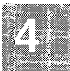
The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment:

```
p = 1 ;
```

where *p* is a character pointer. The **lint** program reports this as suspect. However, in the assignment:

```
p = (char *)1 ;
```

a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. On the other hand, if this code is moved to another machine, it should be looked at carefully. The **-c** option controls the printing of comments about casts. When **-c** is in effect, casts are not checked, and all legal casts are passed without comment, no matter how strange the type mixing seems to be.




## 4.12 Checking for Nonportable Character Use

The **lint** program flags certain comparisons and assignments as illegal or nonportable. For example, the fragment:

```
char c;
  .
  .
  .
if( (c = getchar()) < 0 ) ...
```

works on some machines, but fails on machines where characters always take on positive values. In this case, **lint** issues the message:

```
nonportable character comparison
```

The solution is to declare *c* an integer, since **getchar** is actually returning integer values.

A similar issue arises with bitfields. When assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true on some machines where bitfields are considered as signed quantities. Although a 2-bit field with **int** type cannot hold the value 3, a 2-bit field with **unsigned** type can.

## 4.13 Checking for Assignment of longs to ints

Problems may arise from the assignment of **long** values to **int** values, because of a loss in accuracy in the assignment. This may happen in programs that have been incompletely converted by changing type definitions with **typedef**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to integer values, losing accuracy. Since there are a number of legitimate reasons for assigning longs to integers, you may wish to suppress detection of these assignments by using the **-a** option.

## 4.14 Checking for Strange Constructions

Several perfectly legal but somewhat strange constructions are flagged by **lint**. The generated messages encourage better code quality, clearer style, and may even point out bugs. For example, in the statement:

```
*p++ ;
```

the star (*) does nothing, so **lint** prints:

```
null effect
```

The program fragment:

```
unsigned x ;
if (x < 0) ...
```

is also considered strange since the test will never succeed.

Similarly, the test:

```
if (x > 0)
```

is equivalent to:

```
if (x != 0)
```

which may not be the intended action. In these cases, **lint** prints the message:

```
degenerate unsigned comparison
```

If you use:

```
if( 1 != 0 ) ...
```

then **lint** reports:

```
constant in conditional context
```

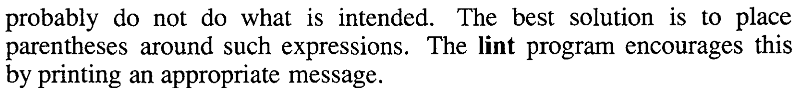
since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs that arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements:

```
if( x&077 == 0 ) ...
```

and:

```
x<< 2 + 40
```



probably do not do what is intended. The best solution is to place parentheses around such expressions. The **lint** program encourages this by printing an appropriate message.

Finally, **lint** checks variables that are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered bad style, is usually unnecessary, and frequently points out a bug.

If you do not want these heuristic checks, you can suppress them by using the **-h** option.

## 4.15 Checking for Use of Older C Syntax

The **lint** program checks for older C constructions. These fall into two classes: assignment operators and initialization.

The older forms of assignment operators (such as, =+, =-, ... ) can cause ambiguous expressions, such as:

```
a =-1 ;
```

which could be taken as either:

```
a =- 1 ;
```

or:

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (such as, +=, -=) have no such ambiguities. To encourage the abandonment of the older forms, **lint** checks for occurrences of these old-fashioned operators.

A similar issue arises with initialization. The older language allowed:

```
int  x  1 ;
```

to initialize *x* to 1. This causes syntactic difficulties. For example:

```
int  x  ( -1 ) ;
```

looks somewhat like the beginning of a function declaration:

```
int  x  ( y ) {  . . .
```

and the compiler must read past *x* to determine what the declaration really is. The problem is even more perplexing when the initializer involves a macro. The current C syntax places an equal sign between the variable and the initializer:

```
int x  =  -1 ;
```

This form is free of any possible syntactic ambiguity.

## 4.16 Checking Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due to alignment restrictions. For example, on some machines it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On other machines, however, double precision values must begin on even-word boundaries; thus, not all such assignments make sense. The **lint** program tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message:

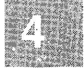
```
possible pointer alignment problem
```

results from this situation.

## 4.17 Checking Expression Evaluation Order

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine-dependent. For example, on machines in which the stack runs backwards, function arguments are probably best evaluated from right to left; on machines with a stack running forward, left to right is probably best. Function calls embedded as arguments of other functions may or may not be treated in the same way as ordinary arguments. Similar issues arise with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

To ensure maximum efficiency of C on a particular machine, the C language leaves the order of evaluation of complicated expressions up to the compiler. Various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and is also used elsewhere in the same expression, the result is undefined.




The **lint** program checks for the important special case where a simple scalar variable is affected. For example, the statement:

```
a[i] = b[i++] ;
```

draws the comment:

```
warning: i evaluation order undefined
```

## 4.18 Embedding Directives

There are occasions when the programmer is smarter than **lint**. There may be valid reasons for illegal type casts, functions with variable numbers of arguments, and other constructions that **lint** finds objectionable. Moreover, as specified in the above sections, the flow of control information produced by **lint** often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Some way of communicating with **lint**, typically to turn off its output, is desirable. Therefore, a number of words are recognized by **lint** when they are embedded in comments in a C source file. These words are called directives. and are invisible to the compiler.

The first directive discussed concerns flow of control information. If a particular place in the program cannot be reached, this can be asserted at the appropriate spot in the program with the directive:

```
/* NOTREACHED */
```

Similarly, if you desire to turn off strict type checking for the next expression, use the directive:

```
/* NOSTRICT */
```

The situation reverts to the previous default after the next expression. The **-v** option can be turned on for one function with the directive:

```
/* ARGSUSED */
```

Comments about a variable number of arguments in calls to a function can be turned off by preceding the function definition with the directive:

```
/* VARARGS */
```

In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. You can define the number of arguments to be checked by placing a digit (giving the number) immediately after the **VARARGS** keyword. For example:

```
/* VARARGS2 */
```

causes only the first two arguments to be checked. Finally, the directive:

```
/* LINTLIBRARY */
```

at the head of a file identifies it as a library declaration file, which is discussed in the next section.

## 4.19 Checking For Library Compatibility

The **lint** program accepts certain library directives, such as:

```
-ly
```

and tests the source files for compatibility with these libraries. This testing is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. These definitions indicate whether a function returns a value, what type a function's return type is, and the number and types of arguments expected by the function. The **VARARGS** and **ARGSUSED** directives can be used to specify features of the library functions.



The **lint** library files are processed almost exactly like ordinary source files. The only difference is that functions defined in a library file but not used in a source file draw no comments. The **lint** program does not simulate a full library-search algorithm, but checks to see if the source files contain redefinitions of library routines.

By default, **lint** checks the programs it is given against a standard library file, which contains descriptions of the programs that are normally loaded when a C program is run. When the **-p** option is in effect, the portable library file is checked. This library contains descriptions of the standard I/O library routines which are expected to be portable across various machines. The **-n** option can be used to suppress all library-checking.

# Chapter 5

# lex: A Lexical Analyzer


5.1 Introduction 5-1

5.2 An Overview of lex Programming 5-2

5.3 How to Format lex Programs 5-3

5.4 Specifying lex Regular Expressions 5-4

5.5 Invoking lex 5-6

5.6 Specifying Character Classes 5-6

5.7 Specifying an Arbitrary Character 5-7

5.8 Specifying Optional Expressions 5-7

5.9 Specifying Repeated Expressions 5-8

5.10 Specifying Alternation and Grouping 5-8

5.11 Specifying Context Sensitivity 5-9

5.12 Specifying Definitions 5-9

5.13 Specifying Expression Repetition 5-10

5.14 Specifying Actions 5-10

5.15 Handling Ambiguous Source Rules 5-14

5.16 Specifying Left Context Sensitivity 5-17

5.17 Specifying Source Definitions 5-19

5.18 Using lex and yacc Together 5-21

5.19 Specifying Character Sets 5-25

5.20 Source Format 5-26

## 5.1 Introduction

A software tool called **lex(CP)** lets you solve a wide class of problems drawn from such tasks as:

- text processing, where you can check the spelling of words for errors
- code enciphering, where you can translate certain patterns of characters into others
- compiler writing, where you can determine what the tokens (smallest meaningful sequences of characters) are in the program to be compiled

The problem common to all of these tasks is recognizing different strings of characters that satisfy certain characteristics. To solve these problems, you can use the compiler's lexical analyzer, also known as **lex**.

It is not essential to use **lex** to handle problems of this kind. You could write programs in a standard language like C to handle them. In fact, what **lex** does is produce such C programs. Therefore, **lex** is called a program generator.) What **lex** offers you is typically a faster, easier way to create programs that perform these tasks. Its weakness is that it often produces C programs that are longer than necessary for the task and that execute more slowly.

So you can understand what **lex** does, the process is briefly described as follows: You begin with the **lex** source (the **lex** specification) that you, the programmer, write to solve the problem. This **lex** source consists of a list of rules specifying sequences of characters (expressions) to be searched for in an input text, and the actions to take when an expression is found. This source is read by the **lex** program generator. The output of the program generator is a C program named **yylex**. This program must be compiled by a host-language C compiler to generate the executable object program that does the lexical analysis. Note that this procedure does not occur automatically. Finally, the lexical analyzer program produced by this process takes as input any source file and produces the desired output, such as altered text or a list of tokens.

The **lex** tool can also be used to collect statistical data on features of the input, such as character count, word length, or number of occurrences of a word. In later sections of this chapter, you will learn how to:

- translate **lex** source

- format **lex** programs
- invoke **lex**
- specify characters and expressions
- write **lex** programs
- use **lex** and **yacc** together

## 5.2 An Overview of lex Programming

Consider a program that deletes from the input all blanks or TABs at the ends of lines. The following lines are all you need:

```
%%
[ \t]+$     ;
```

The program contains a %% delimiter to mark the beginning of the rules and one rule in particular. This rule contains a regular expression that matches one or more instances of the blank or TAB characters (written as \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made up of a blank and a TAB; the + indicates one or more of the previous item; and the dollar sign ($) indicates the end of the line. No action is specified, so **lex** will ignore these characters. Everything else will be copied. To change any remaining string of blanks or TABs to a single blank, add another rule:

```
%%
[ \t]+$     ;
[ \t]+      printf(" ");
```

In the above example, the **lex** program scans for both rules at once, observes at the termination of the string of blanks or TABs whether or not there is a Return, and then executes the desired rule's action. The first rule matches all strings of blanks or TABs at the ends of lines, and the second rule matches all remaining strings of blanks or TABs.

You can also use **lex** alone for simple transformations, or for analysis and gathering of statistics on a lexical level. In addition, you can use **lex** with a parser generator to perform lexical analysis; it is especially easy to interface **lex** and **yacc.** A **lex** program recognizes only regular expressions; **yacc** writes parsers that accept a large class of context-free grammars, but that require a lower-level analyzer to recognize input tokens.

Thus, a combination of **lex** and **yacc** is often appropriate. (The **yacc** program is discussed in "yacc: A Compiler-Compiler.") When you use **lex** as a preprocessor for a later parser generator, it partitions the input stream, and the parser generator assigns structure to the resulting pieces. You can add programs written by other generators or by hand to programs written by **lex**. Users of **yacc** will realize that the name **yylex** is what **yacc** expects its lexical analyzer to be named, so the use of this name by **lex** simplifies interfacing.

The **lex** program generates a finite automaton from the regular expressions specified in the source. It interprets the automaton, rather than compiling it, in order to save space and analyze input faster. The time taken by a **lex** program to recognize and partition an input stream is proportional to the length of the input. The number of **lex** rules or the complexity of the rules is not important in determining speed, unless rules that include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton and, therefore, the size of the program generated by **lex**.

In the program written by **lex**, your programming fragments (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. You can insert either declarations or additional statements in the routine containing the actions, or add subroutines outside this action routine.



The use of **lex** is not limited to sources that can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, **lex** will recognize *ab* and leave the input pointer just before *cd*. This back-up feature can affect program performance.

## 5.3 How to Format lex Programs

A **lex** specification consists of at most three sections: definitions, rules, and user subroutines. The general format of **lex** source is as follows:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The rules section is mandatory. Sections for definitions and user subroutines are optional but, if present, must appear in the indicated order. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum **lex** program is the following (no definitions, no rules) which translates into a program that copies the input to the output unchanged:

```
%%
```

In the **lex** program format shown above, the rules represent your control decisions. They make up a table in which the left column contains regular expressions and the right column contains actions and program fragments to be executed when the expressions are recognized. Thus, the following individual rule might appear:

```
integer    printf("found keyword INT");
```

This looks for the string *integer* in the input stream and prints the following message whenever it appears in the input text:

```
found keyword INT
```

In this example, the C library function **printf()** is used to print the string. The end of the **lex** regular expression is indicated by the first blank or TAB character. If the action is merely a single C expression, you can place it on the right side of the line; if it is compound or takes more than a line, you should enclose it in braces. As a more useful example, suppose you need to change a number of words from British to American spelling. You can start with **lex** rules such as the following:

```
colour          printf("color");
mechanise       printf("mechanize");
petrol          printf("gas");
```

These rules are not quite enough since the word *petroleum* would become *gaseum.* A way of dealing with such problems is described in "Source Format."

## 5.4 Specifying lex Regular Expressions

A regular expression specifies a set of strings to be matched. It contains text characters (that match the corresponding characters in the strings being compared) and operator characters (specifying repetitions, choices, and other features). The letters of the alphabet and the digits are always

text characters. Thus, the following regular expression matches the *integer* string wherever it appears:

*integer*

The following expression looks for the string *a57D*:

```
a57D
```

The operator characters are as follows:

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

If you use any of these characters literally, you need to quote them individually with a backslash (\) or as a group within quotation marks (" "). The quotation mark operator (" ") indicates that whatever is contained between a pair of quotation marks is to be taken as text. Thus, the following matches the string *xyz++* when it appears:

```
xyz"++"
```

Note that you can put only a part of a string in quotation marks. It is harmless but unnecessary to quote an ordinary text character; so the following expression is the same as the one above:

```
"xyz++"
```

Thus, by quoting every nonalphanumeric character being used as a text character, you need not memorize the list of current operator characters.

You can also turn an operator character into a text character by preceding it with a backslash (\) as follows, forming another, less readable, equivalent of the above expressions:

```
xyz\+\+
```

You can also use the quoting feature to get a blank into an expression; normally blanks or TABs end a rule. You must put any blank character not contained within brackets within double quotation marks. Several normal C escapes with the backslash (\) are recognized:

| Escape | Meaning |
|---|---|
| \n | Newline |
| \t | Tab |

| | |
|---|---|
| \b | Backspace |
| \\ | Backslash |

Since Newline is illegal in an expression, you must use a *n*; but it is not required to Escape Tab and Escape Backspace. Every character but Space, Tab, Newline and those listed is always a text character.

## 5.5 Invoking lex

There are two steps in compiling a **lex** source program. First, you must turn the **lex** source into a generated program in the host general-purpose language. Then you must compile and load this program, usually with a library of **lex** subroutines. Note that this program must be compiled as a large model binary. The generated program is in a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library.

You access the library by using the loader flag **-ll**. So an appropriate set of commands is:

```
lex source
cc lex.yy.c -ll
```

**lex** places the resulting program on the usual *a.out* file for later execution. To use **lex** with **yacc**, see "Specifying Source Definitions" in this chapter and "yacc: A Compiler-Compiler". Although the default **lex** I/O routines use the C standard library, the **lex** automaton itself does not do so. If you specify private versions of **input**(), **output**(), and **unput**(), you can avoid the library.

## 5.6 Specifying Character Classes

You can specify classes of characters by using brackets [ ]. The following construction matches a single character, which may be *a*, *b*, or *c*:

```
[abc]
```

Within square brackets, **lex** ignores most operator meanings. Only three characters are special: the backslash (\), the dash (-), and the caret (^). The dash character indicates ranges. For example, the following indicates the character class containing all the lowercase letters, the digits, the angle brackets, and the underline:

```
[a-z0-9<>_]
```

You can specify ranges in either order. If you use the dash between any two characters that are not both uppercase letters, both lowercase letters, or both digits, you will get a warning message that tells you this range of characters is hardware-dependent. If you desire to include the dash in a character class, it should be first or last; thus, the following matches all the digits and the plus and minus signs:

```
[-+0-9]
```

In character classes, the caret (^) operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus, the following matches all characters except *a*, *b*, or *c*, including all special or control characters:

```
[^abc]
```

Alternatively, the following is any character which is not a letter:

```
[^a-zA-Z]
```

The backslash (\) provides an escape within character class brackets, so that you can type characters literally by preceding them with this character.

## 5.7 Specifying an Arbitrary Character

To match almost any character, the dot (.) designates the class of all characters except a Newline. Escaping into octal is possible although nonportable. For example, the following matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

```
[\40-\176]
```

## 5.8 Specifying Optional Expressions

The question mark (?) operator indicates an optional element of an expression. Thus, the following matches either *ac* or *abc*:

```
ab?c
```

Note that the meaning of the question mark here differs from its meaning in the shell.

## 5.9 Specifying Repeated Expressions

Repetitions of classes are indicated by the asterisk (*) and plus (+) operators. For example, the following matches any number of consecutive *a* characters, including zero:

```
a*
```

*a*+ matches one or more instances of *a*. For example, the following matches all strings of lowercase letters:

```
[a-z]+
```

The following matches all alphanumeric strings with a leading alphabetic character:

```
[A-Za-z][A-Za-z0-9]*
```

This is a typical expression for recognizing identifiers in computer languages.

## 5.10 Specifying Alternation and Grouping

The vertical bar ( | ) operator indicates alternation. For example, the following matches either *ab* or *cd*:

```
(ab|cd)
```

Note that parentheses are used for grouping, although they are not necessary at the outside level. You could type:

```
ab|cd
```

However, you should use parentheses for more complex expressions, such as the following example, which matches such strings as *abefef*, *efefef*, *cdef*, and *cddd*, but not *abc*, *abcd*, or *abcdef*:

```
(ab|cd+)?(ef)*
```

## 5.11 Specifying Context Sensitivity

The **lex** program recognizes a small amount of surrounding context. The two simplest operators for this are the caret (^) and the dollar sign ($). If the first character of an expression is a caret (^), the expression is matched at the beginning of a line (after a Newline character or at the beginning of the input stream). This does not conflict with the complementation of character classes, since complementation only applies within brackets. If the last character is a dollar sign, the expression is matched at the end of a line (when immediately followed by Newline ). The latter operator is a special case of the slash (/) operator, which indicates trailing context. The following expression matches the string *ab*, but only if followed by *cd*:

```
ab/cd
```

Thus:

```
ab$
```

is the same as:

```
ab/\n
```



The **lex** program handles left context by specifying start conditions as explained in "Specifying Actions." If a rule is only to be executed when the **lex** automaton interpreter is in start condition *x*, you should enclose the rule in angle brackets:

```
<x>
```

Suppose you consider being at the beginning of a line to be start condition ONE, then the caret (^) operator would be equivalent to:

```
<ONE>
```

## 5.12 Specifying Definitions

Braces ({ }) specify definition expansion (if they enclose a name) or repetitions (if they enclose numbers). For example, the following looks for a predefined string named *digit* and inserts it at that point in the expression:

```
{digit}
```

## 5.13 Specifying Expression Repetition

The definitions are given in the first part of the **lex** input, before the rules. In contrast, the following looks for 1 to 5 occurrences (repetitions) of the character *a*:

```
a{1,5}
```

Finally, an initial percent sign ( % ) is special, since it is the separator for **lex** source segments.

## 5.14 Specifying Actions

When a pattern of text in the input matches an expression, **lex** executes the corresponding action. This section describes some features of **lex** that aid in writing actions. Note that there is a default action that consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus, if you want to absorb the entire input without producing any output, you must provide rules to match everything. When you use **lex** with **yacc**, this is the normal situation. You may consider that actions are what is done instead of copying the input to the output; in general, you can omit a rule which merely copies.

One of the simplest things that you can do is ignore the input by specifying a C null statement. A frequent rule is the following, which ignores the three spacing characters (Spacebar, Tab, and Newline):

```
[ \t\n]     ;
```

Another easy way to avoid writing actions is to use the repeat action character pipe (|), which indicates that the action for this rule is the action for the next rule. The previous example could also have been written as follows with the same result:

```
" "         |
"\t"        |
"\n"        ;
```

The quotation marks around \n and \t are not required.

In more complex actions, you may want to know the actual text that matched an expression like:

```
[a-z]+
```

The **lex** program leaves this text in an external character array named **yytext**. Thus, to print the name found, a rule like the following prints the string in **yytext**:

```
[a-z]+     printf("%s", yytext);
```

The C function **printf** accepts a format argument and data to be printed when a pattern of text in the input matches an expression. In this case, the format is *print string* where the percent sign (%) indicates data conversion, the *s* indicates string type, and the data are the characters in **yytext**. This places the matched string on the output. This action is so common that you can write it as ECHO. For example, the following is the same as the preceding example:

```
[a-z]+     ECHO;
```

Since the default action is to print the characters found, you might ask why give a rule which merely specifies the default action. Such rules are often required to avoid matching some other rule that is not desired. For example, if there is a rule that matches *read*, it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, you need a rule of the form:

```
[a-z]+
```

For more information, see ''Handling Ambiguous Source Rules.''

Sometimes it is more convenient to know the end of that which has been found; hence **lex** also provides a count of the number of characters matched in the variable, *yyleng*.

To count both the number of words and the number of characters in words in the input, you might write the following which accumulates in the variables *chars* the number of characters in the *words* recognized:

```
[a-zA-Z]+   {words++; chars += yyleng;}
```

You can access the last matched character in the string:

```
yytext[yyleng-1]
```

5

Occasionally, a **lex** action may decide that a rule has not recognized the correct span of characters. Two routines aid this situation. First, you can call **yymore**() to indicate that the next input expression recognized is to be tacked onto the end of this input. Normally, the next input string overwrites the current entry in **yytext**. Second, you can call **yyless**(*n*) to indicate that not all of the characters matched by the currently successful expression are needed right now. The argument *n* indicates the number of characters in **yytext** to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the slash (/) operator, but in a different form.

For example, consider a language that defines a string as a set of characters between quotation marks (" "); but, to include a quotation mark in a string, it must be preceded by a backslash (\). The regular expression that matches this is somewhat confusing, so it might be preferable to write:

```
\"[^"]*     {
        if (yytext[yyleng-1] == '\\')
                yymore();
        else
                ... normal user processing
        }
```

which, when faced with a string such as:

```
"abc\"def"
```

will first match the five characters:

```
"abc\
```

and then the call to **yymore**() will tack the next part of the string on the end:

```
"def
```

Note that you should pick up the final quotation mark that terminates the string in code-labeled, normal processing.

You might use the **yyless**() function to reprocess text in various circumstances. Consider the problem in the older C syntax of distinguishing the ambiguity of =-*a*. Suppose you want to treat this as =- *a* and to print a message. A possible rule that prints a message, returns the letter following the operator to the input stream, and treats the operator as =– might be:

```
=-[a-zA-Z] {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-1);
        ... action for =- ...
        }
```

Alternatively, you might want to treat this as = –*a*. To do this, just return the minus sign as well as the letter to the input. The following performs the interpretation:

```
=-[a-zA-Z] {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-2);
        ... action for = ...
        }
```

Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case, and:

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of =–*3*, however, makes the following an even better rule:

```
=-/[^ \t\n]
```

In addition to these routines, **lex** also permits access to the I/O routines it uses. They include:

| **Routine** | **Description** |
| --- | --- |
| **input**() | Returns the next input character. |
| **output**(*c*) | Writes the character *c* on the output. |
| **unput**(*c*) | Pushes the character *c* back onto the input stream to be read later by **input**(). |

By default, these routines are provided as macro definitions, but you can override them and supply private versions. These routines define the relationship between external files and internal characters, and must be retained or modified consistently. You can redefine them to cause input or output to be transmitted to or from strange places, including other programs or internal memory, but the character set that you use must be consistent in all routines. A value of zero returned by **input**() must mean end-of-file, and the relationship between **unput**() and **input**() must be retained or the lookahead will not work. The **lex** program does not look ahead if it does not have to, but every rule containing a slash (/) or ending in one of the following characters implies lookahead:

+ * ? $

Lookahead is also necessary to match an expression that is a prefix of another expression. See the following discussion of the character set used by **lex**. The standard **lex** library imposes a 100-character limit on backup.

Another **lex** library routine that you may need to redefine is **yywrap**(), which is called whenever **lex** reaches an end-of-file. If **yywrap** returns a 1, **lex** continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, you should provide a **yywrap** that arranges for new input and returns 0. This instructs **lex** to continue processing. The default **yywrap** always returns 1.

This routine is also a convenient place to print tables or summaries at the end of a program. Note that it is not possible to write a normal rule that recognizes end-of-file; the only access to this condition is through **yywrap**(). In fact, unless a private version of **input**() is supplied, a file containing nulls cannot be handled, since a value of 0 returned by **input** is considered an end-of-file.

## 5.15 Handling Ambiguous Source Rules

The **lex** program can handle ambiguous specifications. When more than one expression matches the current input, **lex** chooses as follows:

- The longest match is preferred.

- Among rules that match the same number of characters, the first given rule is preferred.

For example, suppose the following rules are given:

```
integer keyword action ...;
[a-z]+     identifier action ...;
```

If the input is *integers*, it is taken as an identifier, because the following matches eight characters:

```
[a-z]+
```

while *integer* matches only seven.

If the input is *integer*, both rules match seven characters, and **lex** selects the keyword rule because it was given first. Anything shorter (for example, *int*) does not match the expression *integer*, so **lex** uses the identifier interpretation.

The principle of preferring the longest match makes certain constructions dangerous, such as the following:

```
.*
```

For example the following might seem a good way of recognizing a string in single quotes:

```
'.*'
```



However, it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input:

```
'first' quoted string here, 'second' here
```

this expression matches:

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the following form which, on the previous input, stops after 'first':

```
'[^'\n]*'
```

The fact that the dot (.) operator does not match a Newline lessens the consequences of errors like this. Therefore, no more than one line is ever matched by such expressions. Don't try to defeat this with expressions like the following or their equivalents, because the **lex** -generated program will try to read the entire input file, causing internal buffer overflows:

```
[.\n]+
```

Note that **lex** is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for only once. For example, suppose you want to count the

occurrences of both *she* and *he* in an input text. Some **lex** rules to do this might be the following where the last two rules ignore everything besides *he* and *she*:

```
she    s++;
he     h++;
\n     |
.      ;
```

Remember that dot (.) does not include the Newline. Since *she* includes *he*, **lex** will normally not recognize the instances of *he* included in *she*, since once it has passed a *she* those characters are gone.

Sometimes you may want to override this choice. The *Reject* action means, "go do the next alternative." It executes whatever rule was second choice after the current rule. It then adjusts the position of the input pointer accordingly. Suppose you want to count the included instances of *he*:

```
she    {s++; REJECT}
he     {h++; REJECT}
\n     |
.      ;
```

These rules are one way of changing the previous example to do just that. After each expression is counted, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that *she* includes *he*, but not vice versa, and omit the *Reject* action on *he*. In other cases, it would not be possible to tell which input characters were in both classes.

Consider the following two rules:

```
a[bc]+      { ... ; REJECT}
a[cd]+      { ... ; REJECT}
```

If the input is *ab*, only the first rule matches; on *ad*, only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, *Reject* is useful whenever the purpose of **lex** is, not to partition the input stream, but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired. Normally the digrams overlap, that is; the word *the* is considered to contain both *th* and *he*. Assuming

you have a two-dimensional array named *digram* that you want to increment, the appropriate source is:

```
%%
[a-z][a-z] {digram[yytext[0]][yytext[1]]++; REJECT}
.       ;
\n      ;
```

where the *Reject* is necessary to pick up a letter pair beginning at every character, rather than at every other character.

Remember that *Reject* does not rescan the input. Instead, it remembers the results of the previous scan. This means that if a rule with trailing context is found and *Reject* executed, you must not have used **unput** to change the characters coming from the input stream. This is the only restriction on manipulating the unprocessed input.

### 5.16 Specifying Left Context Sensitivity

You may need several sets of lexical rules to apply at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The caret (^) operator, for example, is a prior context operator, recognizing immediately preceding left context, just as the dollar sign ($) recognizes immediately following right context. Adjacent left context could be extended to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, because the relevant left context often appears first and at the beginning of a line.

This section describes three means of dealing with different environments:

- using flags, when only a few rules change from one environment to another,
- using start conditions with rules, and
- using several lexical analyzers running together.

In each case, there are rules that recognize the need to change the environment in which the following input text is analyzed and set to some parameters to reflect the change. This may be a flag tested explicitly by your action code; such a flag is the simplest way of dealing with the problem, since **lex** is not involved at all. It may be more convenient, however, for **lex** to remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when **lex** is in that start condition. The current start condition can be changed

at any time. Finally, if the sets of rules for the different environments are very dissimilar, you can, for the sake of clarity, write several distinct lexical analyzers and switch from one to another as you need them.

Consider the following problem: Copy the input to the output changing the word *magic* to *first* on every line that begins with the letter *a*, changing *magic* to *second* on every line that begins with the letter *b*, and changing *magic* to *third* on every line that begins with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag =  0 ; ECHO;}
magic {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

To handle the same problem with start conditions, you must introduce each start condition to **lex** in the definitions section with a line reading

`%Start` *name1 name2*

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. You can reference the conditions at the head of a rule with angle brackets. For example, the following is a rule that **lex** recognizes only when it is in the start condition *name1*:

*<name1>expression*

To enter a start condition, execute the following action statement which changes the start condition to *name1*:

`BEGIN` *name1*;

To return to the initial state, the following resets the initial condition of the **lex** automaton interpreter:

```
BEGIN 0;
```

A rule may be active in several start conditions. For example, the following is a legal prefix:

*<name1,name2,name3>*

Any rule not beginning with the <> prefix operator is always active.

The same example can be written as follows, where the logic is exactly the same as in the previous method of handling the problem, but **lex** does the work rather than your code:

```
%START AA BB CC
%%
^a     {ECHO; BEGIN AA;}
^b     {ECHO; BEGIN BB;}
^c     {ECHO; BEGIN CC;}
\n     {ECHO; BEGIN 0;}
<AA>magic  printf("first");
<BB>magic  printf("second");
<CC>magic  printf("third");
```

## 5.17 Specifying Source Definitions

5

Remember the format of the **lex** source:

{*definitions*}
%%
{*rules*}
%%
{*user routines*}

So far only the rules have been described. You will need additional options, though, to define variables for use in your program and for use by **lex**. These can go either in the definitions section or in the rules section.

Remember that **lex** is turning the rules into a program. Any source that **lex** does not intercept is copied into the generated program. There are three classes of such sources:

1. Any line that is not part of a **lex** rule or action that begins with a blank or TAB is copied into the **lex** -generated program. Source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the **lex** function which contains the actions. This material must look like program fragments and should precede the first **lex** rule.

As a side effect of this, lines that begin with a blank or TAB and contain a comment are passed through to the generated program. You can use this to include comments in either the **lex** source or the generated code. The comments should follow C language conventions.

2. Anything included between lines containing only %{ and %} is copied out as stated earlier. The delimiters are discarded. This format permits entering text like preprocessor statements which begin in column 1 or copying lines that do not look like programs.

3. Anything after the third %% delimiter, regardless of format, is copied out after the **lex** output.

Definitions intended for **lex** are given before the first %% delimiter. Any line in this section not contained between %{ and %} and beginning in column 1 is assumed to define **lex** substitution strings. The format of such lines is causing the translation string to be associated with the following name:

*name translation*

The name and translation must be separated by at least one blank or Tab , and the name must begin with a letter. The translation can then be called out by the {*name*} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D               [0-9]
E               [DEde][-+]?{D}+
%%
{D}+            printf("integer");
{D}+"."{D}*({E})?     |
{D}*"."{D}+({E})?     |
{D}+{E}               printf("real");
```

Note the first two rules for real numbers; each requires a decimal point and contains an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To handle correctly the problem posed by a FORTRAN expression such as *35.EQ.I*, which does not contain a real number, you can use a context-sensitive rule such as the following in addition to the normal rule for integers:

```
[0-9]+/"."EQ    printf("integer");
```

The definitions section can also contain other commands, including a character set table, a list of start conditions, or adjustments to the default

size of arrays within **lex** to accommodate larger source programs. These possibilities are discussed in ''Specifying Character Sets.''

## 5.18 Using lex and yacc Together

If you want to use **lex** with **yacc**, note that what **lex** writes is a program, **yylex**(), which is the name required by **yacc** for its analyzer. Normally, the default main program on the **lex** library calls this routine, but if **yacc** is loaded and its main program is used, **yacc** will call **yylex**(). In this case, each **lex** rule should end with the following where the appropriate token value is returned:

```
return(token);
```

An easy way to get access to **yacc**'s names for tokens is to compile the **lex** output file as part of the **yacc** output file by placing the following line in the last section of **yacc** input:

```
# include "lex.yy.c"
```

Supposing the grammar to be named *good* and the lexical rules to be named *better*, the XENIX command sequence can be:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

You should load the **yacc** library (**-ly**) before the **lex** library to obtain a main program which invokes the **yacc** parser. The generation of **lex** and **yacc** programs can be done in either order.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is an example **lex** source program:

```
%%
      int k;
[0-9]+      {
      k = atoi(yytext);
      if (k%7 == 0)
           printf("%d", k+3);
      else
           printf("%d",k);
      }
```

The rule [0-9]+ recognizes strings of digits; **atoi**() converts the digits to binary and stores the result in *k*. The remainder operator (%) is used to check whether *k* is divisible by 7; if so, it is incremented by 3 as it is

written out. You may object that this program will alter such input items as 49.63 or X7. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add more rules after the active one, as here:

```
%%
             int k;
-?[0-9]+      {
             k = atoi(yytext);
             printf("%d", k%7 == 0 ? k+3 : k);
             }
-?[0-9.]+          ECHO;
[A-Za-z][A-Za-z0-9]+ ECHO;
```

Numerical strings containing a decimal point or preceded by a letter will be picked up by one of the last two rules and not changed. The **if-else** has been replaced by a C-conditional expression to save space; the form, *a?b:c*, means: if *a* then *b* else *c*.

For an example of statistics gathering, here is a program which makes histograms of word lengths, where a word is defined as a string of letters:

```
             int lengs[100];
%%
[a-z]+        lengs[yyleng]++;
.      |
\n     ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
      if (lengs[i] > 0)
            printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram while producing no output. At the end of the input, it prints the table. The final statement, **return**(1), indicates that **lex** is to perform wrapup. If **yywrap**() returns zero (false), it implies that further input is available and the program is to continue reading and processing. Providing a **yywrap**() that never returns true causes an infinite loop.

As a larger example, the following are some parts of a program for converting double-precision FORTRAN to single-precision FORTRAN. Because FORTRAN does not distinguish between uppercase and lowercase letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
.       .
.       .
.       .
z       [zZ]
```

An additional class recognizes whitespace:

```
W       [ \t]*
```

The first rule changes *double precision* to *real*, or *DOUBLE PRECISION* to *REAL*:

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n}   {
     printf(yytext[0]=='d'? "real" : "REAL");
     }
```

5

Care is taken throughout this program to preserve the case of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indicators to avoid confusing them with constants:

```
^"     "[^ 0]    ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as beginning of line, then five blanks, then anything but blank or zero. Note the two different meanings of the caret (^) here. The following rules change double-precision constants to ordinary floating constants:

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+          |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+    |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+    {
     /* convert constants */
     for(p=yytext; *p != 0; p++)
          {
          if (*p == 'd' || *p == 'D')
               *p+= 'e'- 'd';
          ECHO;
          }
```

After the floating point constant is recognized, it is scanned by the **for** loop to find the letter *d* or *D*. The program then adds "'e'-'d'" which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follows a series of names which must be respelled to remove their initial *d*'s. If you use the **yytext** array, the same action suffices for all the names. Only a sample of a rather long list is given here:

```
{d}{s}{i}{n}        |
{d}{c}{o}{s}        |
{d}{s}{q}{r}{t}     |
{d}{a}{t}{a}{n}     |
...
{d}{f}{l}{o}{a}{t}      printf("%s",yytext+1);
```

Another list of names must have initial *d*'s changed to initial *a*'s

```
{d}{l}{o}{g}        |
{d}{l}{o}{g}10      |
{d}{m}{i}{n}1       |
{d}{m}{a}{x}1       {
        yytext[0] += 'a' - 'd';
        ECHO;
        }
```

One routine must have an initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h} {
        yytext[0] += 'r'  - 'd';
        ECHO;
}
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters. For more information on **yacc**, see "yacc: A Compiler-Compiler."

```
[A-Za-z][A-Za-z0-9]*  |
[0-9]+                |
\n                    |
.       ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

## 5.19 Specifying Character Sets

The programs generated by **lex** handle character I/O only through the routines **input**, **output**, and **unput**. Thus, the character representation provided in these routines is accepted by **lex** and employed to return values in **yytext**. For internal use, a character is represented as a small integer. If you use the standard library, a character has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented in the same form as the following character constant:

```
'a'
```

If this interpretation is changed by providing I/O routines that translate the characters, **lex** must be told about it by means of a translation table. You must have this table in the definitions section, and it must be bracketed by lines containing only *%T*. The table contains lines of the following form which indicate the value associated with each character:

{*integer*} {*character string*}

For example:

```
%T
 1    Aa
 2    Bb
...
26    Zz
27    \n
28    +
29    -
30    0
31    1
...
39    9
%T
```

This table maps the lowercase and uppercase letters together into the integers 1 through 26, Newline into 27, plus (+) and minus (-) into 28 and 29, and the digits into 30 through 39. Note the escape for Newline. If you supply a table, you must include every character that is to appear either in the rules or in any valid input. No character may be assigned the number 0, and no character may be assigned a larger number than the size of the hardware character set.

## 5.20 Source Format

Remember the general form of a **lex** source file:

> {*definitions*}
> %%
> {*rules*}
> %%
> {*user subroutines*}

The definitions section contains a combination of the following:

- Definitions, in the form "name space translation"
- Included code, in the form "space code"
- Included code, in the form:

  > %{
  > *code*
  > %}

- Start conditions, given in the form:

  > %S *name1 name2*

- Changes to internal array sizes, in the form:

  > %*x* *nnn*

  where *nnn* is a decimal integer representing an array size, and *x* selects the parameter as follows:

  | Letter | Parameter |
  |---|---|
  | p | positions |
  | n | states |
  | e | tree nodes |
  | a | transitions |
  | k | packed character classes |
  | o | output array size |

- Character set tables, in the form:

  > %T
  > *number space character-string*
  > %T

Lines in the rules section have the following form where the action may be continued on succeeding lines by using braces to delimit it:

*expression action*

Regular expressions in **lex** use the following operators:

| **Operator** | **Description** |
|---|---|
| x | The character *x* |
| ``x'' | An ``x'', even if *x* is an operator |
| \x | An ``x'', even if *x* is an operator |
| [xy] | The character *x* or *y* |
| [x-z] | The character *x*, *y*, or *z* |
| [^x] | Any character but *x* |
| . | Any character but Newline |
| ^x | An *x* at the beginning of a line |
| <y>x | An *x* when **lex** is in start condition y |
| x$ | An *x* at the end of a line |
| x? | An optional *x* |
| x* | 0,1,2, ... instances of *x* |
| x+ | 1,2,3, ... instances of *x* |
| x\|y | An *x* or a *y* |
| (x) | An *x* |
| x/y | An *x* but only if followed by *y* |
| {*xx*} | The translation of *xx* from the definitions section |
| x{*m*,*n*} | *m* through *n* occurrences of x |

# Chapter 6

# yacc: A Compiler-Compiler

6.1 Introduction 6-1

6.2 Basic yacc Specifications 6-4
- 6.2.1 Rules 6-5
- 6.2.2 Actions 6-7
- 6.2.3 How to Prepare the Lexical Analyzer 6-9

6.3 How the Parser Works 6-11

6.4 Ambiguity and Conflicts 6-15

6.5 How to Handle Operator Precedences 6-20

6.6 Error Handling and Recovery 6-23

6.7 The yacc Environment 6-25

6.8 Preparing Specifications 6-26
- 6.8.1 Input Style 6-27
- 6.8.2 Left Recursion 6-27
- 6.8.3 Lexical Tie-ins 6-28
- 6.8.4 Handling Reserved Words 6-29

6.9 Advanced Topics 6-30
- 6.9.1 Simulating Error and Accept in Actions 6-30
- 6.9.2 Accessing Values in Enclosing Rules 6-30
- 6.9.3 Supporting Arbitrary Value Types 6-31
- 6.9.4 yacc Input Syntax 6-32

6.10 Examples 6-35
- 6.10.1 A Simple Example 6-35
- 6.10.2 An Advanced Example 6-37

6.11 Old Features Supported but Not Encouraged 6-43

## 6.1 Introduction

Computer program input generally has some structure; every computer program that accepts input can be thought of as defining an input language which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often lax about checking their inputs for validity.

The **yacc**(CP) program provides a general tool for describing the input to a computer program. The name **yacc** stands for "yet another compiler-compiler." The **yacc** user specifies the structures of input, together with code to be invoked as each input structure is recognized. **yacc** turns this structure specification into a subroutine that handles the input process and controls the flow of the user's application.

The input subroutine produced by **yacc** calls a user-supplied routine to return the next basic input item. Thus, the user can specify input in terms of individual input characters or in terms of higher-level constructs such as names and numbers. The user-supplied routine can also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification. The class of specifications accepted is a general one: LALR (lookahead-left-read) grammars with rules for clarification, referred to as "disambiguating rules."

In addition to compilers such as those for C, APL, Pascal, RATFOR, less conventional languages also use **yacc**, including a phototypesetter language, several desk-calculator languages, a document retrieval system, and a FORTRAN debugging system.

6

Since **yacc** imposes structure on the input to a computer program, the **yacc** user can prepare a specification of the input process, including rules that describe the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Then **yacc** goes through the following steps:

1. The **yacc** program generates a function (parser) to control the input process.

2. The parser calls the user-supplied lexical analyzer to pick up the basic terms (terminal symbols) from the input stream.

3. Terminal symbols are organized according to input structure rules (grammar).

4. Once one of the grammar rules is recognized, the user code supplied for this rule is invoked; these rules, which are actions, have

the ability to return values and make use of the values of other actions.

The **yacc** program is written in a portable dialect of C, and the actions and output subroutines are in C as well. Also, many of the syntactic conventions of **yacc** follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be:

*date* : *month_name day* ',' *year* ;

These represent structures of the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma (,) is enclosed in single righthand quotation marks, implying that it is to appear literally in the input. The colon and semicolon serve as punctuation in the rule and have no significance in controlling the input. Thus, with proper definitions, the following input might be matched by the given rule:

```
July  4, 1776
```

An important part of the input process is performed by the lexical analyzer. The (**lex**) user routine reads the input stream, recognizes the lower-level structures, and communicates these tokens to the parser. A structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols are usually referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the following rules might be used in the previous example:

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;
        .
        .
        .
month_name : 'D' 'e' 'c' ;
```

The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules can waste time and space and can complicate the specification beyond **yacc**'s ability. Usually, the lexical analyzer would recognize the month names and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters, such as the comma, are considered tokens and must also be passed through the lexical analyzer.

Specification files are very flexible. It is relatively easy to add the following rule to the preceding example:

*date* : *month* '/' *day* '/' *year* ;

allowing

```
7/4/1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be added to a working system with minimal effort and with little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as possible with a left-to-right scan; this substantially reduces the chance of reading and computing with bad input data, and the bad data can usually be found quickly. Error handling, provided as part of the input specifications, permits the reentry of bad data or continues the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, representing a design error; or the specifications may require more powerful recognition than **yacc** has available. (This can often be corrected by making the lexical analyzer more powerful or by rewriting some of the grammar rules.) While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems. The constructions which are difficult for **yacc** to manage are also frequently difficult for people to manage.



The following sections describe:

- preparing grammar rules,
- preparing the user-supplied actions associated with the grammar rules,
- preparing lexical analyzers,
- using the parser,

- handling operator precedences in arithmetic expressions,
- detecting errors and recovering from them,
- why **yacc** may be unable to produce a parser from a specification and what to do about it,
- the operating environment and special features of the parsers **yacc** produces, and
- some suggestions that can improve the style and efficiency of the specifications.

## 6.2 Basic yacc Specifications

Names refer to either tokens or nonterminal symbols. The **yacc** program requires token names to be declared as such. In addition, you may want to include the lexical analyzer and other programs as part of the specification file. Thus, every specification file consists of three sections:

- declarations
- rules (grammar)
- programs

Double percent (%%) marks separate the sections. (The percent sign (%) is generally used in **yacc** specifications as an escape character.)

For example, a full-specification file looks like this:

```
declarations
%%
rules
%%
programs
```

The declaration and program sections can be empty. Thus, the smallest legal **yacc** specification is:

```
%%
rules
```

Spaces, TAB, and Newline are ignored, except that they may not appear in names or in multicharacter reserved symbols. Comments can appear wherever a name is legal and are enclosed in /* ... */, as in C.

### 6.2.1 Rules

The rules section is made up of one or more grammar rules. A grammar rule has the form:

*A* : *BODY* ;

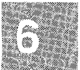
where:

- *A* represents a nonterminal name,
- *BODY* represents a sequence of zero or more names and literals, and
- colon and the semicolon represent **yacc** punctuation.

Names can be of arbitrary length and can be made up of letters, dot (.), the underscore (_), and noninitial digits. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule can represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotation marks ( ' ). As in C, the backslash (\) is an escape character within literals and all the C escapes are recognized:

```
'\n'  NEWLINE
'\r'  RETURN
'\"   Single quotation mark
'\\'  Backslash
'\t'  TAB
'\b'  BACKSPACE
'\f'  FORMFEED
'\xxx'      "xxx" in octal
```



For a number of technical reasons, the ASCII NULL character should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, then you can use the pipe symbol ( | ) to avoid rewriting the left-hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus, the following grammar rules:

```
A : B  C  D ;
A : E  F  ;
A : G  ;
```

can be given to **yacc** as:

```
A : B  C  D
  | E  F
  | G
  ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, but doing so makes the input much more readable and easier to change.

If a nonterminal symbol matches the empty string, you can indicate this in the following manner:

```
empty : ;
```

Also, since you must declare names that represent tokens, you can do this by typing the following in the declarations section:

%token *name1 name2*

Every nonterminal symbol must appear on the left-hand side of at least one rule. (For more information, see "How the Parser Works," "How to Handle Operator Precedences," and "Error Handling and Recovery.")

The start symbol has particular importance. The parser is designed to recognize the start symbol; this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is the first character on the left-hand side of the first grammar rule in the rules section. You can declare the start symbol explicitly in the declarations section using the **%start** keyword:

**%start** *symbol*

A special token (called the endmarker) signals the end of the input to the parser. If the tokens up to, but not including, the endmarker form a structure that matches the start symbol, the parser function returns to its caller. After the endmarker is seen, the parser accepts the input. If the endmarker is seen in any other context, it is an error.

The user-supplied lexical analyzer is responsible for returning the endmarker when appropriate; for more information, see "How the Parser Works." Usually the endmarker represents a particular I/O status, such as end-of-file or end-of-record.

### 6.2.2 Actions

With each grammar rule, you can associate actions to be performed each time that rule is recognized in the input process. These actions can return values and take values returned by previous actions. You can also make the lexical analyzer return values for tokens.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. You specify an action with one or more statements enclosed in braces ({ }). For example, the following are grammar rules with actions:

```
A : '(' B ')'
               {       hello( 1, "abc" );   }
```

and:

```
XXX : YYY ZZZ
               { printf("a message\n");
                 flag = 25;}
```

For easy communication between the actions and the parser, you must alter the action statements slightly. In this context, the dollar sign ($) is used as a signal to **yacc**.

To return a value, the action normally sets the *$$* pseudo-variable to some value. For example, the following action does nothing but return the value 1:

```
{ $$ = 1; }
```

6

To obtain the values returned by previous actions and the lexical analyzer, the action may use the $1, $2, ... pseudo-variables, which refer to the values returned by the components of the right-hand side of a rule. Thus, if the rule is as follows, $2 has the value returned by C, and $3 has the value returned by D:

```
A : B C D ;
```

As a more concrete example, consider the following rule:

*expr* : '(' *expr* ')' ;

The value returned by this rule is usually the value of the *expr* in parentheses. You can indicate this by typing:

```
expr : '(' expr ')' { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the following form frequently need not have an explicit action:

```
A : B ;
```

In the preceding examples, all the actions came at the end of their rules. Sometimes, you may need to get control before a rule is fully parsed. The **yacc** program lets you write an action in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual method by the actions to the right of it. In turn, the rule can access the values returned by the symbols to its left. Thus, in the following rule, the effect is to set *x* to 1, and *y* to the value returned by C:

```
A : B
    {   $$ = 1;   }
    C
    {    x = $2;     y = $3;    }
    ;
```

Actions that do not terminate a rule are handled by **yacc**. In turn, **yacc** manufactures a new nonterminal symbol name, and a new rule matching this name, to the empty string. The interior action is triggered by recognizing this added rule. The **yacc** program actually treats the previous example as if it had been written as:

```
$ACT  : /* empty */
                        {   $$ = 1;   }
      ;

A     : B   $ACT   C
                        {    x = $2;     y = $3;    }
      ;
```

In many applications, the actions do not produce output directly; instead, you can construct a data structure (such as a parse tree) in memory and apply transformations to it before the output is generated. Parse trees are particularly easy to construct if you have the routines to build and maintain the desired tree structure.

For example, suppose there is a C function *node*, written so that the following call creates a node with label L and descendants n1 and n2 and returns the index of the newly created node:

```
node( L, n1, n2 )
```

Since the **yacc** parser uses only names beginning in *yy*, you should avoid such names. In these examples, all the values are integers.

You can build the parse tree by supplying actions such as the following in the specification:

```
expr  :  expr '+' expr
                        {  $$ = node( '+', $1, $3 );  }
```

You can define other variables to be used by the actions. For instance, declarations (tokens) and definitions can appear in the declarations section, enclosed in the *%{* and *%}* marks. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example, you could type the following in the declarations section, making *variable* accessible to all of the actions:

```
%{ int variable = 0; %}
```

### 6.2.3 How to Prepare the Lexical Analyzer

You must use a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called **yylex**. The function returns an integer, called the token number, which represents the kind of token read. If there is a value associated with that token, you should assign it to the external variable *yylval.*

6

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. Either you or **yacc** should choose these token numbers. In either case, using the *# define* mechanism of C lets the lexical analyzer return these numbers symbolically. For example, suppose that the token name *DIGIT* is defined in the

declarations section of the **yacc** specification file. The relevant portion of the lexical analyzer might look like this:

```
yylex(){
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch( c ) {
                ...
        case '0':
        case '1':
          ...
        case '9':
                yylval = c-'0';
                return( DIGIT );
                ...
                }
        ...
```

The intent is to return a token number *DIGIT* and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is in the programs section of the specification file, the identifier, *DIGIT*, is defined as the token number associated with the token *DIGIT*.

This method leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using token names in the grammars reserved or significant in C or the parser. For example, if you use the token names **if** or **while**, you will cause severe difficulties when the lexical analyzer is compiled. The token name **error** is reserved for error handling so you should not use it carelessly.

As mentioned previously, either you or **yacc** can choose the token numbers. By default, **yacc** chooses the token numbers. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

When you assign a token number to a token (including literals), you can follow the first appearance of the token name or literal in the declarations section with a positive integer. This integer specifies the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the token number of the endmarker must be equal to or less than 0. You cannot redefine this token number. Therefore, you should prepare every lexical analyzer to return a 0 or a negative value as a token number upon reaching the end of its input.

A very useful tool for constructing lexical analyzers is **lex**, discussed in "lex: A Lexical Analyzer." These lexical analyzers are designed to work in close harmony with **yacc** parsers. The specifications for lexical analyzers use regular expressions instead of grammar rules. You can use **lex** to produce complicated lexical analyzers, but some languages (such as Fortran ) do not fit any theoretical framework, so you must design their lexical analyzers by hand.

## 6.3 How the Parser Works

The **yacc** program turns the specification file into a C program that parses the input according to the specification given. The parser produced by **yacc** consists of a finite state machine with a stack. The parser is capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels. Initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it: *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

6

1. Based on its current state, the parser determines whether it needs a lookahead token to decide what action should be taken. If it needs one and does not have one, it calls **yylex** to obtain the next token.

2. Using the current state and the lookahead token if needed, the parser determines its next action and executes it. This may result in states being pushed onto the stack or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a *shift* action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is then cleared.

The *reduce* action keeps the stack from growing without bounds. *Reduce* actions are appropriate when the parser has seen the right-hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule, replacing the right-hand side with the left-hand side. You may have to consult the lookahead token to decide whether to *reduce*. The default action (represented by a dot (.)) is often a *reduce* action.

*Reduce* actions are associated with individual grammar rules. Grammar rules can also contain small integer numbers, which can be confusing. The following action refers to grammar rule 18:

```
.        reduce 18
```

while the following action refers to state 34:

```
IF       shift 34
```

Suppose the rule being reduced is as follows:

```
A : x y z ;
```

The *reduce* action depends on the left-hand symbol (A in this case) and the number of symbols on the right-hand side (three in this case). To *reduce*, first pop off the top three states from the stack. In general, the number of states popped equals the number of symbols on the right side of the rule. In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose.

After popping these states, you uncover the state that the parser was in before it began to process the rule. Using this uncovered state and the symbol on the left side of the rule, do a shift of A. A new state is obtained and pushed onto the stack, and parsing continues.

There are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, so the processing of the left-hand symbol is called a *goto* action. In particular, the lookahead token is cleared by a shift and is not affected by a *goto*. In any case, the uncovered state contains an entry such as the following that causes state 20 to be pushed onto the stack and become the current state:

```
A        goto 20
```

In effect, the *reduce* action pops the states off the stack until it goes back to the state where the right-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off of the stack; the uncovered state is, in fact, the current state.

The *reduce* action is also important in the treatment of actions and values that you supply. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack running in parallel holds the values returned from the lexical analyzer and the actions. Then, when a shift takes place, the *yylval* external variable is copied onto the value stack. After the values are returned from the user code, the *reduce* action is carried out. When the *goto* action is done, the *yyval* external variable is copied onto the value stack. The pseudo-variables $1, $2, and so on, refer to the value stack.

The other two parser actions are simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing; error recovery (as opposed to the detection of error) is described in ''Preparing Specifications.''

Consider the following example:

```
%token DING DONG DELL
%%
rhyme : sound  place
      ;
sound : DING  DONG
      ;
place : DELL
      ;
```

When you invoke **yacc** with the **-v** option, **yacc** produces a file called *y.output* with a readable description of the parser. The *y.output* file that

corresponds to the previous grammar (with some statistics stripped off the end) is:

```
state 0
        $accept : _rhyme $end

        DING shift 3
        . error

        rhyme goto 1
        sound goto 2

state 1
        $accept : rhyme_$end

        $end accept
        . error

state 2
        rhyme : sound_place

        DELL shift 5
        . error

        place goto 4

state 3
        sound : DING_DONG

        DONG shift 6
        . error

state 4
        rhyme : sound place_ (1)

        . reduce 1

state 5
        place : DELL_ (3)

        . reduce 3

state 6
        sound : DING DONG_ (2)

        . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The underscore character (_) is used to indicate what has been seen and what is yet to come in each rule. Suppose the input is:

```
DING DONG DELL
```

Follow the steps of the parser while processing this input. Initially, the current state is state 0. The parser needs to refer to the input to decide between the actions available in state 0, so it reads the first token, *DING*, which becomes the lookahead token. The action in state 0 on *DING* is *shift 3*, so the parser pushes state 3 onto the stack and clears the lookahead token. State 3 becomes the current state. The parser reads the next token, *DONG*, which becomes the lookahead token. The action in state 3 on the *DONG* token is *shift 6*, so the parser pushes state 6 onto the stack and clears the lookahead token. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead token, the parser reduces by rule 2:

```
sound : DING  DONG
```

This rule has two symbols on the right-hand side. It pops two states, 6 and 3, off the stack, uncovering state 0. By consulting the description of state 0 and by looking for a *goto* on *sound*, it obtains:

```
sound goto 2
```

The parser then pushes state 2 onto the stack, making it the current state.

Now the parser reads *DELL*, the next token in state 3. Since the action is *shift 5*, the parser pushes state 5 onto the stack and clears the lookahead token. The stack now contains 0, 2, and 5. In state 5, the action is *reduce* by rule 3, which has one symbol on its right-hand side. So the parser pops state 5 off the top of the stack, uncovering state 2. On *place*, the left-side of rule 3, the *goto* in state 2 is state 4. The stack now contains 0, 2, and 4. In state 4, the action is *reduce* by rule 1, which has two symbols on its right-hand side. Therefore, the parser pops states 2 and 4 off the top of the stack, uncovering state 0. In state 0, a *goto* on *rhyme* causes the parser to enter state 1, where it reads the input and obtains the endmarker $*end* in the *y.output* file. When the parser sees the endmarker, the action in state 1 is to accept it, successfully completing the parse.



You should consider how the parser works when confronted with incorrect strings like *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, and so forth. Once you understand this process, you will be prepared for more complicated problems.

## 6.4 Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the following

grammar rule forms an arithmetic expression by putting two expressions together with a minus sign between them:

*expr* : *expr* '-' *expr*

Unfortunately, this grammar rule does not completely specify how complex inputs should be structured. For instance, if the input is:

*expr* - *expr* - *expr*

the rule allows this input to be structured as either

( *expr* - *expr* ) - *expr*

or:

*expr* - ( *expr* - *expr* )

(The first is called left association; the second, right association).

The **yacc** program detects such ambiguities when it is attempting to build the parser. Suppose the parser receives the following input:

*expr* - *expr* - *expr*

When the parser has read the second *expr*, the following input matches the right-hand side of the previous grammar rule:

*expr* - *expr*

The can reduce the input by applying this rule; then, the input is reduced to *expr* (the left-hand side of the rule). The parser reads the final part of the input:

- *expr*

and reduces again. This takes the interpretation to be left-associative.

Alternatively, when the parser sees:

*expr* - *expr*

it can defer the immediate application of the rule and continue reading the input until it sees:

*expr* - *expr* - *expr*

It can then apply the rule to the rightmost of the three symbols, reduce the input by *expr*, and leave the following:

*expr - expr*

Now the rule can be reduced once more and take the right-associative interpretation. Thus, once it reads the following, the parser can do one of two legal things, a *shift* or a *reduce*, with no way of deciding between them:

*expr - expr*

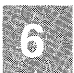
This is called a *shift/reduce* conflict. The parser can also have a choice of two legal reductions; this is called a *reduce/reduce* conflict. Note that there are never any *shift/shift* conflicts.

When there are *shift/reduce* or *reduce/reduce* conflicts, **yacc** still produces a parser by selecting one of the valid steps whenever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating* rule.

The **yacc** program invokes two disambiguating rules by default:

1. In a *shift/reduce* conflict, the default is to do the *shift.*
2. In a *reduce/reduce* conflict, the default is to *reduce* by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred, whenever there is a choice, in favor of shifts. Rule 2 gives you crude control over the behavior of the parser in this situation, but you should try to avoid *reduce/reduce* whenever possible.




Conflicts can arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than **yacc** can construct. Using actions within rules can also cause conflicts if the action must be completed before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate and leads to an incorrect parser. For this reason, **yacc** always reports the number of *shift/reduce* and *reduce/reduce* conflicts resolved by rule 1 and rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read without conflicts. This is why most previous parser generators have considered conflicts to be fatal errors. Experience has shown that rewriting unnaturally produces slower parsers. Thus, **yacc** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an if-then-else construction:

```
stat : IF '(' cond ')' stat
     | IF '(' cond ')' stat ELSE stat
     ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule is called the simple-if rule, and the second, the if-else rule.

These two rules form an ambiguous construction, because input of the form:

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

is structured according to these rules in one of two ways:

```
IF ( C1 ) {
          IF ( C2 ) S1
          }
ELSE S2
```

or:

```
IF ( C1 ) {
          IF ( C2 ) S1
          ELSE S2
          }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the *IF* immediately preceding the *ELSE*. In this example, consider the situation where the parser sees the following and is looking at the *ELSE*:

```
IF ( C1 ) IF ( C2 ) S1
```

The parser can immediately reduce by the simple-if rule to get the following:

```
IF ( C1 ) stat
```

Then read the remaining input:

```
ELSE S2
```

and *reduce* the following by the if-else rule:

```
IF ( C1 ) stat ELSE S2
```

This leads to the first of the previous groupings of the input.

On the other hand, if the *ELSE* is shifted, *S2* is read, and the right-hand portion of the following is reduced by the if-else rule:

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

You get the following, which you can reduce by the simple-if rule:

```
IF ( C1 ) stat
```

This leads to the second of the previous groupings of the input, which is the preferred grouping.

Once again, the parser can do two valid things because there is a *shift/reduce* conflict. The application of disambiguating rule 1 tells the parser to *shift* in this case, which produces the preferred grouping.

This *shift/reduce* conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, conflicts are frequent, each of them associated with an input symbol and a set of previously read inputs. These inputs are characterized by the state of the parser.

The **yacc** conflict messages are best understood by examining the **-v** (verbose) option output file. For example, the output corresponding to the preceding conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

          stat : IF ( cond ) stat_        (18)
          stat : IF ( cond ) stat_ELSE stat

          ELSE    shift 45
                  reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, including the parser actions and

the grammar rules active in the state. Recall that the underline marks the portion of the grammar rules which has been seen. Thus, in the example, in state 23 the parser has seen input corresponding to:

`IF` ( *cond* ) *stat*

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it may *shift* into state 45, which will have as part of its description the following line because the *ELSE* has been shifted in this state:

*stat* : IF ( *cond* ) *stat* ELSE_*stat*

In state 23, the alternative action, described by dot "." , is taken if the input symbol is not mentioned explicitly in the above actions. Thus, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

*stat* : IF '(' *cond* ')' *stat*

Notice that the numbers following *shift* commands refer to other states, while the numbers following *reduce* commands refer to grammar rule numbers.

In the *y.output* file, the rule numbers are printed after those rules that can be reduced. In most states, there will be, at most, a *reduce* action possible. This *reduce* action is usually the default command. If you encounter unexpected *shift/reduce* conflicts, you should look at the verbose output to decide whether the default actions are appropriate.

## 6.5 How to Handle Operator Precedences

The one common situation where the rules for resolving conflicts are not sufficient is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions are described by the precedence levels for operators, together with information about left or right associativity. You can use ambiguous grammars with appropriate disambiguating rules to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic idea is to write grammar rules of the following forms for all desired binary and unary operators:

*expr* : *expr* OP *expr*

and:

*expr* : UNARY *expr*

This method creates an ambiguous grammar with many parsing conflicts. You can set disambiguating rules for the precedence, or binding strength, of all the operators and the associativity of the binary operators. This information should let **yacc** resolve the parsing conflicts in accordance with these rules and construct a parser that realizes the preferred precedences and associativities.

These precedences and associativities are attached to tokens in the declarations section. They are attached by a series of lines beginning with a **yacc** keyword: **%left**, **%right**, or **%nonassoc**, followed by a list of tokens. All of the tokens on the same line should have the same precedence and associativity; the lines are listed in order of increasing precedence or binding strength.

Thus the following describes the precedences and associativities of the four arithmetic operators:

```
%left '+' '-'
%left '*' '/'
```

Plus and minus are left-associative and have lower precedence than asterisk (*) and slash (/), which are also left-associative. The **%right** keyword describes right-associative operators, and the **%nonassoc** keyword describes operators, like the .LT. operator in FORTRAN, all of which may not associate with themselves. Thus, the following is illegal in FORTRAN and would be described in **yacc** with the **%nonassoc** keyword:

```
A .LT. B .LT. C
```

6

As an example of the behavior of these declarations, you could use the following description:

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr    : expr '=' expr
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | NAME
        ;
```

to structure the input:

```
a = b = c*d - e - f*g
```

as:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When you use this method, you must give a precedence to unary operators. Sometimes unary operators and binary operators have the same symbolic representations with different precedences. An example is the unary and binary symbol '-'. The unary minus gives the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. You can use the keyword **%prec** to change the precedence level associated with a particular grammar rule. The **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon (;), followed by a token name or literal. The **%prec** keyword causes the precedence of the grammar rule to become that of the following token name or literal. For example, to give unary minus the same precedence as multiplication, you might use the following rule:

```
%left '+' '-'
%left '*' '/'

%%

expr    : expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | '-' expr %prec '*'
        | NAME
        ;
```

If you declare a token by **%left**, **%right**, and **%nonassoc**, you need not declare it by **%token** as well.

The precedences and associativities which **yacc** uses to resolve parsing conflicts give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.

2. Each grammar rule has a precedence and an associativity, which are those of the precedence and associativity of the last token or literal in the body of the rule. If you use the **%prec** construction, it overrides this default. Some grammar rules may have no precedence and associativity.

3. When there is a *reduce/reduce* conflict, or there is a *shift/reduce* conflict and either the input symbol or the grammar rule has no precedence and associativity, then the parser uses the two disambiguating rules given at the beginning of the section and reports the conflicts.

4. If there is a *shift/reduce* conflict, and both the grammar rule and the input character have a precedence and an associativity, then the conflict resolves in favor of the action (*shift* or *reduce*) with the higher precedence. If the precedences are the same, the associativity is used; left-associative implies *reduce*, right-associative implies *shift*, and nonassociative implies *error*.

Conflicts resolved by precedence are not counted in the number of *shift/reduce* and *reduce/reduce* conflicts reported by **yacc**. So mistakes in the specification of precedences may disguise errors in the input grammar. Be sparing with precedences, using them exactly as described here, until you gain some experience. The *y.output* file is useful in determining whether the parser is actually doing what you intended it to do.

## 6.6 Error Handling and Recovery

Error handling can be difficult, especially since many of the problems you will see are semantic ones. When an error is found, for example, you may have to reclaim parse-tree storage, delete or alter symbol-table entries, or set switches to avoid generating further output.

You will seldom need to stop processing when you find an error. You should try to continue scanning the input to find further syntax errors. However, this may lead to the problem of getting the parser restarted after an error. A class of algorithms to perform this scanning discards a number of tokens from the input string and adjusts the parser so that input can continue.

To allow you some control over this process, **yacc** provides a simple, effective feature. The token name *error* which is reserved for error handling, can be used in grammar rules to suggest places where errors are expected and where recovery might take place. The parser pops its stack until it enters a state where the token *error* is legal. It then treats the token *error* as the current lookahead token and performs the action encountered. The parser then resets the lookahead token to the token that caused the error. If you do not specify any special error rules, the processing halts when **yacc** detects an error.

To prevent a cascade of error messages, the parser, after detecting an error, remains in error state until it has read and shifted three tokens. If

the parser detects an error when it is already in error state, it gives no message and deletes the input token.

As an example, a rule of the following form would, for every syntax error, make the parser to skip over the statement in which the error is seen:

```
stat : error
```

More precisely, the parser scans ahead, looking for three tokens that might legally follow a statement, and starts processing at the first of these; if the beginnings of statements are not sufficiently distinctive, the parser may make a false start in the middle of a statement and end up reporting a second error where there is none.

You can use actions with these special error rules. With these actions, you might, for example, reinitialize tables or reclaim symbol table space.

These preceding error rules are general and difficult to control. You can use easier rules such as:

```
stat : error ';'
```

Here, when it detects an error, the parser tries to skip over the statement by skipping to the next ´;´. All tokens after the error and before the next ´;´ cannot be shifted and are discarded. When the parser sees ´;´, it reduces this rule and cleans up any action associated with it.

Another form of error rule arises in interactive applications, where you may want to reenter a line after an error. A possible error rule might be:

```
input : error '\n' { printf( "Reenter line: "); } input
                   { $$ = $4;}
```

There is one potential difficulty with this approach. The parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is clearly unwanted. For this reason, there is a method that you can use to force the parser to believe that it has fully recovered from an error. In an action, the following statement resets the parser to its normal mode:

```
yyerrok ;
```

It would be better to rewrite the last example as:

```
input  :  error '\n'
                    { yyerrok;
                      printf( "Reenter last line: " ); }
                    input
                    { $$ = $4; }
                    ;
```

As mentioned previously, the token seen immediately after the *error* symbol is the input token where the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might assume the job of finding the correct place to resume input. In this case, you must clear the previous lookahead token. In an action, the following statement will have this effect:

```
yyclearin ;
```

For example, suppose the action after the error tries to call a sophisticated resynchronization routine (supplied by you) that attempts to advance the input to the beginning of the next valid statement. After this routine is called, **yylex** returns the first token in a legal statement. The parser would have to discard the illegal token and reset the error state. You could do this with the following rule:

```
stat  :  error
              { resynch();
                yyerrok ;
                yyclearin ; }
              ;
```

These methods are crude, but they do allow for a simple, effective recovery of the parser from many errors. Moreover, you can get control to deal with the error actions required by other portions of the program.

## 6.7 The yacc Environment

When you input a specification to **yacc**, the output goes to a file of C programs, called *y.tab.c* on most systems. The integer-valued function produced by **yacc** is named **yyparse.** When it is called, it in turn repeatedly calls **yylex**, the lexical analyzer that you supply to obtain input tokens. Eventually, either the parser detects an error, in which case (if no error recovery is possible) **yyparse** returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts its input. (In this case, **yyparse** returns the value 0.)

You must set up much of the environment for this parser to obtain a working program. For example, as with every C program, a program called **main** must be defined that eventually calls **yyparse**. In addition, the **yyerror** routine prints a message when the parser detects a syntax error.

You must supply these two routines in one form or another. The **yacc** program has default versions of **main** and **yyerror** in a library, which can simplify the initial learning process. The name of this library, which is system dependent, is accessed in many systems by a **-ly** argument to the loader. The following shows the sources of two simple default programs:

```
main(){
        return( yyparse() );
        }
```

and:

```
# include <stdio.h>

yyerror(s) char *s; {

        fprintf( stderr, "%s\n", s );
        }
```

The argument to **yyerror** is a string containing an error message, usually the string *syntax error*. The average application will want to do better than this. Ordinarily, you should keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable, *yychar*, contains the lookahead token number at the time of the error's detection; this can be useful for giving better diagnostics. Since you will probably supply the **main** program, (for example, to read arguments) the **yacc** library is useful only in small projects or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If you set it to a nonzero value, the parser outputs a verbose description of its actions, including a discussion of which input symbols have been read and what the parser actions are. Depending on the operating environment, you may be able to set this variable by using a debugging system.

## 6.8 Preparing Specifications

This section contains various hints for preparing specifications that are efficient, easy to change, and clear. The individual subsections are independent.

### 6.8.1 Input Style

The rules for input style are:

1. Use uppercase letters for token names, lowercase letters for non-terminal names.

2. Put grammar rules and actions on separate lines. This lets you change one without changing the other.

3. Put all rules with the same left-hand side together. Put the left-hand side in only once, and every following rule begin with a vertical bar.

4. Put a semicolon only after the last rule with a given left-hand side, and on a separate line. This lets you add new rules easily.

5. Indent rule bodies by two tab stops and action bodies by three tab stops.

The examples in the text of this section follow this style (where space permits). You must make up your own mind about these stylistic questions; however, the key problem is using these rules in the middle of extensive action code.

### 6.8.2 Left Recursion

The algorithm used by the **yacc** parser encourages so-called left recursive grammar rules, or rules of the form:

*name* : *name* rest_of_rule ;

These rules arise frequently when you write specifications for lists and sequences:

```
list  : item
      | list ',' item
      ;
```

and:

```
seq  : item
     | seq item
     ;
```

In each of these cases, the parser reduces the first rule for the first item only, and the second rule is reduced for the second and all succeeding items.

With right-recursive rules, such as the following, the parser would be a bit bigger, and the items would be seen and reduced from right to left:

```
seq : item
    | item  seq
    ;
```

More seriously, an internal stack in the parser would be in danger of overflowing if the parser were to read a very long sequence. Thus, you should use left recursion wherever applicable.

Consider whether a sequence with zero elements has any meaning and, if so, consider writing the sequence specification with an empty rule:

```
seq : /* empty */
    | seq item
    ;
```

Once again, the first rule is reduced exactly once, before the first item is read, and then the second rule is reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts may arise if you ask **yacc** to determine which empty sequence it has seen when it hasn't seen enough to know.

### 6.8.3 Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. For another, names might be entered into a symbol table in declarations, but not in expressions.

To handle this situation, you can create a global flag that is examined by the lexical analyzer and set by actions. Let's say a program consists of 0 or more declarations followed by 0 or more statements:

```
%{
  int dflag;
%}
  ...  other declarations ...

%%

prog     : decls   stats
         ;

decls    : /* empty */
                 {        dflag = 1;   }
         | decls declaration
         ;

stats    : /* empty */
                 {        dflag = 0;   }
         | stats statement
         ;
     ...  other rules ...
```

The *dflag* flag is now 0 when reading statements and 1 when reading declarations, except for the first token in the first statement. The parser must see this token before it can tell that the declaration section has ended and the statements have begun. In many cases, this single-token exception does not affect the lexical scan.

This approach can be overdone. Nevertheless, it represents a way of doing some things that are difficult to do otherwise.

### 6.8.4 Handling Reserved Words

Some programming languages permit you to use words like *if*, that are normally reserved as label or variable names, provided that this does not conflict with the legal use of these names in the programming language. This substitution is extremely hard to do in the framework of **yacc**; it is difficult to pass information to the lexical analyzer telling it, "This instance of 'if' is a keyword, and the next instance is a variable." You should try to reserve keywords and not use them as variable names.

## 6.9 Advanced Topics

This section discusses several advanced features of **yacc**.

### 6.9.1 Simulating Error and Accept in Actions

You can simulate the parsing actions of *error* and *accept* in an action by using the macros, YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0. YYERROR causes the parser to behave as if the current input symbol were a syntax error; *yyerror* is called, and error recovery takes place. You can use these methods to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

### 6.9.2 Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The method is the same as with ordinary actions; you use a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider:

```
sent    : adj  noun  verb  adj  noun
                { look at the sentence ... }
        ;

adj     : THE    { $$ = THE; }
        | YOUNG  { $$ = YOUNG; }
        ...
        ;

noun    : DOG    { $$ = DOG; }
        | CRONE  { if( $0 == YOUNG ){
                        printf( "what?\n" );
                        }
                   $$ = CRONE;
                 }
        ;
        ...
```

In the action following the word *CRONE*, a check is made to ensure that the preceding shifted token was not *YOUNG*. Obviously, this is only possible when a great deal is known about what might precede the *noun* symbol in the input. Nevertheless, this method can save you a great deal of trouble, especially when you want to exclude a few combinations from an otherwise regular structure.

### 6.9.3 Supporting Arbitrary Value Types

By default, actions and lexical analyzers return values that are integers. The **yacc** program can also support values of other types, including structures. In addition, **yacc** keeps track of the types and inserts appropriate union member names so that the resulting parser will be strictly type checked. The **yacc** value stack is declared to be a *union* of the various types of values desired. You declare the union and associate union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $*n* construction, **yacc** automatically inserts the appropriate union name so that no unwanted conversions will take place. In addition, type-checking commands such as **lint**(C) will be less verbose.

You can use three methods to provide for this typing. First, you must define the *union*, since other programs, notably the lexical analyzer, must know about the union-member names. Second, you must associate a union-member name with tokens and nonterminals. Finally, you can describe the type of those few values which **yacc** cannot easily determine.

To declare the union, you include the following in the declaration section:

```
%union {
        body of union1, body of union2
        }
```

This declares the **yacc** value stack and the external variables *yylval* and *yyval* to have types equal to this union. If you invoked **yacc** with the **-d** option, you copy the union declaration onto the *y.tab.h* file. Alternatively, you can declare the union in a header file, and use a typedef to define the variable *YYSTYPE* to represent this union. Thus, your header file might also have said:

```
typedef union {
        body of union1, body of union2
        } YYSTYPE;
```

You must include the header file in the declarations section using %{ and %}.

Once you have defined *YYSTYPE*, you must associate the union-member names with the various terminal and nonterminal names. You can use the following construction to indicate a union member name:

< *name* >

If this follows one of the **%token**, **%left**, **%right**, and **%nonassoc** keywords, the union member name is associated with the tokens listed. Thus, the following causes any reference to values returned by these two tokens to be tagged with the union-member name *optype*:

```
%left  <optype>  '+'  '-'
```

You use another keyword, **%type**, similarly to associate union-member names with nonterminals:

```
%type  <nodetype> expr stat
```

There are a couple of cases where these methods are insufficient. If there is an action within a rule, the value returned by this action has no predefined type. Similarly, references to left context values (such as $0) leave **yacc** with no easy way of determining the type. In this case, you can impose a type on the reference by inserting a union member name between < and > immediately after the first $. An example of this usage is:

```
rule : aaa { $<intval>$ = 3; } bbb
       { fun( $<intval>2, $<other>0 ); }
  ;
```

There is little justification for this syntax, but the situation does arise occasionally.

For more information, see "Accessing Values in Enclosing Rules."

The facilities described in this subsection are not triggered until you use them. In particular, the use of **%type** will turn on these facilities. When you use them, there is a fairly strict level of checking. For example, if you use $$ or $*n* to refer to something with no defined type, it is diagnosed as an error. If you do not trigger these facilities, the **yacc** value stack is used to hold *int*. A sample specification is shown in the following section.

### 6.9.4 yacc Input Syntax

This section describes the **yacc** input syntax as a **yacc** specification. The **yacc** input-specification language is an LR(2) grammar. The language becomes complex when an identifier is seen in a rule immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule, which has an action embedded in it.

As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (such as skipping blanks, Newline or comments) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIER, but never as part of C_IDENTIFIER.

## Example

```
              /* grammar for the input to yacc */

          /*  basic  entities  */
%token IDENTIFIER        /* includes identifiers and literals */
%token  C_IDENTIFIER     /* identifier followed by colon      */
%token  NUMBER           /* [0-9]+     */

          /* reserved words:  %type => TYPE, %left => LEFT, etc. */

%token  LEFT  RIGHT  NONASSOC  TOKEN  PREC  TYPE  START  UNION

%token  MARK    /* the %% mark */
%token  LCURL   /* the %{ mark */
%token  RCURL   /* the %} mark */

          /* ascii character literals stand for themselves */

%start  spec

%%

spec      : defs  MARK  rules  tail
          ;

tail      : MARK     { Eat up the rest of the file }
          | /* empty: the second MARK is optional */
          ;

defs      : /* empty */
          | defs  def
          ;

def       : START  IDENTIFIER
          | UNION  { Copy union definition to output }
          | LCURL  { Copy C code to output file }  RCURL
          | ndefs  rword  tag  nlist
          ;
```

(*Continued on next page.*)

**Example** (Continued)

```
rword   : TOKEN
        | LEFT
        | RIGHT
        | NONASSOC
        | TYPE
        ;

tag     : /* empty: union tag is optional */
        | '<'  IDENTIFIER  '>'
        ;

nlist   : nmno
        | nlist  nmno
        | nlist  ','  nmno
        ;

nmno    : IDENTIFIER          /* Literal illegal with %type */
        | IDENTIFIER  NUMBER  /* Illegal with %type */
        ;

        /* rules section */

rules   : C_IDENTIFIER  rbody  prec
        | rules  rule
        ;

rule    : C_IDENTIFIER  rbody  prec
        | '|'  rbody  prec
        ;

rbody   : /* empty */
        | rbody  IDENTIFIER
        | rbody  act
        ;

act     : '{'  { Copy action, translate $$, etc. } '}'
        ;

prec    : /* empty */
        | PREC  IDENTIFIER
        | PREC  IDENTIFIER  act
        | prec  ';'
        ;
```

## 6.10 Examples

This section provides some examples to illustrate the features of **yacc** described in this chapter. The first example is a simple **yacc** specification for a small desk calculator. The second example is an advanced **yacc** specification for a desk calculator that uses floating-point arithmetic.

### 6.10.1 A Simple Example

This example gives the complete **yacc** specification for a small desk calculator. The desk calculator has 26 registers, labeled *a* through *z*, and accepts arithmetic expressions made up of the operators +, -, *, /, % (mod operator), & (bitwise AND), | (bitwise OR), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise, it is. As in C, an integer that begins with 0 (zero) is likely octal; otherwise, it is decimal.

As an example of a **yacc** specification, the desk calculator shows how precedences and ambiguities are used, and demonstrates how to recover from simple errors. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; this job is better done by the lexical analyzer.

**Example**

```
%{
#  include  <stdio.h>
#  include  <ctype.h>

int  regs[26];
int  base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* precedence for unary minus */
```

*(Continued on next page.)*

## Example (Continued)

```
%%     /* beginning of rules section */

list  :  /*  empty  */
      | list  stat  '\n'
      | list  error  '\n'
              { yyerrok; }
      ;

stat  : expr
              { printf( "%d\n", $1 ); }
      |       LETTER  '='  expr
                     { regs[$1] = $3; }
      ;

expr  : '(' expr ')'
              { $$ = $2; }
        | expr '+' expr
              { $$ = $1 + $3; }
        | expr '-' expr
              { $$ = $1 - $3; }
        | expr '*' expr
              { $$ = $1 * $3; }
        | expr '/' expr
              { $$ = $1 / $3; }
        | expr '%' expr
              { $$ = $1 % $3; }
        | expr '&' expr
              { $$ = $1 & $3; }
        | expr '|' expr
              { $$ = $1 | $3; }
        | '-' expr %prec UMINUS
              { $$ = - $2; }
        | LETTER
              { $$ = regs[$1]; }
        | number
        ;

number  : DIGIT
              { $$ = $1; base = ($1==0) ? 8 : 10; }
        | number DIGIT
              { $$ = base * $1 + $2; }
        ;
```

*(Continued on next page.)*

**Example** (Continued)

```
%%      /* start of programs */

yylex() {      /*  lexical analysis routine  */
               /*  returns LETTER for a lowercase letter, */
               /*  yylval = 0 through 25  */
               /*  return DIGIT for a digit, */
               /*  yylval = 0 through 9  */
               /*  all other characters */
               /*  are returned immediately  */

        int  c;

        while( (c=getchar()) == ' ' )  { /* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
                yylval = c - 'a';
                return ( LETTER );
                }
        if( isdigit( c ) ) {
                yylval = c - '0';
                return(  DIGIT  );
                }
        return( c );
        }
```



### 6.10.2 An Advanced Example

This section describes an example of a grammar using some of the advanced features discussed in earlier sections. The desk calculator example in the previous section is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, arithmetic operations including +, -, *, /, unary -, and = (assignment), and has 26 floating point variables, *a* through *z*. Moreover, it also understands intervals, written as follows where *x* is less than or equal to *y*:

```
( x,y)
```

There are 26 interval valued variables *A* through *Z* that you can also use. Assignments return no value and print nothing, while expressions print the (floating or interval) value.

This example explores many features of **yacc** and C. Intervals, represented by a structure, which consist of the left and right endpoint

values, are stored as double-precision values. This structure is given the type name *INTERVAL*, by using *typedef*. The **yacc** value stack can also contain floating-point scalars and integers (used to index into the arrays holding the variable values). Notice that this strategy depends on the ability to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

You might also note the use of YYERROR to handle error conditions that use division by an interval containing 0 and by an interval presented in the wrong order. In effect, the error-recovery mechanism of **yacc** throws away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also uses syntax to keep track of the type (such as, scalar or interval) of intermediate expressions. Note that a scalar can be promoted automatically to an interval if the context demands an interval value. This procedure causes a large number of conflicts when the grammar is run through **yacc**: 18 *shift/reduce* and 26 *reduce/reduce* actions. You can see the problem by looking at the two input lines:

```
2.5 + ( 3.5 - 4. )
```

and:

```
2.5 + ( 3.5 , 4. )
```

Notice that 2.5 is used in an interval-valued expression in the second example, but this is not known until the comma (,) is read; by this time, 2.5 is finished, and the parser cannot go back and change it. You may need to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. You can circumvent this problem by having two rules for each binary interval-valued operator: one for when the left operand is a scalar, and one for when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion is applied automatically. However, there are still many cases where the conversion may be applied or not, leading to the previously noted conflicts. You can resolve these conflicts by listing the rules that yield scalars first in the specification file. In this way, you resolve the conflicts and keep scalar-valued expressions as scalar values until they are forced to become intervals.

This way of handling multiple types is instructive, but not generally applicable. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better for you to practice in a more normal programming language environment to keep the type information as part of the value, not as part of the grammar.

The unusual feature concerning lexical analysis is the treatment of floating point constants. The C library routine **atof** is used to convert a character string to a double-precision value. If the lexical analyzer detects an error, it responds by returning an illegal token in the grammar, provoking a syntax error in the parser, and, therefore, provoking error recovery.

**Example**

```
%{

#  include  <stdio.h>
#  include  <ctype.h>

typedef  struct  interval  {
        double  lo,  hi;
        }  INTERVAL;

INTERVAL  vmul(),  vdiv();

double    atof();

double    dreg[ 26 ];
INTERVAL  vreg[ 26 ];

%}

%start    lines

%union  {
        int  ival;
        double  dval;
        INTERVAL  vval;
        }

%token  <ival>  DREG  VREG    /* indices into dreg, vreg arrays */

%token  <dval>  CONST         /* floating point constant */

%type   <dval>  dexp          /* expression */

%type   <vval>  vexp          /* interval expression */

        /*  precedence  information  about  the  operators  */

%left   '+'  '-'
%left   '*'  '/'
%left   UMINUS       /*  precedence  for  unary  minus  */
```

(*Continued on next page.*)

6

**Example** (Continued)

```
%%

lines   : /*  empty  */
        | lines  line
        ;

line    : dexp '\n'
                { printf( "%15.8f\n", $1 ); }
        | vexp  '\n'
                { printf( "(%15.8f, %15.8f )\n", $1.lo, $1.hi ); }
        | DREG  '=' dexp '\n'
                { dreg[$1] = $3; }
        | VREG  '='  vexp  '\n'
                { vreg[$1] = $3; }
        | error '\n'
                { yyerrok; }
        ;

dexp    : CONST
        | DREG
                { $$ = dreg[$1]; }
        | dexp '+' dexp
                { $$ = $1 + $3; }
        | dexp '-' dexp
                { $$ = $1 - $3; }
        | dexp '*' dexp
                { $$ = $1 * $3; }
        | dexp '/' dexp
                { $$ = $1 / $3; }
        | '-' dexp %prec UMINUS
                { $$ = - $2; }
        | '(' dexp ')'
                { $$ = $2; }
        ;

vexp    : dexp
                { $$.hi = $$.lo = $1; }
        | '(' dexp ',' dexp ')'
                {
                  $$.lo  =  $2;
                  $$.hi  =  $4;
                  if(  $$.lo  >  $$.hi  ){
                          printf("interval out of order\n");
                          YYERROR;
                          }
                      }
```

(*Continued on next page.*)

**Example** (Continued)

```
        | VREG
                { $$ = vreg[$1]; }
        | vexp '+' vexp
                { $$.hi = $1.hi + $3.hi;
                      $$.lo = $1.lo + $3.lo; }
        | dexp '+' vexp
                { $$.hi = $1 + $3.hi;
                      $$.lo = $1 + $3.lo; }
        | vexp '-' vexp
                { $$.hi = $1.hi - $3.lo;
                      $$.lo = $1.lo - $3.hi; }
        | dexp '-' vexp
                { $$.hi = $1 - $3.lo;
                      $$.lo = $1 - $3.hi;}
        | vexp '*' vexp
                { $$ = vmul( $1.lo, $1.hi, $3 ); }
        | dexp '*' vexp
                { $$ = vmul( $1, $1, $3 ); }
        | vexp '/' vexp
                { if ( dcheck( $3 ) ) YYERROR;
                  $$ = vdiv( $1.lo, $1.hi, $3 ); }
        | dexp '/' vexp
                { if ( dcheck( $3 ) ) YYERROR;
                  $$ = vdiv( $1, $1, $3 ); }
        | '-' vexp  %prec UMINUS
                { $$.hi = -$2.lo; $$.lo = -$2.hi; }
        | '(' vexp ')'
                {       $$  =  $2;  }
        ;

%%

# define BSZ 50  /* buffer size for fp numbers */

        /* lexical analysis */

yylex(){
        register  c;
              { /* skip over blanks */ }
        while( ( c = getchar() ) == ' ' )

        if ( isupper(c) ){
                yylval.ival = c - 'A';
                return( VREG );
                }
        if ( islower(c) ){
                yylval.ival = c - 'a';
                return( DREG );
                }
```

*(Continued on next page.)*

**Example** (Continued)

```
        if( isdigit( c ) || c=='.' ){
                /* gobble up digits, points, exponents */

                char buf[BSZ+1],  *cp = buf;
                int  dot = 0,  exp = 0;

                for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

                        *cp = c;
                        if ( isdigit(c) ) continue;
                        if ( c == '.' ) {
                                if ( dot++ || exp ) return( '.' );
                            /* above causes syntax error */
                                continue;
                                }
                        if (c == 'e' ) {
                                if ( exp++ ) return( 'e' );
                            /* above causes syntax error */
                                continue;
                                }

                        /* end of number */
                        break;
                        }
                *cp = '\0';
                if( (cp-buf) >= BSZ )
                      printf( "constant too long:  truncated\n");
                else  ungetc( c, stdin );
                      /* above pushes back last char read */
                yylval.dval = atof ( buf );
                return( CONST );
                }
        return( c );
        }

INTERVAL  hilo( a, b, c, d )  double  a, b, c, d; {
        /* returns the smallest interval containing a, b, c, and  d */
        /* used by *, / routines */
        INTERVAL v;

        if( a>b ) { v.hi = a; v.lo = b; }
        else { v.hi = b; v.lo = a; }

        if( c>d ) {
                if ( c>v.hi ) v.hi = c;
                if ( d<v.lo ) v.lo = d;
                }
        else  {
                if ( d>v.hi ) v.hi = d;
                if ( c<v.lo ) v.lo = c;
                }
        return(  v  );
        }
```

(*Continued on next page.*)

**Example** (Continued)

```
INTERVAL vmul( a, b, v ) double a, b;  INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
        }

dcheck( v ) INTERVAL v; {
        if( v.hi >= 0. && v.lo <= 0. ){
                printf( "divisor interval contains 0.\n" );
                return(1);
                }
        return(0);
        }

INTERVAL  vdiv( a, b, v )  double a, b;  INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

## 6.11 Old Features Supported but Not Encouraged

This section covers synonyms and features that are supported for historical continuity, but that are not encouraged for various reasons:

1. You can delimit literals with double quotation marks (“ ”).

2. Literals can be more than one character long. If all the characters are alphabetic, numeric, or underscore, the type number of the literal is defined as if the literal did not have the quotation marks around it. Otherwise, you will have difficulty finding the value for such literals. The use of multicharacter literals is likely to mislead those unfamiliar with **yacc**, since it suggests that **yacc** is doing a job that must be actually done by the lexical analyzer.

3. In most places where ‘%’ is legal, you can use a backslash (\). In particular, the double backslash (\\) is the same as %%, \left, the same as **%left**; and so on.

4. There are a number of other synonyms:

   %< is the same as %**left**
   %> is the same as %**right**
   %**binary** and %**2** are the same as %**nonassoc**
   %**0** and %**term** are the same as %**token**
   %= is the same as %**prec**

5. Actions can also have the following form and you can drop the braces if the action is a single C statement:

```
={ ... }
```

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

# Chapter 7

# Using Signals


7.1 Introduction 7-1

7.2 Using the Signal System Call 7-1
  7.2.1 Disabling a Signal 7-2
  7.2.2 Restoring a Signal's Default Action 7-3
  7.2.3 Catching a Signal 7-4
  7.2.4 Restoring a Signal 7-6
  7.2.5 Program Example 7-7

7.3 Catching Several Signals 7-8

7.4 Controlling Execution with Signals 7-8
  7.4.1 Delaying a Signal's Action 7-9
  7.4.2 Using Delayed Signals with System Calls 7-10
  7.4.3 Using Signals in Interactive Programs 7-10

7.5 Using Signals in Multiple Processes 7-12
  7.5.1 Protecting Background Processes 7-12
  7.5.2 Protecting Parent Processes 7-13

## 7.1 Introduction

This chapter explains how to use C library functions to process signals sent to a program by the XENIX system. A signal is the system's response to an unusual condition that occurs during execution of a program, such as a user pressing the DELETE key or the system detecting an illegal operation. A signal interrupts normal execution of the program and initiates an action such as terminating the program or displaying an error message.

The **signal(S)** system call of the standard C library lets a program define the action of a signal. You can use the system call to disable a signal to prevent it from affecting the program. It can also be used to give a signal a user-defined action.

You can often use the **signal** system call with the **setjmp**(S) and **longjmp**(S) system calls to redefine and reshape the action of a signal. These functions let programs save and restore the execution state of a program, and give a program a means to jump from one state of execution to another without a complex assembly language interface.

To use the **signal** system call, you must put the following line at the beginning of the program:

```
#include <signal.h>
```

The *signal.h* file defines the various manifest constants used as arguments by the system call. To use the **setjmp** and **longjmp** system calls, you must put the following line at the beginning of the program:

```
#include <setjmp.h>
```

The *setjmp.h* file contains the declaration for the type **jmp_buf**, a template for saving a program's current execution state.

7

## 7.2 Using the Signal System Call

The **signal** system call changes the action of a signal from its current function to one an alternate one. The system call has the following form:

**signal** (*sigtype*, *ptr*)

where:

- *sigtype* is an integer or a manifest constant that defines the signal to be changed, and
- *ptr* is a pointer to the function defining the new action or a manifest constant giving a predefined action.

The **signal** system call always returns a pointer value, which defines the signal's previous action and can be used in subsequent calls to restore the signal to its previous value.

The *sigtype* can be:

| | |
|---|---|
| SIGINT | Interrupt signal caused by pressing the Delete key. |
| SIGQUIT | Quit signal caused by pressing the Quit key. |
| SIGHUP | Hang-up signal caused by hanging up the line when connected to the system by a modem. |

The *ptr* can be:

| | |
|---|---|
| SIG_IGN | No action (ignore the signal). |
| SIG_DFL | Default action. |

For more information on signal constants, see **signal**(S) in the *XENIX Programmer's Reference*.

For example, the following system call changes the action of the interrupt signal to no action:

```
signal(SIGINT, SIG_IGN)
```

The signal will have no effect on the program. The default action is usually to terminate the program.

The following sections show how to use the **signal** system call to disable, change, and restore signals.

### 7.2.1 Disabling a Signal

You can disable a signal, that is, prevent it from affecting a program, by using the SIG_IGN constant with **signal**. The system call has the following form:

**signal** (*sigtype*, **SIG_IGN**)

where *sigtype* is the manifest constant of the signal you wish to disable. For example, the following system call disables the interrupt signal:

```
signal(SIGINT, SIG_IGN);
```

You use this system call to prevent a signal from terminating a program that is executing in the background (for example, a child process that is not using the terminal for input or output). The system passes signals generated from keystrokes at a terminal to all programs that have been invoked from that terminal. This means that pressing the Delete key to stop a program that is running in the foreground will also stop a program running in the background if it has not disabled that signal. For example, in the following program fragment, **signal** is used to disable the interrupt signal for the child:

```
#include <signal.h>

main ()
{

      if ( fork() == 0) {
            signal(SIGINT, SIG_IGN);
            /* Child process. */

      }

/* Parent process. */

}
```

This call does not affect the parent process, which continues to receive interrupts as before. Note that if the parent process is interrupted, the child process continues to execute until it reaches its normal end.

7

### 7.2.2 Restoring a Signal's Default Action

You can restore a signal to its default action using the SIG_DFL constant with **signal**. The system call has the following form:

**signal** (*sigtype*, **SIG_DFL**)

where *sigtype* is the manifest constant defining the signal you wish to restore. For example, the following system call restores the interrupt signal to its default action:

```
signal (SIGINT, SIG_DFL)
```

You use this system call to restore a signal after it has been temporarily disabled to keep it from interrupting critical operations. For example, in the following program fragment, the second call to **signal** restores the signal to its default action:

```
#include <signal.h>
#include <stdio.h>

main ()
{
      FILE *fp;
      char *record[BUF], filename[MAX];

      signal (SIGINT, SIG_IGN);
      fp = fopen(filename, "a");
      fwrite(fp, BUF, record, 512);
      signal (SIGINT, SIG_DFL);

}
```

In this example, the interrupt signal is ignored while a record is read from the file given by *fp*.

### 7.2.3 Catching a Signal

You can catch a signal and define your own action for it by providing a system call that defines the new action and giving that system call as an argument to **signal**. The function call has the following form:

**signal** (*sigtype*, *newptr*)

where:

- *sigtype* is the manifest constant defining the signal to be caught, and
- *newptr* is a pointer to the function defining the new action.

For example, the following signal system call changes the action of the interrupt signal to the action defined by the function **catch**:

```
signal(SIGINT, catch)
```

This signal call might be used to let a program do additional processing before terminating. In the following program fragment, the **catch** function defines the new action for the interrupt signal:

```
#include <signal.h>

main ()
{
        int catch ();

        printf("Press INTERRUPT key to stop.\n");
        signal (SIGINT, catch);
        while () {
                /* Body */
        }
}

catch ()
{
        printf("Program terminated.\n");
        exit(1);
}
```

The **catch**() function prints the message ''Program terminated'' before stopping the program with the **exit**(S) function.

A program can redefine the action of a signal at any time. Thus, many programs define different actions for different conditions. For example, in the following program fragment, the action of the interrupt signal depends on the return value of a function named **keytest**:

```
#include <signal.h>

main ()
{
        int catch1 (), catch2 ();

        if (keytest() == 1)
                signal(SIGINT, catch1);
        else
                signal(SIGINT, catch2);

}
```

Later, the program can change the signal to the other action or even a third action.

When using a function pointer in the **signal** call, you must make sure that the function name is defined before the call. In the program fragment shown above, **catch1** and **catch2** are explicitly declared at the beginning of the main program function. Their formal definitions are assumed to appear after the **signal** call.

### 7.2.4 Restoring a Signal

You can restore a signal to its previous value by saving the return value of a **signal** call, then using this value in a subsequent call. The signal system call has the following form:

**signal** (*sigtype*, *oldptr*)

where:

- *sigtype* is the manifest constant defining the signal to be restored, and
- *oldptr* is the pointer value returned by a previous **signal** call.

This system call is typically used to restore a signal when its previous action may be one of many possible actions. For example, in the following program fragment, the previous action depends solely on the return value of a function **keytest**:

```
#include <signal.h>

main ()
{
	int catch1(), catch2();
	int (*savesig)();

	if (keytest() == 1)
		signal(SIGINT, catch1);
	else
		signal(SIGINT, catch2);

	savesig = signal (SIGINT, SIG_IGN);
	compute();
	signal(SIGINT, savesig);

}
```

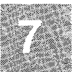
In this example, the old pointer is saved in the variable *savesig*. This value is restored after the **compute** function returns.

### 7.2.5 Program Example

This section shows by an example how to use the **signal** system call to create a modified version of **system**. In this example, **system** disables all interrupts in the parent process until the child process has completed its operation. It then restores the signals to their previous actions. You can invoke this with the following program fragment:

```
#include <stdio.h>
#include <signal.h>

system(s)    /* run command string s */
char *s;
{
      int status, pid, w;
      register int (*istat)(), (*qstat)();

      if ((pid = fork()) == 0) {
            execl("/bin/sh", "sh", "-c", s, NULL);
            exit(127);
      }
      istat = signal(SIGINT, SIG_IGN);
      qstat = signal(SIGQUIT, SIG_IGN);
      while ((w = wait(&status)) != pid && w != -1)
            ;
      if (w == -1)
            status = -1;
      signal(SIGINT, istat);
      signal(SIGQUIT, qstat);
      return(status);
}
```

Note that the parent uses the **while** statement to wait until the child's process ID *(pid)* is returned by **wait**. If **wait** returns the error code "-1", no more child processes are left, so the parent returns the error code as its own status.



## 7.3 Catching Several Signals

There are many more signals besides SIGINT, SIGQUIT, and SIGHUP. For a complete list, see **signal**(S) in the *XENIX Programmer's Reference*. In the following program fragment, all signals are caught by the same function. This function makes use of the specific signal number which is passed as a parameter by the system:

```
#include <signal.h>

main()
{
        int i;
        int catch();

        for (i = 1; i <= NSIG; ++i)
                signal(i, catch);
        /*
         * Body
         */
}

catch(sig)
int sig;
{
        signal(sig, SIG_IGN);
        if (sig != SIGINT && sig != SIGQUIT && sig != SIGHUP)
                printf("Oh, oh.  Signal %d was received.\n", sig);
        exit(1);
}
```

The constant NSIG, the total number of signals, is defined in the file *signal.h*.

Note that the first action of the **catch** function is to ignore the specific signal that was caught. This is necessary because the system automatically resets a caught signal to its default action.

## 7.4 Controlling Execution with Signals

You need not use signals solely as a means of immediately terminating a program. You can redefine many signals to delay their actions or even cause actions that terminate a portion of a program without terminating the entire program. The following sections describe ways that you can catch signals and use them to control a program.

### 7.4.1 Delaying a Signal's Action

You can delay the action of a signal by catching the signal and redefining its action to be nothing more than setting a globally-defined flag. Such a signal does nothing to the current execution of the program. Instead, the program continues uninterrupted until it can test the flag to see if a signal has been received. It can then respond according to the value of the flag.

The key to a delayed signal is that all functions return execution to the exact point at which the program was interrupted. If the function returns normally, the program continues execution just as if no signal had occurred.

Delaying a signal is especially useful in programs that must not be stopped at an arbitrary point. If, for example, a program updates a linked list, you can delay the action of a signal to prevent it from interrupting the update and destroying the list. In the following program fragment, the **delay** function, used to catch the interrupt signal, sets the globally-defined flag *sigflag* and returns immediately to the point of interruption:

```
#include <signal.h>
int sigflag;

main ()
{
        int delay ();
        int (*savesig) ();
        extern int sigflag;

        signal(SIGINT, delay);  /* Delay the signal. */
        updatelist();
        savesig = signal(SIGINT, SIG_IGN);  /* Disable the signal. */
        if (sigflag)
                /* Process delayed signals if any. */
}

delay ()
{
        extern int sigflag;

        sigflag=1;
}
```

In this example, if the signal is received while **updatelist** is executing, it is delayed until after **updatelist** returns. Note that the interrupt signal is disabled before processing the delayed signal to prevent a change to *sigflag* when it is being tested.

Note that the system automatically resets a signal to its default action immediately after the signal is processed. If your program delays a signal, make sure that you redefine the signal after each interrupt. Otherwise, the default action will be taken on the next occurrence of the signal.

### 7.4.2 Using Delayed Signals with System Calls

When you use a delayed signal to interrupt the execution of a XENIX system function, such as **read** or **wait**, the system forces the function to stop and return an error code. This action, unlike actions taken during execution of other functions, discards all processing performed by the system function. A serious error can occur if a program interprets a system-function error caused by delayed signals as a normal error. For example, if a program receives a signal when reading the terminal, all characters read before the interruption are lost, making it appear as though no characters were typed.

Whenever a program intends to use delayed signals during calls to system calls, the program should include a check of the function return values to ensure that an error was not caused by an interruption. In the following program fragment, the program checks the current value of the *intflag* interrupt flag to make sure that the EOF value returned by **getchar** actually indicates the end of the file:

```
if (getchar() == EOF)
     if (intflag)
          /* EOF caused by interrupt */
     else
          /* true end-of-file */
```

### 7.4.3 Using Signals in Interactive Programs

You can use signals in interactive programs to control the execution of their various commands and operations. For example, you can use a signal in a text editor to interrupt the current operation (such as, displaying a file) and return the program to a previous operation (for instance, waiting for a command).

To provide this control, the function that redefines the signal's action must be able to return execution of the program to a meaningful location, not just to the point of interruption. The standard C library provides two system calls to do this: **setjmp** and **longjmp**. The **setjmp** system call saves a copy of a program's execution state. The **longjmp** system call changes the current execution state to a previously saved state. The

system calls cause a program to continue execution at an old location with old register values and status as if no operations had been performed between the time the state was saved and the time it was restored.

The **setjmp** system call has the following form:

**setjmp** (*buffer*)

*Buffer* is the variable to receive the execution state. It must be declared explicitly with type **jmp_buf** before it is used in the call. For example, in the following program fragment, **setjmp** copies the execution of the program to the variable *oldstate* defined with type **jmp_buf**:

```
jmp_buf oldstate;


setjmp(oldstate);
```

Note that after a **setjmp** call, the *buffer* variable contains values for the program counter, the data and address registers, and the process status. You must not modify these values.

The **longjmp** function has the following form:

**longjmp** (*buffer*)

*Buffer* is the variable containing the execution state. It must contain values previously saved with a **setjmp** system call. The system call copies the values in the **buffer** variable to the program counter, data and address registers, and the process status table. Execution continues as if it had just returned from the **setjmp** system call which saved the previous execution state. For example, in the following program fragment, **setjmp**

7

saves the execution state of the program at the location just before the main processing loop and **longjmp** restores it on an interrupt signal:

```
#include <signal.h>
#include <setjmp.h>

main()
{
      int onintr();

      setjmp(sjbuf);
      signal(SIGINT, onintr);

      /* main processing loop */
}

onintr ()
{
printf("\nInterrupt\n");
longjmp(sjbuf);
}
```

In this example, the action of the interrupt signal as defined by *onintr* is to print the message ''Interrupt'' and restore the old execution state. When an interrupt signal is received in the main processing loop, execution passes to *onintr,* which prints the message, then passes execution back to the main program function, making it appear as though control is returning from the **setjmp** system call.

## 7.5 Using Signals in Multiple Processes

The XENIX system passes all signals generated at a given terminal to all programs invoked at that terminal. This means that a program has potential access to a signal even if that program is executing in the background or as a child to some other program. The following sections explain how signals can be used in multiple processes.

### 7.5.1 Protecting Background Processes

Any program that has been invoked and followed by the shell's background symbol (&) is executed as a background process. Such programs usually do not use the terminal for input or output. Also, they complete their tasks silently. Since these programs do not need additional input, the shell automatically disables the signals before executing the program. This means signals generated at the terminal do not affect execution of the program. This is how the shell protects the program from signals intended for other programs invoked from the same terminal.

In some cases, a program that has been invoked as a background process can also attempt to catch its own signals. If it succeeds, the protection from interruption given to it by the shell is defeated, and signals intended for other programs will interrupt the program. To prevent this, any program which is intended to be executed as a background process should test the current state of a signal before redefining its action. A program should redefine a signal only if the signal has not been disabled. For example, in the following program fragment, the action of the interrupt signal is changed only if the signal is not currently being ignored:

```
#include <signal.h>

main()
{
        int catch();

        if (signal(SIGINT, SIG_IGN) != SIG_IGN)
                signal(SIGINT, catch);

        /* Program body. */
}
```

This step lets a program continue to ignore signals if it is already doing so, and changes the signal if it is not.

### 7.5.2 Protecting Parent Processes

A program can create and wait for a child process that catches its own signals if and only if the program protects itself by disabling all signals before calling the **wait** function. By disabling the signals, the parent process prevents signals intended for the child processes from terminating the parent process' call to **wait**. This prevents serious errors that may result if the parent process continues execution before the child processes are finished.

For example, in the following program fragment, the interrupt signal is disabled in the parent process immediately after the child process is created:

```
#include <signal.h>

main ()
{
        int (*saveintr)();

        if (fork () == 0)
                execl( ... );

        saveintr = signal (SIGINT, SIG_IGN);
        wait( &status );
        signal (SIGINT, saveintr);
}
```

The signal's action is restored after the **wait** function returns normal control to the parent.

# Chapter 8

# adb: A Program Debugger

8.1 Introduction 8-1

8.2 Starting and Stopping adb 8-1
- 8.2.1 Starting with a Program File 8-1
- 8.2.2 Starting with a Core Image File 8-2
- 8.2.3 Starting adb with Data Files 8-3
- 8.2.4 Starting with the Write Option 8-3
- 8.2.5 Starting with the Prompt Option 8-4
- 8.2.6 Leaving adb 8-4

8.3 Displaying Instructions and Data 8-4
- 8.3.1 Forming Addresses 8-5
- 8.3.2 Forming Expressions 8-5
- 8.3.3 Choosing Data Formats 8-11
- 8.3.4 Using the = Command 8-13
- 8.3.5 Using the ? and / Commands 8-14
- 8.3.6 An Example: Simple Formatting 8-15

8.4 Debugging Program Execution 8-16
- 8.4.1 Executing a Program 8-17
- 8.4.2 Setting Breakpoints 8-18
- 8.4.3 Displaying Breakpoints 8-19
- 8.4.4 Continuing Execution 8-19
- 8.4.5 Stopping a Program with Interrupt and Quit 8-20
- 8.4.6 Single-Stepping a Program 8-20
- 8.4.7 Killing a Program 8-21
- 8.4.8 Deleting Breakpoints 8-21
- 8.4.9 Displaying the C Stack Backtrace 8-21
- 8.4.10 Displaying CPU Registers 8-22
- 8.4.11 Displaying External Variables 8-23
- 8.4.12 A 286 Example: Tracing Multiple Functions 8-23
- 8.4.13 A 386 Example: Tracing Multiple Functions 8-28

8.5 Using the adb Memory Maps 8-32
- 8.5.1 Displaying the Memory Maps 8-32
- 8.5.2 Changing the Memory Map 8-35

8.5.3 Creating New Map Entries 8-35
8.5.4 Validating Addresses 8-36

8.6 Miscellaneous Features 8-37
8.6.1 Combining Commands on a Single Line 8-37
8.6.2 Creating adb Scripts 8-37
8.6.3 Setting Output Width 8-38
8.6.4 Setting the Maximum Offset 8-39
8.6.5 Setting Default Input Format 8-39
8.6.6 Using XENIX Commands 8-40
8.6.7 Computing Numbers and Displaying Text 8-40
8.6.8 An Example: Directory and Inode Dumps 8-41

8.7 Patching Binary Files 8-43
8.7.1 Locating Values in a File 8-43
8.7.2 Writing to a File 8-44
8.7.3 Making Changes to Memory 8-45

## 8.1 Introduction

The **adb(CP)** program is a debugging tool for C and assembly language programs. It carefully controls the execution of a program while letting you examine and modify it's data and text areas.

This chapter explains how to use **adb**. In particular, it explains how to:

- start the debugger
- display program instructions and data
- run, breakpoint, and single-step a program
- patch program files and memory

It also illustrates techniques for debugging C programs, and explains how to display information in non-ASCII data files.

## 8.2 Starting and Stopping adb

The **adb** program provides a powerful set of commands that lets you examine, debug, and repair executable binary files as well as examine non-ASCII data files. To use these commands, you must invoke **adb** from a shell command line and specify the file or files you wish to debug. The following sections explain how to start **adb** and describe the types of files available for debugging.

### 8.2.1 Starting with a Program File

You can debug any executable C or assembly language program file using the following form:

**adb** [*filename* ]

where *filename* is the name of the program file to be debugged. The **adb** program opens the file and prepares its text (instructions) and data for subsequent debugging. For example, the following command prepares the program named *sample* for examination and execution:

```
adb sample
```

Once started, **adb** prompts with an asterisk (*) and waits for you to enter commands. If you have given the name of a file that does not exist or is in the wrong format, **adb** will display an error message first, then wait for

8

commands. For example, suppose you invoke **adb** with the following command:

```
adb sample
```

If the file *sample* does not exist, **adb** displays the following message:

```
adb: cannot open 'sample'
```

You can also start **adb** without a filename. In this case, **adb** searches for the default file *a.out* in your current working directory and prepares it for debugging. The *a.out* executable file is created by the C compiler when a program is compiled and linked successfully. Thus, typing:

```
adb
```

is the same as typing:

```
adb a.out
```

The **adb** program displays an error message and waits for a command if the *a.out* file does not exist.

### 8.2.2 Starting with a Core Image File

The **adb** program also lets you examine the core image files of programs that caused fatal system errors. Core image files contain the contents of the CPU registers, stack, and memory areas of the program at the time the error occurred and provide a way to determine the cause of an error.

To examine a core image file with its corresponding program, you must give the names of both the core and the program file. The command line has the following form:

**adb** *programfile corefile*

where:

- *programfile* is the filename of the program that caused the error, and

- *corefile* is the filename of the core image file generated by the system.

then **adb** uses information from both files to provide responses to your commands.

If you do not give a core image file, **adb** searches for the default *core* file in your current working directory. If such a file is found, **adb** uses it regardless of whether or not the file belongs to the given program. You can prevent **adb** from opening this file by using the hyphen (-) in place of the core filename. For example, the following command prevents **adb** from searching your current working directory for a core file:

```
adb sample -
```

### 8.2.3 Starting adb with Data Files

You can use **adb** to examine a data file by giving the name of the data file in place of the program or core file. For example, to examine a data file named *outdata*, type:

```
adb outdata
```

The **adb** program opens this file and lets you examine its contents.

This method of examining files is very useful if the file contains non-ASCII data. The **adb** program provides a way to look at the contents of the file in a variety of formats and structures. Note that **adb** may display a warning when you give the name of a non-ASCII data file in place of a program file. This usually happens when the content of the data file is similar to a program file. Like core files, data files cannot be executed.

### 8.2.4 Starting with the Write Option

You can make changes and corrections in a program or data file using **adb**, if you open it for writing using the **-w** option. For example, the following command opens the program file *sample* for writing:

```
adb -w sample
```

You can then use **adb** commands to examine and modify this file.

Note that the **-w** option causes **adb** to create a given file if it does not already exist. The option also lets you write directly to memory after executing the given program. See ''Patching Binary Files.''

### 8.2.5 Starting with the Prompt Option

You can define your **adb** prompt using the **-p** option. The option has the following form:

**-p** *prompt*

where *prompt* is any combination of characters. If you use spaces, enclose the *prompt* in quotes. For example, the following command sets the prompt:

```
adb -p "Mar 10->" sample
```

The new prompt takes the place of the default prompt (*) when **adb** begins to prompt for commands.

Make sure there is at least one space between the **-p** and the new prompt; otherwise **adb** will display an error message. Note that **adb** automatically supplies a space at the end of the new prompt, so you need not.

### 8.2.6 Leaving adb

You can stop **adb** and return to the system shell using the **$q** or **$Q** command. You can also stop the debugger by pressing CTRL-D.

You cannot stop the **adb** command by pressing the Quit or Delete key. **adb** ignores Quit; Delete is caught by **adb** and causes it to wait for a new command.

## 8.3 Displaying Instructions and Data

The **adb** program provides several commands for displaying the instructions and data of a given program and the data of a given data file. The commands have the following form:

*address* [, *count* ] =*format*

*address* [, *count* ] ?*format*

*address* [, *count* ] /*format*

where:

- *address* is a value or expression giving the location of the instruction or data item,

- *count* is an expression giving the number of items to be displayed, and
- *format* is an expression defining how to display the items.

The equal sign (=), question mark (?), and slash (/) tell **adb** from what source to take the item for display.

The following sections explain how to form addresses, how to choose formats, and the meaning of each of the display commands.

### 8.3.1 Forming Addresses

In **adb**, every address has the following form:

[ *segment* :] *offset*

where:

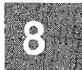
- *segment* is an expression giving the address of a specific segment of 8086/286/386 memory, and
- *offset* is an expression giving an offset from the beginning of the specified segment to the desired item.

Segments and offsets are formed by combining numbers, symbols, variables, and operators. The following are some valid addresses:

```
0:1
0x0bce:772
```

The *segment*: is optional. If not given, the most recently typed segment is used.

### 8.3.2 Forming Expressions




Expressions contain decimal, octal, and hexadecimal integers, symbols, **adb** variables, register names, and a variety of arithmetic and logical operators.

## Decimal, Octal, and Hexadecimal Integers

A decimal integer must begin with a nonzero decimal digit. An octal number must begin with a zero and may have octal digits only. Hexadecimal numbers must begin with the prefix *0x* and may contain decimal digits and the letters *a* through *f* (in both upper, and lowercase). The following are valid numbers:

| Decimal | Octal | Hexadecimal |
|---|---|---|
| 34 | 042 | 0x22 |
| 4090 | 07772 | 0xffa |

Although every decimal number is displayed with a trailing decimal point (.), you cannot use the decimal point when typing the number.

## Symbols

A symbol is the name of a global variable or function defined within the program being debugged, and is equal to the address of the given variable or function. Symbols are stored in the program's symbol table, and are available if the symbol table has not been stripped from the program file. For more information, see **strip**(CP) in the *XENIX Programmer's Reference*).

When evaluating expressions that include functions, you can evaluate a function by specifying its name or its symbol table name Symbols in the symbol table are no more than eight characters long, and those defined in C programs are given leading underscores (_). The following are examples of symbols:

```
main  _main hex2bin     __out_of
```

Note that if the spelling of any two symbols is the same (except for a leading underscore), **adb** will ignore the second symbol and allow references only to the first. For example, if both "main" and "_main" exist in a program, then **adb** accesses only the first to appear in the source and ignores the other.

When you use the question mark (?) command, **adb** uses the symbols found in the symbol table of the program file to create symbolic addresses. Thus, the command sometimes gives a function name when it displays data. This does not happen if you use the **?** command for text (instructions) and the slash (/) command for data. You cannot address local variables.

## adb Variables

The **adb** program automatically creates a set of its own variables whenever you start the debugger. These variables are set to the addresses and sizes of various parts of the program file as defined below:

| Variable | Definition |
|---|---|
| b | base address of data segment |
| d | size of data |
| e | entry address of the program |
| m | execution type |
| n | number of segments |
| s | size of stack |
| t | size of text |

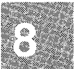
A user can access storage locations using the **adb** defined variables. The following request prints these variables:

```
$v
```

The **adb** program reads the program file to find the values for these variables. If the file does not seem to be a program file, then **adb** leaves the values undefined.

You can use the current value of an **adb** variable in an expression by preceding the variable name with a less than (<) sign. For example, the current value of the base variable *b* is:

```
<b
```



You can create your own variables or change the value of an existing variable by assigning a value to a variable name with the greater than (>) sign. The assignment has the following form:

*expression > variable-name*

where

- *expression* is the value to be assigned to the variable, and
- *variable-name* must be a single letter.

For example, the following assignment gives the hexadecimal value "0x2000" to the variable *b*:

```
0x2000>b
```

You can display the values of all currently defined **adb** variables using the **$v** command. The command lists the variable names followed by their values in the current format. The command displays any variable whose value is not zero. If a variable also has a nonzero segment value, the variable's value is displayed as an address; otherwise it is displayed as a number.

## Current Address

The **adb** program has two special variables that keep track of the last address to be used in a command and the last address to be typed with a command. The dot (.) variable, also called the current address, contains the last address to be used in a command. The double quotation marks (" ") variable contains the last address to be typed with a command. The dot and " " variables are usually the same except when you use implied commands, such as the Newline and caret (^) characters. (These automatically increment and decrement dot, but leave " " unchanged.)

You can use both the dot and the " " in any expression. The less than (<) sign is not required. For example, the following command displays the value of the current address:

```
.=
```

and the following command displays the last address to be typed:

```
"=
```

## Register Names

The **adb** program lets you use the current value of the CPU registers when evaluating expressions. You can give the value of a register by preceding its name with the less than (<) sign. The **adb** program recognizes the following register names:

| **286 Registers** | | **386 Registers** | |
|---|---|---|---|
| ax | accumulator | eax | accumulator |
| cx | counter | ecx | counter |
| dx | data | edx | data |
| bx | base | ebx | base |
| sp | stack pointer | esp | stack pointer |
| bp | base pointer | ebp | base pointer |
| si | source index | esi | source index |
| di | destination index | edi | destination index |
| es | extra segment | es | extra segment |
| cs | code segment | cs | code segment |
| ss | stack segment | ss | stack segment |
| ds | data segment | ds | data segment |
| | | fs | extra segment |
| | | gs | extra segment |
| fl | flags register | efl | flags register |
| ip | instruction pointer | eip | instruction pointer |

All 286 and 386 registers can be evaluated in expressions on XENIX 386, but only 286 registers can be evaluated in expressions on XENIX 286.

For example, the value of the 286 **ax** register can be evaluated in an expression by specifying the register as follows:

```
<ax
```

Note that you can not use register names unless either you start **adb** with a *core* file, or the program is currently being run under **adb** control.

8

## Operators

You can combine integers, symbols, variables, and register names with the following operators:

| Unary | Meaning |
|---|---|
| ~ | Not |
| - | Negative |
| * | Contents of location |

| Binary | Meaning |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| % | Integer division |
| & | Bitwise And |
| \| | Bitwise inclusive Or |
| ^ | Modulo |
| # | Round up to the next multiple |

Unary operators have higher precedence than binary operators. All binary operators have the same precedence. Thus, the following expression evaluates to 10:

```
2*3+4
```

and the following expression evaluates to 18:

```
4+2*3
```

You can change the precedence of the operations in an expression by using parentheses. For example, the following expression evaluates to 10:

```
4+(2*3)
```

---

*Note*

The **adb** program uses 32-bit arithmetic. This means that values that exceed 2,147,483,647 (decimal) are displayed as negative values.

The unary * operator treats an expression as a pointer to an address. An expression using this operator resolves to the value stored at the given address. For example, the following expression resolves to the value stored at the address ''0x1234'':

```
*0x1234
```

whereas the following is just equal to ''0x1234'':

```
0x1234
```

---

### 8.3.3 Choosing Data Formats

Data of different forms can be displayed by specifying a string of format commands. A format command is a letter that specifies the format in which data is displayed. One or more letter commands can be concatenated with an integer to specify the number of times the letter commands are displayed.

The following illustrates each letter command and associated data format displayed:

| Letter | Format |
|---|---|
| o | 2 bytes in octal |
| d | 2 bytes in decimal |
| D | 4 bytes in decimal |
| x | 2 bytes in hexadecimal |
| X | 4 bytes in hexadecimal |
| u | 2 bytes as an unsigned integer |
| f | 4 bytes in floating point |
| F | 8 bytes in floating point |
| c | 1 byte as a character |
| s | A null terminated character string |
| i | Machine instruction |
| b | 1 byte in octal |
| a | The current symbolic address |
| A | The current absolute address |
| n | A Newline |
| r | A blank space |
| t | A horizontal TAB |

A letter command can be used by itself or combined with other commands to present a combination of data in different forms.

You can use the **d**,**o**,**x**, and **u** commands to display **int** type variables; you can use **D** and **X** to display **long** variables or 32-bit values. The **f** and **F** commands can be used to display single- and double-precision floating-point numbers. The **c** command displays **char** type variables, and the **s** command is for arrays of **char** that end with a null character (null terminated strings).

The **i** command displays machine instructions in 8086/286/386 mnemonics. The **b** command displays individual bytes and is useful for displaying data associated with instructions, or the high or low bytes of registers.

You usually combine the **a**,**r**, and **n** commands with other commands to make the display more readable. For example, the following format commands display the current address after each instruction:

```
ia
```

You can precede each format with a count of the number of times you wish it to be repeated. For example, the following format commands display four ASCII characters:

```
4c
```

You can also combine format requests to provide elaborate displays. For instance, the following commands display four octal words followed by their ASCII interpretation from the data space of the core image file:

```
<b,-1/4o4^8Cn
```

In this example, the display starts at the address ``<b,'' the base address of the program's data. The display continues until the end-of-the-file since the negative count ``-1'' causes an indefinite execution of the commands until an error condition, such as the end of the file, occurs. The command **4o** displays the next four words (16-bit values) as octal numbers. The command **4^** then moves the current address back to the beginning of these four words and the **C** command redisplays them as eight ASCII characters. Finally, **n** sends a Newline character to the terminal. The **C** command causes values to be displayed as ASCII characters if they are in the range 32 to 126. If the value is in the range 0 to 31, it is displayed as an at sign (@) followed by a lowercase letter. For example, the value 0 is displayed as @. The at sign itself is displayed as a double at sign @@.

### 8.3.4 Using the = Command

The equal sign (=) command displays a given address in a given format. The command is used primarily to display instruction and data addresses in simpler form, or to display the results of arithmetic expressions. For example, typing the following displays the absolute address of the symbol ``main'' (giving the segment and offset):

```
main=A
```

Typing the following displays (in decimal) the sum of the variable *b* and the hexadecimal value *0x2000*:

```
<b+0x2000=D
```

If a count is given, the same value is repeated that number of times. For example, typing the following displays the value of ``main'' twice:

```
main,2=x
```

If no address is given, the current address is used instead. This is the same as the following command:

```
.=
```

If you do not specify a format, the previous format given for this command is used. For example, in the following sequence, both ''main'' and ''_start'' are displayed in hexadecimal:

```
main=x
_start=
```

### 8.3.5 Using the ? and / Commands

You can display the contents of a text or data segment with the **?** and / commands. The commands have the following form:

[ *address* ] [, *count* ] **?** [ *format* ]

[ *address* ] [, *count* ] / [ *format* ]

where:

- *address* is an address with the given segment,
- *count* is the number of items you wish to display, and
- *format* is the format of the items you wish to display.

You use the **?** command to display instructions in the text segment. For example, the following command displays five instructions starting at the address ''main,'' and the address of each instruction displays immediately before it:

```
main,5?ia
```

The following command displays the instructions, with no addresses other than the starting address:

```
main,5?i
```

You use the / command to check the values of variables in a program, especially variables for which no name exists in the program's symbol table. For example, the following command displays the value (in hexadecimal) of a local variable:

```
<bp-4?x
```

Local variables are generally at some offset from the address indicated by the **bp** register.

### 8.3.6 An Example: Simple Formatting

The following example illustrates how to combine formats in **?** or / commands, to display different types of values when stored together in the same program. This program has the following source statements:

```
char  str1[ ]     = "This is a character string" ;
int   one         = 1 ;
int   number              = 456 ;
long  lnum        = 1234 ;
float fpt         = 1.25 ;
char  str2[ ]     = "This is the second character string" ;

main()
{
      one = 2;
}
```

The program is compiled and stored in a file named *sample*.

To start the session, type:

```
adb sample -
```

You can display the value of each individual variable by giving its name and corresponding format in a / command. For example, typing:

```
str1/s
```

displays the contents of *str1* as a string:

```
_str1:     This is a character string
```

The following command:

```
number/d
```

displays the contents of *number* as a decimal integer:

```
_number:   456.
```

You can choose to view a variable in a variety of formats. For example, you can display the **long** variable *lnum* as a 4-byte decimal, octal, and hexadecimal number by typing the following:

```
lnum/D
_lnum:      1234
lnum/O
_lnum:      02322
lnum/X
_lnum:      0x4D2
```

You can also examine all variables as a whole. For example, if you wish to see them all in hexadecimal, type:

```
str1,5/8x
```

This command displays eight hexadecimal values on a line, and continues for five lines.

Since the data contains a combination of numeric and string values, it is worthwhile to display each value as both a number and a character to see where the actual strings are located. You can do this with one command by typing:

```
str1,5/4x4^8Cn
```

In this case, the command displays four values in hexadecimal, then the same values as eight ASCII characters. The caret (^) is used four times, immediately before displaying the characters to set the current address back to the starting address for that line.

To make the display easier to read, you can insert a tab between the values and characters, and give an address for each line by typing

```
str1,5/4x4^8t8Cna
```

## 8.4 Debugging Program Execution

The **adb** program provides a variety of commands to control the execution of programs being debugged. The following sections explain how to use these commands as well as how to display the contents of memory and registers.

Note that C compiler does not normally generate statement labels for programs. This means it is not possible to refer to individual C statements when using the debugger. In order to use execution commands effectively, you must be familiar with the instructions generated by the C compiler and how they relate to individual C statements. One useful technique is to create an assembly-language listing of your C program before using **adb**, then refer to the listing as you use the debugger. To create an assembly-language listing, use the **-S** option of the **cc** command. For more information, see **cc**(CP) in the *XENIX Programmer's Reference*.

### 8.4.1 Executing a Program

You can execute a program using the **:r** or **:R** command. The command has the following form:

[ *address* ] [,*count* ] **:r** [ *arguments* ]

[ *address* ] [,*count* ] **:R** [ *arguments* ]

where:

- *address* gives the address at which to start execution,
- *count* is the number of breakpoints you wish to skip before one is taken, and
- *arguments* are the command line arguments, such as filenames and options, that you wish to pass to the program.

If no *address* is given, then the start of the program is used. Thus, to execute the program from the beginning, type:

```
:r
```

If a *count* is given, **adb** will ignore all breakpoints until the given number have been encountered. For example, the following command causes **adb** to skip the first 5 breakpoints:

```
,5:r
```

If you specify arguments, each of them must be separated by at least one space. The arguments are passed to the program in the same way the system shell passes command-line arguments to a program. You may use the shell-redirection symbols if you wish.

The **:R** command passes the command arguments through the shell before starting program execution. This means you can use shell metacharacters in the arguments to refer to multiple files or other input values. The shell expands arguments containing metacharacters before passing them on to the program.

The :R command is especially useful if the program expects multiple filenames. For example, the following command passes the argument "[a-z]*.s" to the shell where it is expanded to a list of the corresponding filenames before being passed to the program:

```
:R [a-z]*.s
```

The **:r** and **:R** commands remove the contents of all registers and destroy the current stack before starting the program. This kills any previous copy of the program you may have been running.

### 8.4.2 Setting Breakpoints

You can set a breakpoint in a program by using the **:br** command. Breakpoints cause execution of the program to stop when it reaches the specified address. Control then returns to **adb**. The command has the following form:

*address* [, *count* ] **:br** *command*

where:

- *address* must be a valid instruction address,
- *count* is a count of the number of times you wish the breakpoint to be skipped before it causes the program to stop, and
- *command* is the **adb** command you wish to execute when the breakpoint is taken.

Breakpoints are typically set to stop program execution at a specific place in the program, such as the beginning of a function, so that the contents of registers and memory can be examined. For example, the following command sets a breakpoint at the start of the function named "main":

```
main:br
```

The breakpoint is taken just as control enters the function and before the function's stack frame is created.

A breakpoint with a count is typically used within a function, that is called several times during execution of a program, or within the instructions that correspond to a **for** or **while** statement. Such a breakpoint lets the program continue to execute until the given function or instructions have been executed for the specified number of times. For example, the following command sets a breakpoint at the fifth repetition of the function "light":

```
light,5:br
```

The breakpoint does not stop the function until it has been called at least five times.

Note that no more than 16 breakpoints at a time are allowed.

### 8.4.3 Displaying Breakpoints

You can display the location and count of each currently defined breakpoint by using the **$b** command. The command displays a list of the breakpoints given by address. If the breakpoint has a count and/or a command, these are given as well.

The **$b** command is useful if you have created several breakpoints in your program.

### 8.4.4 Continuing Execution

You can continue program execution after it has been stopped by a breakpoint by using the **:co** command. The command has the following form:

[ *address* ] [,*count*] **:co** [*signal*]

where:

- *address* is the address of the instruction at which you wish to continue execution,
- *count* is the number of breakpoints you wish to ignore, and
- *signal* is the number of the signal to send to the program. For more information, see **signal**(S) in the XENIX *Programmer's Reference*.

If you don't specify an *address*, the program starts at the next instruction after the breakpoint. If you do specify *count*, **adb** ignores the first *count* breakpoints.

8

### 8.4.5 Stopping a Program with Interrupt and Quit

You can stop program execution at any time by pressing the Delete, CTRL-\, or Quit keys. These keys stop the current program and return control to **adb**. The keys are especially useful for programs that have infinite loops or other program errors.

Note that whenever you press the Delete, CTRL-\, or Quit key to stop a program, **adb** automatically saves the signal and passes it to the program, if you start it again by using the **:co** command. This is very useful if you wish to test a program that uses these signals as part of its processing.

If you wish to continue program execution, but you do not wish to send the signals, type:

```
:co  0
```

The command argument **0** prevents a pending signal from being sent to the program.

### 8.4.6 Single-Stepping a Program

You can single-step a program, that is, execute it one instruction at a time, using the **:s** command. The command executes an instruction and returns control to **adb**. The command has the following form:

[*address* ] [, *count* ] **:s**

where:

- *address* must be the address of the instruction you wish to execute, and
- *count* is the number of times you wish to repeat the command.

If you do not specify an *address*, **adb** uses the current address. If you specify a *count*, **adb** continues to execute each successive instruction until *count* instructions have been executed. For example, the following command executes the first 5 instructions in the function *main*:

```
main,5:s
```

### 8.4.7 Killing a Program

You can kill the program you are debugging by using the **:k** command. The command kills the process created for the program and returns control to **adb**. The command is typically used to clear the current contents of the CPU registers and stack and begin the program again.

### 8.4.8 Deleting Breakpoints

You can delete a breakpoint from a program by using the **:dl** command. The command has the following form:

*address* **:dl**

where *address* is the address of the breakpoint you wish to delete.

The **:dl** command is typically used to delete breakpoints you no longer wish to use. Typing the following deletes the breakpoint set at the start of the function "main":

```
main:dl
```

### 8.4.9 Displaying the C Stack Backtrace

You can trace the path of all active functions by using the **$c** command. The command lists the names of all functions that have been called but have not yet returned control, as well as the address from which each function was called, and the arguments passed to it.

For example, the following command displays a backtrace of the C language functions called:

```
$c
```

By default, the **$c** command displays all calls. If you wish to display just a few, you must supply a count of the number of calls you wish to see. For example, the following command displays upto 25 calls in the current call path:

```
,25$c
```

Note that function calls and arguments are put on the stack after the function has been called. If you put breakpoints at the entry point to a function, the function will not appear in the list generated by the **$c** command.

You can remedy this problem by placing breakpoints a few instructions into the function.

### 8.4.10 Displaying CPU Registers

You can display the contents of all CPU registers by using the **$r** command. The command displays the name and contents of each register in the CPU as well as the current value of the program counter, and the instruction at the current address.

#### Registers for XENIX 286

The **adb** program displays registers in the following format when executing in XENIX 286; the value of each register is given in the current default format:

```
ax    0x0         fl    0x0
bx    0x0         ip    0x0
cx    0x0         cs    0x0
dx    0x0         ds    0x0
di    0x0         ss    0x0
si    0x0         es    0x0
sp    0x0         sp    0x0
0:0:  addb  al,bl
```

#### Registers for XENIX 386

The **adb** program displays registers in the following format when executing in XENIX 386; the value of each register is given in the current default format:

```
eax   0x81000     efl   0x246
ebx   0x0         eip   0x142
ecx   0x0         cs    0x3f
edx   0x8         ds    0x47
edi   0x0         es    0x47
esi   0x0         fs    0x47
ebp   0x0         gs    0x47
esp   0x7fef8     ss    0x47
0x3f:0x142:     push      ebp
```

### 8.4.11 Displaying External Variables

You can display the values of all external variables in a program by using the **$e** command. External variables are variables in your program that have global scope, or have been defined outside of any function. This may include variables that have been defined in library routines used by your program.

The **$e** command is useful whenever you need a list of the names for all available variables, or to quickly summarize their values. The command displays one name on each line with the variable's value (if any) on the same line.

For example, use the $**e** command to display the following external variables and their values in hexadecimal format in a program:

```
_environ:  0xff08
_fcnt:     0x0
_gcnt:     0x0
_hcnt:     0x0
_errno:    0x0
__bufend:  0x0
__xcstar:  0x0
__xistar:  0x0
_end:      0x0
__stdbuf:  0x1b0
_iob:      0x0
_edata:    0x0
___argv:   0xff00
__acrtus:  0x0
__xcend:   0x0
__sibuf:   0x0
__lastbu:  0x130
__xiend:   0x0
__smbuf:   0x0
__sobuf:   0x0
```

### 8.4.12 A 286 Example: Tracing Multiple Functions

The following example illustrates how to execute a program under **adb** control using XENIX 286. In particular, it shows how to set breakpoints,

start the program, and examine registers and memory. The program to be examined has the following source statements:

```
int      fcnt,gcnt,hcnt;
h(x,y)
{
    int hi; register int hr;
    hi = x+1;
    hr = x-y+1;
    hcnt++ ;
    hj:
    f(hr,hi);
}

g(p,q)
{
    int gi; register int gr;
    gi = q-p;
    gr = q-p+1;
    gcnt++ ;
    gj:
    h(gr,gi);
}

f(a,b)
{
    int fi; register int fr;
    fi = a+2*b;
    fr = a+b;
    fcnt++ ;
    fj:
    g(fr,fi);
}

main()
{
    f(1,1);
}
```

The program is compiled and stored in a file named *sample*. To start the session, type:

```
adb sample
```

This starts **adb** and opens the corresponding program file. There is no core image file.

The first step is to set breakpoints at the beginning of each function. You can do this with the **:br** command. For example, to set a breakpoint at the start of function "f," type:

```
f:br
```

You can use similar commands for the ''g'' and ''h'' functions. Once you have created the breakpoints, you can display their locations by typing:

```
$b
```

This command lists the address, optional count, and optional command associated with each breakpoint. In this case, the command displays:

```
breakpoints
count bkpt          command
1     _h
1     _g
1     _f
```

The next step is to display the first five instructions in the ''f'' function. Type:

```
f,5?ia
```

This command displays five instructions, each preceded by its symbolic address.

```
_f:               push      bp
_f+1.:            mov       bp,sp
_f+3.:            mov       ax,4
_f+6.:            call      near __chkstk
_f+9.:            push      di
_f+10.:
```

You can display five instructions in the ''g'' function without their addresses by typing:

```
g,5?i
```

The system displays:

```
_g:  push bp
     mov  bp,sp
     mov  ax,4.
     call near __chkstk
     push di
```

To begin program execution, type:

```
:r
```

then **adb** displays the following message and begins to execute:

```
sample: running
```

8

As soon as **adb** encounters the first breakpoint (at the beginning of the "f" function), it stops execution and displays the following message:

```
breakpoint _f:   push  bp
```

Since execution to this point caused no errors, you can remove the first breakpoint by typing:

```
f:dl
```

You can continue the program by typing:

```
:co
```

The **adb** program displays the following message and begins program execution at the next instruction:

```
sample: running
```

Execution continues until the next breakpoint, where **adb** displays the following message:

```
breakpoint _g:   push  bp
```

You can now trace the path of execution by typing:

```
$c
```

The display shows that only three functions are active: "f," "main," and "start":

```
_f (1., 1.)                          from _main+18.
_main (1., 5922., 5926.)     from __start+50.
__start                              from start0+5.
```

Although the breakpoint has been set at the start of function "g," it will not be listed in the backtrace until its first few instructions have been executed. To execute these instructions, type:

```
,5:s
```

then **adb** single-steps the first five instructions. Now you can list the backtrace again. Type:

```
$c
```

This time, the list shows four active functions:

```
_g (2.,3.)              from _f+39
_f (1.,1.)              from _main+18
_main (1., 5922., 5926)     from __start+50
__start ()              from start0+5
```

You can display the contents of the integer variable *fcnt* by typing:

```
fcnt/D
```

This command displays the value of *fcnt* found in memory. The number should be *1*. You can continue execution of the program and skip the first 10 breakpoints by typing:

```
,10:co
```

In response to this, **adb** starts the program and then displays the running message again. The program does not stop until adb encounters exactly 10 breakpoints, when it displays the following message:

```
breakpoint _g:   push bp
```

To show that these breakpoints have been skipped, you can display the backtrace again, by typing:

```
$c
```

For XENIX 286, your system displays:

```
_f (2., 11.)                  from _h+36.
_h (10., 9.)            from _g+38.
_g (11., 20.)           from _f+39.
_f (2., 9.)             from _h+36.
_h (8., 7.)             from _g+38.
_g (9., 16.)            from _f+39.
_f (2., 7.)             from _h+36.
_h (6., 5.)             from _g+38.
_g (7., 12.)            from _f+39.
_f (2., 5.)             from _h+36.
_h (4., 3.)             from _g+38.
_g (5., 8.)             from _f+39.
_h (2., 3.)                   from _g+38.
_g (2., 1.)                   from _f+39.
_f (1., 1.)                   from _main+18.
_main(1., 5922.,   5926.)     from __start+50.
__start()               from start0+5.
```

8

### 8.4.13 A 386 Example: Tracing Multiple Functions

The following example illustrates how to execute a program under **adb** control using XENIX 386. In particular, it shows how to set breakpoints, start the program, and examine registers and memory. The program to be examined has the following source statements:

```
int      fcnt,gcnt,hcnt;
h(x,y)
{
   int hi; register int hr;
   hi = x+1;
   hr = x-y+1;
   hcnt++ ;
   hj:
   f(hr,hi);
}

g(p,q)
{
   int gi; register int gr;
   gi = q-p;
   gr = q-p+1;
   gcnt++ ;
   gj:
   h(gr,gi);
}

f(a,b)
{
   int fi; register int fr;
   fi = a+2*b;
   fr = a+b;
   fcnt++ ;
   fj:
   g(fr,fi);
}

main()
{
   f(1,1);
}
```

The program is compiled and stored in a file named *sample*. To start the session, type:

```
adb sample
```

This starts **adb** and opens the corresponding program file. There is no core image file.

The first step is to set breakpoints at the beginning of each function. You can do this with the **:br** command. For example, to set a breakpoint at the start of function "f," type:

```
f:br
```

You can use similar commands for the "g" and "h" functions. Once you have created the breakpoints, you can display their locations by typing:

```
$b
```

This command lists the address, optional count, and optional command associated with each breakpoint. In this case, the command displays:

```
breakpoints
count bkpt          command
1     _h
1     _g
1     _f
```

The next step is to display the first five instructions in the "f" function. Type:

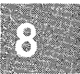
```
f,5?ia
```

This command displays five instructions, each proceeded by its symbolic address:

```
_f:        push   ebp
_f+0x1:    mov    ebp,esp
_f+0x3:    sub    esp, 0x8
_f+0x9:    push   ebx
_f+0xa:    push   edi
_f+0xb:
```

You can display five instructions in the "g" function without their addresses by typing:

```
g,5?i
```



In this case, the display is:

```
_g:   push  ebp
      mov   ebp, esp
      sub   esp, 0x8
      push  ebx
      push  edi
```

To begin program execution, type:

```
:r
```

then **adb** displays the following message and begins to execute:

```
sample: running
```

As soon as **adb** encounters the first breakpoint (at the beginning of the "f" function), it stops execution and displays the following message:

```
breakpoint _f:   push  ebp
```

Since execution to this point caused no errors, you can remove the first breakpoint by typing:

```
f:dl
```

You can continue the program by typing:

```
:co
```

then **adb** displays the following message and begins program execution at the next instruction:

```
sample: running
```

Execution continues until the next breakpoint, where **adb** displays the following message:

```
breakpoint _g:   push  ebp
```

You can now trace the path of execution by typing:

```
$c
```

The commands show that only three functions are active: "f," "main," and "start":

```
_f (0x1, 0x1)                from _main+0x15
_main(0x1, 0x187ef20, 0x187ef28) from __start+0x39
__start ()              from start0+0xc
```

Although the breakpoint has been set at the start of function "g," it will not be listed in the backtrace until its first few instructions have been executed. To execute these instructions, type:

```
,5:s
```

The **adb** program single-steps the first five instructions. Now you can list the backtrace again. Type:

```
$c
```

This time, the list shows four active functions:

```
_g (0x2,0x3)                    from _f+0x2c
_f (0x1,0x1)                    from _main+0x15
_main (0x1, 0x187ef20, 0x187ef28)     from __start+0x39
__start ()              from start0+0xc
```

You can display the contents of the integer variable *fcnt* by typing:

```
fcnt/D
```

This command displays the value of *fcnt* found in memory. The number should be *1*. You can continue execution of the program and skip the first 10 breakpoints by typing:

```
,10:co
```

now **adb** starts the program; then it displays the running message again. It does not stop the program until it encounters exactly ten breakpoints. It displays the following message:

```
breakpoint _g:   push  ebp
```

To show that these breakpoints have been skipped, you can display the backtrace again, by typing:

**$c**

8

For XENIX 386, your system displays:

```
_f (0x2,0x11)                    from _h+0x29
_h (0x10,0xf)                    from _g+0x2b
_g (0x11,0x20)                   from _f+0x2c
_f (0x2,0xf)                     from _h+0x29
_h (0xe,0xd)                     from _g+0x2b
_g (0xf,0x1c)                    from _f+0x2c
_f (0x2,0xd)                     from _h+0x29
_h (0xc,0xb)                     from _g+0x2b
_g (0xd,0x18)                    from _f+0x2c
_f (0x2,0xb)                     from _h+0x29
_h (0xa,0x9)                     from _g+0x2b
_g (0xb,0x14)                    from _f+0x2c
_f (0x2,0x9)                     from _h+0x29
_h (0x8,0x7)                     from _g+0x2b
_g (0x9,0x10)                    from _f+0x2c
_f (0x2,0x7)                     from _h+0x29
_h (0x6,0x5)                     from _g+0x2b
_g (0x7,0xc)                     from _f+0x2c
_f (0x2,0x5)                     from _h+0x29
_h (0x4,0x3)                     from _g+0x2b
_g (0x5,0x8)                     from _f+0x2c
_f (0x2,0x3)                     from _h+0x29
_h (0x2,0x1)                     from _g+0x2b
_g (0x2,0x3)                     from _f+0x2c
_f (0x1,0x1)                     from _main+0x15
_main (0x1,0x187ef20,0x187ef28)  from __start+0x39
__start () from start0+0xc
```

## 8.5 Using the adb Memory Maps

The **adb** program prepares a set of maps for the text and data segments in your program and uses these maps to access items that you request for display. The following sections describe how to view these maps, and how they are used to access the text and data segments.

### 8.5.1 Displaying the Memory Maps

You can display the contents of the memory maps using the **$m** command. The command has the following form:

**$m** [ *segment* ]

where *segment* is the number of a segment used in the program.

The command displays the maps for all segments in the program using information taken from either the program and core files or directly from memory.

### Displays for XENIX 286

If you have started **adb** but have not executed the program, the **$m** command display has the following form for XENIX 286:

```
Text Segments
Seg # File Pos     Vir Size     Phys Size     'sample' - File
63.   160.         3712.        2462.

Data Segments
Seg # File Pos     Vir Size     Phys Size     'sample' - File
71.   160.         3712.        2462.
```

If you have executed the program, the command displays the following form for XENIX 286:

```
Text Segments
Seg # File Pos     Vir Size     Phys Size     'sample' - memory
63.   160.         3712.        2462.

Data Segments
Seg # File Pos     Vir Size     Phys Size     'sample' - memory
71.   160.         3712.        2462.
```

### Displays for XENIX 386

The **$m** command has the following form for XENIX 386:

```
Text Segments       File - 'sample'
Seg # File Pos     Vir Size     Phys Size     Reloc Base
0x3f  0x400        0xb48        0xb48         0x0

Data Segments      File - 'sample'
Seg # File Pos     Vir Size     Phys Size     Reloc Base
0x47  0x1000       0xe90        0x460         0x1880000
```

Each entry gives the segment number, file position, and physical size of a segment. The segment number is the starting address of the segment. The

file position is the offset from the start of the file to the contents of the segment. The physical size is the number of bytes the segment occupies in the program or core file. The filenames to the right of the display are the program and core filenames.

If you have executed the program, the command displays the following form for XENIX 386:

```
Text Segments      File - 'sample'
Seg #File Pos     Vir Size    Phys Size  Reloc Base
0x3f  0x400       0xb48       0xb48      0x0

Data Segments     File - 'sample'
Seg #File Pos     Vir Size    Phys Size  Reloc Base
0x47  0x1000      0x1880e90   0x460      0x1880000
```

where virtual size is the number of bytes the segment occupies in memory. This size is sometimes different from the size of the segment in the file and will often change as you execute the program. This is due to expansion of the stack or allocation of additional memory during program execution. The filenames to the right always name program files. The file position value is ignored.

## Giving Segment Numbers

If you give a segment number with the command, **adb** displays information only about that segment. For example, the following command displays a map for segment 63 only:

```
$m 63
```

The display has the following form for XENIX 286:

```
Segment #= 63.
Type= Text
File position= 160.
Virtual Size= 3712.
Physical Size= 2048.
```

The display has the following form for XENIX 386:

```
Segment # = 0x3f
Type = Text
File position = 0x3f
Virtual size = 0x400
Physical Size = 0x400
Reloc Base = 0x0
```

### 8.5.2 Changing the Memory Map

You can change the values of a memory map by using the **?m** and **/m** commands. These commands assign specified values to the corresponding map entries. The commands have the following form:

**?m** *segment-number file-position size*

and:

**/m** *segment-number file-position size*

where:

- *segment-number* gives the number of the segment map you wish to change,
- *file-position* gives the offset in the file to the beginning of the given address,
- *size* gives the segment size in bytes,
- **?m** assigns values to a text segment entry, and
- **/m** assigns values to a data segment entry.

For example, the following command changes the file position for segment 0x3f in the text map to 0x2000:

```
?m 0x3f 0x2000
```

The following command changes the file position for segment 0x47 in the data map to 0x0:

```
/m 0x47 0x0
```



### 8.5.3 Creating New Map Entries

You can create new segment maps and add them to your memory map by using the **?M** and **/M** commands. Unlike **?m** and **/m**, these commands

create a new map instead of changing an existing one. These commands have the following form:

> **?M** *segment-number file-position size*

and:

> **/M** *segment-number file-position size*

where:

- *segment-number* gives the number of the segment map you wish to create,
- *file-position* gives the offset in the file to the beginning of the given address, and
- *size* gives the segment size in bytes.

The **?M** command creates a text segment entry; **/M** creates a data segment entry. The segment number must be unique. You cannot create a new map entry that has the same number as an existing one.

The **?M** and **/M** commands are especially useful if you wish to access segments that are otherwise allocated to your program. For example, the following command creates a text segment entry for segment 0x47 whose size is 0x9c8 bytes:

```
?M 0x47 0x0 0x9c8
```

### 8.5.4 Validating Addresses

Whenever you use an address in a command, **adb** checks the address to make sure it is valid. To validate the address, **adb** uses the segment number, file position, and size values in each map entry. If an address is correct, **adb** carries out the command; otherwise, it displays an error message.

The first step **adb** takes when validating an address is to check the segment value to make sure it belongs to the appropriate map. Segments used with the **?** command must appear in the text segments map; segments used with the / command must appear in the data segments map. If the value does not belong to the map, **adb** displays a bad segment error.

The next step is to check the offset to see if it is in range. The offset must be within the following range:

```
0 <= offset <= segment-size
```

If it is not in this range, **adb** displays a bad address error.

If **adb** is currently accessing memory, the validating segment and offset are used to access a memory location and no other processing takes place. If **adb** is accessing files, it computes an effective file address like the following, then uses this effective address to read from the corresponding file:

```
effective-file-address = offset + file-position
```

## 8.6 Miscellaneous Features

The following sections explain a number of useful commands and features of **adb**.

### 8.6.1 Combining Commands on a Single Line

You can give more than one command on a line by separating the commands with a semicolon (;). The commands are performed one at a time, starting at the left. Changes to the current address and format are carried to the next command. If an error occurs, the remaining commands are ignored.

One typical combination is to place a **?** command after an **l** command. For example, the following command searches for and displays a string that begins with the characters *Th*:

```
?l 'Th'; ?s
```



### 8.6.2 Creating adb Scripts

You can direct **adb** to read commands from a text file instead of the keyboard by redirecting **adb**'s standard input file at invocation. To redirect

the standard input, use the standard redirection symbol < and supply a filename. For example, to read commands from the file *script*, type:

```
adb sample <script
```

The file you supply must contain valid **adb** commands. Such files are called script files, and can be used with any invocation of the debugger.

Reading commands from a script file is very convenient when you wish to use the same set of commands on several different object files. Scripts are typically used to display the contents of core files after a program error. For example, you can use a file containing the following commands to display most of the relevant information about a program error:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8xna
```

### 8.6.3 Setting Output Width

You can set the maximum width (in characters) of each line of output created by **adb** by using the **$w** command. The command has the following form:

*n*$w

where *n* is an integer number giving the width in characters of the display. You can give any width that is convenient for your given terminal or display device. The default width, when **adb** is first invoked, is 80 characters.

The command is typically used when redirecting output to a lineprinter or special terminal. For example, the following command sets the display width to 120 characters, a common maximum width for lineprinters:

```
120$w
```

### 8.6.4 Setting the Maximum Offset

The **adb** program normally displays memory and file addresses as the sum of a symbol and an offset. This helps associate the instructions and data you are viewing with a given function or variable. When first invoked, **adb** sets the maximum offset to 255. This means instructions or data that are no more than 255 bytes from the start of the function or variable are given symbolic addresses. Instructions or data beyond this point are given numeric addresses.

In many programs, the size of a function or variable is actually larger than 255 bytes. For this reason, **adb** lets you change the maximum offset to accommodate larger programs. You can change the maximum offset by using the **$s** command. The command has the following form where *n* is an integer giving the new offset:

*n*$s

For example, the following command increases the maximum possible offset to 4095:

```
4095$s
```

All instructions and data that are no more than 4095 bytes away are given symbolic addresses.

Note that you can disable all symbolic addressing by setting the maximum offset to zero. All addresses will be given numeric values instead.

### 8.6.5 Setting Default Input Format

You can set the default format for numbers used in commands with the **$d** (decimal), **$o** (octal), and **$x** (hexadecimal) commands. The default format tells **adb** how to interpret numbers that do not begin with *0* or *0x*, and how to display numbers when no specific format is given.

The commands are useful if you wish to work with a combination of decimal, octal, and hexadecimal numbers. For example, if you use the following combination, you can give addresses in hexadecimal without prepending each address with *0x*:

```
$x
```

Furthermore, **adb** displays all numbers in hexadecimal except those that are specifically requested to be in some other format.

When you first start **adb**, the default format is decimal. You can change this at any time and restore it as necessary using the **$d** command.

### 8.6.6 Using XENIX Commands

You can execute XENIX commands without leaving **adb** by using the **adb** escape command **!**. The escape command has the following form:

> **!** *command*

where *command* is the XENIX command you wish to execute. The command must have any required arguments. The **adb** passes this command to the system shell which executes it. When finished, the shell returns control to **adb**.

For example, to display the date, type:

```
! date
```

The system displays the date at your terminal and restores control to **adb**.

### 8.6.7 Computing Numbers and Displaying Text

You can perform arithmetic calculations while in **adb** by using the = command. This command directs **adb** to display the value of an expression in a given format.

You use the = command to convert numbers in one base to another, to double-check the arithmetic performed by a program, and to display complex addresses in easier form. For example, the following command displays the hexadecimal number ''0x2a'' as the decimal number 42:

```
0x2a=d
```

however, the following command displays it as the ASCII character asterisk (*):

```
0x2a=c
```

Expressions in a command may have any combination of symbols and operators. You can also compute the value of external symbols, by typing:

```
main+5=X
```

This is helpful if you wish to check the hexadecimal value of an external symbol address.

Note that the = command can also be used to display literal strings at your terminal. This is especially useful in an **adb** script where you may wish to display comments about the script as it performs its commands. For example, the following command spaces three lines:

```
=3n"C Stack Backtrace"
```

The system then displays the following message:

```
C Stack Backtrace
```

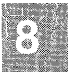
### 8.6.8 An Example: Directory and Inode Dumps

This example illustrates how to create **adb** scripts to display the contents of a directory file and the inode map of a XENIX file system. The directory file is assumed to be named *dir*, and contains a variety of files. The XENIX file system is assumed to be associated with the device file */dev/src*, and has the necessary permissions for you to read it.

To display a directory file, you must create an appropriate script, then start **adb** with the name of the directory, redirecting its input to the script.

First, you can create a script file named *script*. A directory file normally contains one or more entries. Each entry consists of an unsigned *inumber* and a 14-character filename. You can display this information by adding the following command to the script file:

```
0,-1?ut14cn
```



This command displays one entry for each line, separating the number and filename with a tab. The display continues to the end of the file. If you place the following command at the beginning of the script, **adb** will display the strings as headings for the columns of numbers:

```
="inumber"8t"Name"
```

Once you have the script file, type:

```
adb dir - <script
```

(The dash (-) is used to prevent **adb** from attempting to open a core file.) The **adb** program reads the commands from the script and displays the following:

```
inumber     name
652         .
82          ..
5971        cap.c
5323        cap
0           pp
```

To display the inode table of a file system, you must create a new script, then start **adb** with the filename of the device associated with the file system (such as the hard disk drive).

The structure of an inode table entry is defined in the file */usr/include/sys/ino.h.* Each inode entry includes:

- an unsigned short containing the mode and type of the file
- a short containing the number of links to the file
- an unsigned short containing the owner's user ID
- an unsigned short containing the owner's group ID
- a long containing the size of the file in bytes
- an array of 40 bytes containing the disk block addresses (only 39 of the 40 address bytes are in use: 13 addresses of 3 bytes each)
- a long giving the time the file was last accessed
- a long giving the time the file was last modified
- a long giving the time the file was created

The inode table starts at the address ''04000'' of the filesystem. This is the address of the beginning of block 2 of the file system. (For a discussion of the layout of a file system, see **filesystem**(F) of the *XENIX User's Reference.*)

You can display the first entry by typing:

```
04000,-1?onororon2un40b3Y2na
```

Several Newlines are inserted within the display to make it easier to read.

To use the script on the inode table of */dev/src*, type:

```
adb /dev/src - <script
```

(Again, the dash (-) is used to prevent an unwanted core file.) Each entry in the inode table display has the following form:

```
0.:2048.:  040755
           046      03     03
           640      0
           0121     06     0      0       0       0    0          0
           0        0      0      0       0       0    0          0
           0        0      0      0       0       0    0          0
           0        0      0      0       0       0    0          0
           0        0      0      0       0       0    0          0
           1986 Dec  4 13:55:49   1986 Nov 20 20:46:13   1986 Nov 20 20:46:13
```

## 8.7 Patching Binary Files

You can make corrections or changes to any file, including executable binary files, by using the **w** and **W** commands and invoking **adb** with the **-w** option. The following sections describe how to locate and change values in a file.

### 8.7.1 Locating Values in a File

You can locate specific values within a file by using the **l** and **L** commands. The commands have the following form:

[ *address* ] ?**l** *value*

[ *address* ] ?**L** *value*

where:

- *address* is the address at which to start the search,
- **l** command searches for 2 byte values,
- **L** command searches for 4 byte values, and
- *value* is the value (given as an expression) to be located.

The following command starts the search at the current address, and continues until the first match or the end of the file:

```
?l
```

If the value is found, the current address is set to that value's address.

### 8.7.2 Writing to a File

You can write to a file by using the **w** and **W** commands. The commands have the following form:

[ *address* ] ?**w** *value*

[ *address* ] ?**W** *value*

where:

- *address* is the address of the value you wish to change,
- **w** command writes 2 byte values,
- **W** writes 4 bytes, and
- *value* is the new value.

For example, the following commands change the word "This" to "The":

```
?l 'Th'
?W 'The'
```

Note that **W** is used to change all four characters.

### 8.7.3 Making Changes to Memory

You can also make changes to memory whenever a program has been executed. If you have used an **:r** command with a breakpoint to start program execution, subsequent **w** commands cause **adb** to write to the program in memory rather than the file. This is useful if you wish to make changes to a program's data as it runs, for example, to change the value of program flags or constants temporarily.

# Chapter 9

# ld: the XENIX Link Editor


9.1 Introduction 9-1

9.2 Using the Link Editor 9-1

9.3 Link Editor Options 9-1

9.4 The Executable Object File 9-4

9.5 Communal Variable Allocation 9-5

9.6 Pointer and Integer Sizes 9-6

9.7 Segment and Register Sizes 9-8

## 9.1 Introduction

The XENIX link editor **ld**(CP) is a companion tool to both the C compiler **cc** and the macro assembler **masm**(CP).

The **ld** tool creates executable files by combining object modules and resolving external references. The inputs to **ld** are relocatable object files produced by the C compiler or the macro assembler.

For a synopsis of the information presented in this chapter, we refer you to the **ld** page in the *XENIX Reference*.

## 9.2 Using the Link Editor

The link editor is invoked with the following form:

ld [*options* ] *filename1* *filename2* ...

where *options* are of the form described in the next section, and *filename* must be either an object file or an archive library containing object files.

Input object files and archive libraries of object files are linked together to form an executable file. If there are no unresolved references encountered, this file will then be made executable.

Object files have the form *name*.o throughout the examples in this chapter. The names of actual input object files need not follow this convention.

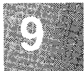
If you merely want to link the object files *file1.o* and *file2.o*, then the following command is sufficient:

```
ld file1.o file2.o
```

No directives to **ld** are necessary. If no errors are encountered during the link edit, the output is left on the default file *a.out*.

## 9.3 Link Editor Options

Input object files are linked in the order in which they are encountered. Options may be interspersed with filenames on the command line. The ordering of options is not significant.



Every option for **ld** must be preceded by a dash (-) on the **ld** command line. Any options that carries an argument is separated from that

argument by white space (blanks or tabs). The following is a summary of all the available options:

**-A** *num* — Creates a stand-alone program whose expected load address (in hexadecimal) is *num*. This option sets the absolute flag in the header of the *a.out* file. Such program files can only be executed as standalone programs.

**-B** *num* — Sets the text selector bias to the specific hexadecimal number.

**-c** *num* — Alters the default target CPU in the x.out header. num can be 0, 1, 2, or 3, indicating 8086, 80186, 80286 and 80386 processors, respectively. The default on 8086/80286 systems is 0. The default on 80386 systems is 3. Note that this option only alters the default. If object modules containing code for a higher-numbered processor are linked, then this will take precedence over the default.

**-C** — Ignores case when matching symbols. Normally, the link editor is case-sensitive.

**-D** *num* — Sets the data to the specific hexadecimal number.

**-F** *num* — Sets the size of the program stack to *num* bytes where *num* is hexadecimal. In programs configured with the **-M0**, **-M1**, or **-M2** option, this option changes the default stack size of 1000 bytes (hexadecimal) to *num* bytes (hexadecimal). In programs configured with the **-M3** option, the size of the stack is automatically controlled by the 80386; the **-F** option is not needed in this case. The **-F** option is incompatible with the **-A** option.

**-g** — Retains symbolic information for use with the **sdb**(CP) debugging program.

**-i** — Creates separate instruction and data spaces for small model programs. When the output

file is executed, the program text and data areas are allocated separate physical segments. The text portion will be read-only and shared by all users executing the file.

**-m** *mapfile* — Instructs the link editor to produce a mapfile that contains a description of all the segments in the executable file as well as listings of all public symbols and their values (sorted by both name and value).

**-M***x* — Informs the link editor of the nature of the memory model. The model *x* may be **s** (small), **m** (middle), **l** (large), **h** (huge), or **e** (mixed). The arguments **s**, **m**, and **l** are mutually exclusive.

**-N** *pagesize* — Forces the alignment of each segment to *pagesize* (which should be a multiple of 512) boundaries within the linker output file. The default is 1024 for 80386 programs. 8086/80186/80286 programs do not normally have page-aligned x.out files and the default for these is 0.

**-n** *num* — Instructs the link editor to truncate all symbols to a length equal to the specified *num*.

**-o** *name* — Produces an output object file named *name*. Overrides the default object file name *a.out*.

**-P** — Do not pack segments. Normally, the link editor attempts to pack all logical segments that do not have a group association into the same physical segment. This switch disables packing.

**-r** — Produces a relocatable object module as output.

**-R***x* *num* — Used in conjunction with the **-M3s** option to relocate a data segment specified by the *num* argument; added to the final target value of data fixups. **-Rt** is used to relocate text segments. The default for both data and text segments is 0. (This option applies only to the 80386.)

**-s** relocating
: Instructs the link editor to strip the line number entries and the symbol table information from the output object file.

**-S** *num*
: Sets the maximum number of segments allowed to *num*, which must be ≤ 1024. The default maximum is 128.

**-u** *symname*
: Enters *symname* as an undefined link editor symbol in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

**-v** *num*
: Takes the specified *number* as a decimal version number identifying the **a.out** that is produced. The version stamp is 2, 3, or 5 for the XENIX version and is stored in the system header.

## 9.4 The Executable Object File

Object files are produced both by the assembler (typically as a result of invoking the compiler) and by **ld**. **ld** accepts relocatable object files as input and produces an output object file.

Files produced from the compiler/assembler always contain three segments, called _TEXT, _DATA, and _BSS . The _TEXT segment contains the instruction text (e.g. executable instructions), the _DATA segment contains initialized data variables, and the _BSS (blank static storage) segment contains uninitialized data variables. The following program fragment will serve to illustrate:

```
int i = 100;    /* initialized variable */
char abc[200];    /* uninitialized variable */

main()
{
      abc[i] = 0;    /* assignment */
}
```

Compiled code from the assignment would be stored in _TEXT . The variable **i** would be located in _DATA , and the uninitialized string of characters **abc** would be located in the _BSS segment.

There is one exception to this rule: both initialized and uninitialized *statics* are placed in the _DATA segment.

## 9.5 Communal Variable Allocation

A communal variable is an uninitialized global variable. The link editor follows a number of rules in allocation of communal variables. They are as follows:

- If there are defined multiple communal variables of the same name, the link editor chooses the length of the largest definition and allocates that amount of space in the C_COMMON segment.
- If there is a definition of the variable that is initialized (a public definition), it takes precedence over all communal definitions and the link editor allocates the length specified by the PUBDEF in the _DATA segment.
- If there is more than one public definition, the link editor generates an error message saying that the symbol is multiply defined.

The following example illustrates these rules. Suppose you link the following three modules, containing these global declarations:

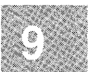
```
A:   char headr[512];
B:   char headr[128];
C:   char headr[256];
```

The link editor recognizes all three object modules (*A*,*B*,*C*) as containing declarations for *headr*, an uninitialized array. Then **ld** chooses the definition in module A as the largest of the three and allocates 512 bytes for *headr* in the C_COMMON segment.

Now suppose that the declarations were as follows:

```
A:   char headr[512];
B:   char headr[128] = "adc";
C:   char headr[256];
```

Module *B*'s array has been initialized and, according to the rules followed by **ld**, it takes precedence over all other declarations. 128 bytes are allocated for *headr* in the segment _DATA.



Note that in this case, any subsequent addressing beyond *headr*[127] will have unpredictable results.

The simplest way to avoid these dangers is to put all global declarations in a single header file that is included in all modules that reference them.

## 9.6 Pointer and Integer Sizes

The following tables define the bit sizes of text and data pointers in each program memory model enabled by the **-M0**, **-M1**, or **-M2** option.

**Table 9.1**

**8086/80286 Memory-Model Text and Data Pointers**

| Model | Data Pointer | Text Pointer | Integer |
|---|---|---|---|
| Small | 16 | 16 | 16 |
| Medium | 16 | 32 | 16 |
| Large | 32 | 32 | 16 |
| Huge | 32 | 32 | 16 |

The following table defines the bit sizes of text and data pointers in each program memory model enabled by the **-M3** option.

**Table 9.2**

**80386 Memory-Model Text and Data Pointers**

| Model | Data Pointer | Text Pointer | Integer |
|---|---|---|---|
| Pure-Text Small | 32 | 32 | 32 |

The following table lists the default text- and data-segment names, and the default module name for each object file created by the **-M0**, **-M1**, or **-M2** option.

**Table 9.3**

**8086/80286 Memory-Model Defaults**

| Model | Text | Data | Module |
|---|---|---|---|
| Small | _TEXT | _DATA | *filename* |
| Medium | *module*_TEXT | _DATA | *filename* |
| Large | *module*_TEXT | _DATA | *filename* |
| Huge | *module*_TEXT | _DATA | *filename* |

The following table lists the default text- and data- segment names and the default module name for each object file created by the **-M3** option.

**Table 9.4**

**80386 Memory-Model Defaults**

| Model | Text | Data | Module |
|---|---|---|---|
| Pure-Text Small | _TEXT | _DATA | *filename* |

## 9.7 Segment and Register Sizes

The following table summarizes the structure of text and data segments for the four possible program memory models enabled by the **-M0**, **-M1**, or **-M2** option.

**Table 9.5**

**8086/80286 Memory-Models Summary**

| Model | Text | Data | Segment Registers |
|---|---|---|---|
| Small | 1† | 1† | CS=DS=SS |
| Medium | 1 per module | 1 | DS=SS |
| Large | 1 per module | 1 | DS=SS |
| Huge | 1 per module | 1 | DS=SS |

† In impure-text small-model programs, text and data occupy the same segment. In pure-text programs, they occupy different segments and the register **CS** != **DS**.

The following table summarizes the structure of text and data segments for the two possible program memory models enabled by the **-M3** option.

**Table 9.6**

**80386 Memory-Model Summary**

| Model | Text | Data | Segment Registers |
|---|---|---|---|
| Pure-Text Small | 1 per module | 1 | CS!=DS,DS=ES=SS |

# Chapter 10

# m4: A Macro Processor


10.1 Introduction 10-1

10.2 Invoking m4 10-2

10.3 Defining Macros 10-2

10.4 Quoting 10-3

10.5 Using Arguments 10-6

10.6 Using Built-in Arithmetic Values 10-7

10.7 Manipulating Files 10-8

10.8 Using System Commands 10-9

10.9 Using Conditionals 10-9

10.10 Manipulating Strings 10-10

10.11 Printing 10-11

## 10.1 Introduction

The **m4** macro processor defines and processes specially defined strings of characters called macros. You can use the **m4** macro processor to enhance your programming language by defining a set of macros to be processed by **m4** and then using these macros in your programs. You can supplement your programming language with these macros to make your program more structured, readable, or appropriate for a particular application.

The major function of the **m4** macro processor is to replace one string of text with another as is done by the **#define** statement in C or the **define** construct in the **ratfor**(CP) command. Besides the straightforward replacement of one string of text with another, **m4** also provides:

- macros with arguments
- conditional macro expansions
- arithmetic expressions
- file-manipulation facilities
- string-processing functions

The basic operation of **m4** is to copy its input to its output. As the input is read, each alphanumeric string (that is, string of letters and digits) is checked. If the string is the name of a macro, the name of the macro is replaced by its defining text. The resulting string is reread by **m4**. Macros can also be called with arguments, in which case the arguments are collected and substituted in the right places in the defining text before **m4** rescans the text.

The **m4** macro processor provides a collection of about twenty built-in macros. In addition, the user can define new macros. This chapter describes some of the most commonly used built-in macros and explains how you can define your own macros. Built-in and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process. For more information about the built-in macros, see **m4**(CP) in the *XENIX Programmer's Reference*.

## 10.2 Invoking m4

To invoke **m4**, use a command of the form:

**m4** [*filenames*]

Filename arguments are processed in order. If there are no arguments, or if an argument is a dash (-), then the standard input is read. The processed text is written to the standard output, and can be redirected as shown by the following command:

**m4** *file1 file2* - >*outputfile*

Note the use of the dash in the above example to indicate processing of the standard input after *file1* and *file2* have been processed by **m4**.

## 10.3 Defining Macros

The primary built-in function of **m4** is **define**, which is used to define new macros. The following statement defines the *name* string as *stuff*:

**define**(*name*, *stuff*)

All subsequent occurrences of *name* will be replaced by *stuff*. *Name* must be alphanumeric and must begin with a letter. (The underscore (_) counts as a letter.) The term *stuff* means any text, including text that contains balanced parentheses; it may stretch over multiple lines. The following example defines N to be 100 and uses this symbolic constant in a later **if** statement:

```
define(N, 100)
  .
  .
  .
if (i>N)
```

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by a left parenthesis it is assumed to have no arguments. This is the situation for N, since it is actually a macro with no arguments. Thus, when it is used, no parentheses are needed following its name.

You should also notice that a macro name is only recognized as such if it appears surrounded by nonalphanumerics, as shown in the following statements:

```
define(N, 100)
  ...
if (NNN > 100)
```

The NNN variable is absolutely unrelated to the defined macro N, even though it contains three N's.

Macro names or arguments can also be defined in terms of other names or arguments. The following statements define M and N each to be 100:

```
define(N, 100)
define(M, N)
```

In **m4**, if M is defined as N or as 100, M is 100. Therefore, even if N subsequently changes, M does not.

This behavior arises because **m4** expands macro names into their defining text as soon as it possibly can. This means that when the N string is seen, as the arguments of **define** are being collected, it is immediately replaced by 100. Therefore, you could have used the following statement in the first place:

```
define(M, 100)
```

If this isn't what you want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions as shown in the statements below:

```
define(M, N)
define(N, 100)
```

Now M is defined as the string N, so when you ask for M later, you will always get the value of N at that time (because the M will be replaced by N, which, in turn, will be replaced by 100).

## 10.4 Quoting

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by a grave accent and a single quotation mark ('and') is not expanded immediately, but has the marks stripped off. The following statements remove the punctuation

marks from the N as the argument is being collected, but they have served their purpose, and M is defined as the N string, not as 100:

```
define(N, 100)
define(M, `N')
```

The general rule is that **m4** always strips off one level of single quotation marks whenever it evaluates something. This is true even outside of macros. If you want the word *define* to appear in the output, you have to quote it in the input, as shown below:

```
`define' = 1;
```

As another similar instance, consider redefining N with the following statements:

```
define(N, 100)
 ...
define(N, 200)
```

The N in the second definition is evaluated as soon as it is seen; that is, it is replaced by 100, which is the same as the following statement:

```
define(100, 200)
```

This statement is ignored by **m4**, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine N, you must delay the evaluation by quoting as shown below:

```
define(N, 100)
 ...
define(`N', 200)
```

In **m4**, it is often wise to quote the first argument of a #define statement.

If the acute and grave marks ( ´ and ` ) are not convenient for some reason, you can change the marks with the built-in function **changequote**. For example, the following statement defines the new quotation marks to be the left and right brackets:

```
changequote([, ])
```

You can restore the original characters by typing:

```
changequote
```

There are two additional built-in functions that are related to **define**. The built-in function **undefine** removes the definition of some macro or built-in function. The following statement removes the definition of N:

```
undefine('N')
```

Built-in functions can be removed with **undefine**, as in the following statement:

```
undefine('define')
```

---

*Note*

Once you remove a built-in function, you cannot get it back.

---

The built-in function **ifdef** determines whether a macro is currently defined. For instance, suppose that either *xenix* or *unix* is defined according to a particular implementation of a program. To perform operations according to the system you are using, you could use the following statements:

```
ifdef('xenix', 'define(system,1)' )
ifdef('unix', 'define(system,2)' )
```

Don't forget the punctuation marks in the previous example.

The **ifdef** function actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as shown in the following statement:

```
ifdef('xenix', on XENIX, not on XENIX)
```

## 10.5 Using Arguments

So far you have learned the simplest form of macro-processing, that is, replacing one string with another (fixed) string. User-defined macros can also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**), any occurrence of *$n* will be replaced by the *n*th argument when the macro is actually used. Thus, the **bump** macro, shown here, generates code to increment its argument by 1:

```
define(bump, $1 = $1 + 1)
```

Therefore, calling the **bump** macro as shown below will cause *x* to become *x*+1:

```
bump(x)
```

A macro can have as many arguments as you want, but only the first nine, **$1** to **$9**, are accessible. (The macro name itself is **$0**.) Arguments not supplied are replaced by null strings, so you can define a macro, **cat**, which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Therefore, the following statement equals the expression *xyz*:

```
cat(x, y, z)
```

The arguments **$4** through **$9** are null, since no corresponding arguments were provided.

Leading unquoted spaces, TABs, or Newlines that occur during argument collection are discarded. All other white space is retained. Therefore, the following statement defines *a* to be *b   c*:

```
define(a, b  c)
```

Arguments are separated by commas, but parentheses are counted properly, so a comma protected by parentheses does not terminate an argument. Therefore, in the statement below there are only two arguments; the second is literally *(b,c)*:

```
define(a, (b,c))
```

Of course, a bare comma or parenthesis can be inserted by quoting it.

## 10.6 Using Built-in Arithmetic Values

The **m4** processor provides two built-in functions for doing arithmetic on integers. The simplest is **incr**, which increments its numeric argument by 1. Thus, to handle the common programming situation where you want a variable to be defined as one more than N, use the following statements:

```
define(N, 100)
define(N1, `incr(N)')
```

Then *N1* is defined as one more than the current value of *N*.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the following operators (in decreasing order of precedence):

<table>
<tr><th>Precedence</th><th>Operator</th><th>Name</th></tr>
<tr><td>1</td><td>+</td><td>Arithmetic Addition</td></tr>
<tr><td></td><td>-</td><td>Arithmetic Negation</td></tr>
<tr><td>2</td><td>**</td><td>Exponential</td></tr>
<tr><td></td><td>^</td><td>Exponential</td></tr>
<tr><td>3</td><td>*</td><td>Multiplication</td></tr>
<tr><td></td><td>/</td><td>Division</td></tr>
<tr><td></td><td>%</td><td>Remainder</td></tr>
<tr><td>4</td><td>+</td><td>Addition</td></tr>
<tr><td></td><td>-</td><td>Subtraction</td></tr>
<tr><td>5</td><td>= =</td><td>Equal</td></tr>
<tr><td></td><td>!=</td><td>Not Equal</td></tr>
<tr><td></td><td><</td><td>Less than</td></tr>
<tr><td></td><td><=</td><td>Less than or Equal to</td></tr>
<tr><td></td><td>></td><td>Greater than</td></tr>
<tr><td></td><td>>=</td><td>Greater than or Equal to</td></tr>
<tr><td>6</td><td>!</td><td>Logical NOT</td></tr>
<tr><td>7</td><td>&</td><td>Logical AND</td></tr>
<tr><td></td><td>&&</td><td>Logical AND</td></tr>
<tr><td>8</td><td>|</td><td>Logical OR</td></tr>
<tr><td></td><td>||</td><td>Logical OR</td></tr>
</table>

You can use parentheses to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** depends on the implementation.

For example, suppose you want M to be **2**N+1**, you can use the following statements:

```
define(N, 3)
define(M, `eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple (for example, just a number); this usually gives the result you want and is good programming form.

## 10.7 Manipulating Files

You can include a new file in the input at any time by using the built-in function **include**. The following statement inserts the contents of *filename* in place of the **include** command:

**include**(*filename*)

The contents of the file are often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file.

It is a fatal error if **include** cannot access the file named. To get some control over this situation, you can use the alternate form **sinclude**. **Sinclude** (silent include) says nothing and continues if it can't access the file.

You can also divert the output of **m4** to temporary files during processing and output the collected material upon command. The **m4** macro processor maintains nine of these diversions, numbered 1 through 9. The following statement puts all subsequent output at the end of a temporary file referred to as *n*:

**divert**(*n*)

Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion using the following statement:

```
undivert
```

This brings back all diversions in numeric order, and **undivert** with arguments brings back selected diversions in a given order. The act of undiverting discards the diverted text, as does diverting into a diversion whose number is not between 0 and 9, inclusive. The value of **undivert** is not the diverted text. Furthermore, the diverted material is not rescanned for macros.

The built-in **divnum** function returns the number of the currently active diversion. This is zero during normal processing.

## 10.8 Using System Commands

You can run any program in the local operating system with the built-in **syscmd** function. For example, the following statement runs the **date** command:

```
syscmd(date)
```

Normally, **syscmd** would be used to create a file for a subsequent **include** command.

To make unique filenames easily, the built-in function **maketemp** is provided with specifications identical to the system function **mktemp**. A string of ''XXXXX'' in the argument is replaced by the process ID of the current process.

## 10.9 Using Conditionals

There is a built-in conditional called **ifelse** that enables you to perform arbitrary conditional testing. In the simplest form, the following statement compares the two strings *a* and *b*:

```
ifelse(a, b, c, d)
```

If *a* and *b* are identical, **ifelse** returns the string *c*; otherwise, it returns *d*. Therefore, you might define a macro called **compare**, which compares two strings and returns ''yes'' if they are the same or ''no'' if they are different, as shown:

```
define(compare, `ifelse($1, $2, yes, no)')
```

Note the quotation marks, which prevent the premature evaluation of **ifelse**. If the fourth argument is missing, it is treated as empty. **Ifelse** can actually have any number of arguments, and thus provides a limited form of multiple-decision capability. For example, the following statement

compares the *a* and *b* strings. If they match, the result is *c*, and if *d* is the same as *e*, the result is *f*. Otherwise, the result is *g*.

**ifelse**(*a*, *b*, *c*, *d*, *e*, *f*, *g*)

If the final argument is omitted, the result is null, so the following statement evaluates to *c* if *a* matches *b*, and to null otherwise.

```
ifelse(a, b, c)
```

## 10.10 Manipulating Strings

The built-in **len** function returns the length of the string that makes up its argument. The following statement will return a value of 6:

```
len(abcdef)
```

All characters within the parentheses are counted, so the following statement will return a value of 5:

```
len((a,b))
```

The built-in **substr** function produces substrings of strings. The following statement returns the substring of *s* that starts at position *i* (origin zero) and is *n* characters long:

**substr**(*s*,*i*,*n*)

If *n* is omitted, the rest of the string is returned, so the following statement will return the string "ow is the time":

```
substr('now is the time', 1)
```

If *i* or *n* is out of range, various things result. For example, the following statement returns the index (position) in *s1* where the string *s2* occurs, or -1 if it doesn't occur:

```
index(s1,s2)
```

As with the **substr** function, the origin for strings is 0.

The built-in **translit** command performs character transliteration. The following command modifies *s* by replacing any character found in *f* with the corresponding character of *t*:

**translit**(*s*, *f*, *t*)

The following statement replaces the vowels with the corresponding digits:

```
translit(s, aeiou, 12345)
```

If *t* is shorter than *f*, characters that don't have an entry in *t* are deleted, as a limiting case. If *t* is not present at all, characters from *f* are deleted from *s*. Therefore, the following statement deletes vowels from "s":

```
translit(s, aeiou)
```

There is also a built-in function called **dnl** which deletes all characters that follow it up to and including the next Newline. It is useful for throwing away empty lines that otherwise tend to clutter up **m4** output. For example, if you use any of the following statements, the Newline character at the end of each line is not part of the definition, so it is copied into the output where it may not be wanted:

```
define(N, 100)
define(M, 200)
define(L, 300)
```

If you add **dnl** to each of these lines, the Newline characters will disappear.

The following statements illustrate another way to eliminate unwanted Newline characters:

```
divert(-1)
     define(...)
     ...
divert
```

## 10.11 Printing

The built-in command **errprint** writes its arguments out on the standard error file. Thus, you can use the statement below to print a "fatal error" message:

```
errprint('fatal error')
```

The **dumpdef** function is a debugging aid that dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise, you get the ones you name as arguments. Don't forget the punctuation marks.

# Chapter 11

# sdb: The Symbolic Debugger


11.1 Introduction 11-1

11.2 Using sdb 11-1
- 11.2.1 Starting sdb with a Program File 11-2
- 11.2.2 Starting sdb with a Core Image 11-2
- 11.2.3 Printing a Stack Trace 11-4
- 11.2.4 Examining Variables 11-4
- 11.2.5 Specifying Variable Formats 11-5
- 11.2.6 Leaving sdb 11-6

11.3 Displaying and Manipulating Source Files 11-6
- 11.3.1 Displaying the Source File 11-6
- 11.3.2 Setting the Current File or Function 11-7
- 11.3.3 Setting the Current Line 11-7
- 11.3.4 Searching for Regular Expressions 11-8

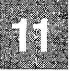
11.4 Controlling Program Execution 11-9
- 11.4.1 Setting and Deleting Breakpoints 11-9
- 11.4.2 Single Stepping through a Program 11-10
- 11.4.3 Running the Program 11-11
- 11.4.4 Calling Functions and Procedures 11-12

11.5 Debugging Machine-Language Programs 11-13
- 11.5.1 Displaying Machine-Language Statements 11-13
- 11.5.2 Manipulating Registers 11-13

11.6 Using XENIX Shell Commands 11-13

11.7 A Sample sdb Session 11-14

## 11.1 Introduction



This chapter describes a symbolic debugger **sdb(CP)**, as implemented for C and assembly language programs on the XENIX operating system.

You can use the **sdb** program both for examining core images of aborted programs and for providing an environment in which execution of a program can be monitored and controlled. The **sdb** program allows you to interact with a debugged program at the source language level. It carefully controls the execution of a program while letting you examine and modify the program's data and text areas.

This chapter explains how to use **sdb**. In particular, it explains how to:

- Start and stop the debugger
- Display and manipulate instructions and data in source files
- Control and monitor program execution
- Debug machine language programs

A tutorial provided at the end of this chapter shows you how to work your way through your program using **sdb**.

## 11.2 Using sdb

The **sdb** program provides a powerful set of commands to let you examine, debug, and repair source files. To realize the full power of **sdb**, you need to compile the source program with the **-Zi** option using a command of the following form:

```
cc -Zi prgm.c
```

This causes the compiler to generate additional information about the variables and statements of the compiled program. When the **-Zi** option has been specified, **sdb** can be used to obtain a trace of the called functions at the time of the abort and interactively display the values of variables.

There are two basic ways to use **sdb**: by running your program file under control of **sdb**, or by using **sdb** to examine the core image file left by a program that failed. The first way lets you see what the program is doing up to the point at which it fails (or skip around the failure point and proceed with the run). The second method lets you check the status at the moment of failure, which may or may not disclose the reason the program failed. Both of these methods are discussed in the following sections.

### 11.2.1 Starting sdb with a Program File

You can debug any executable C or assembly language program file by typing a command line of the following form:

**sdb** [*filename* ]

where *filename* is the name of the program file to be debugged. The **sdb** program opens the file and prepares its text (instructions) and data for subsequent debugging. For example, the following command prepares the program named **sample** for examination and execution:

```
sdb sample
```

Once started, **sdb** prompts with an asterisk (*) and waits for you to type commands. If you have given the name of a file that does not exist or is in the wrong format, **sdb** will display an error message first, then wait for commands.

You can also start **sdb** without a filename. In this case, **sdb** searches for the default file *a.out* in your current working directory and prepares it for debugging. Thus, the following command:

```
sdb
```

is the same as typing:

```
sdb a.out
```

The **sdb** program displays an error message and waits for a command if the *a.out* file does not exist.

### 11.2.2 Starting sdb with a Core Image

The **sdb** program lets you examine the core image files of programs that caused fatal system errors. When debugging a core image from an aborted program, **sdb** reports which line in the source program caused the error, and allows all variables to be accessed symbolically and displayed in the correct format.

To illustrate the process of debugging a core image file, we will use a hypothetical file called *prgm.c* and show a typical set of commands and responses for this process. First, you must compile and execute the program by typing the following:

```
cc -Zi prgm.c -o prgm
```

To execute the program, type:

```
prgm
```

For this example, suppose that an error occurred while executing the program, causing a core dump. The output resulting from this error is:

```
Bus error - core dumped
```

Now invoke the **sdb** program and examine the core dump to determine the causes of the error using the following command:

```
sdb prgm
```

A possible response from the **sdb** program is:

```
main:25:   x[i]=0;
*
```

This output means that the bus error occurred in function **main** at line 25 and outputs the source text of the offending line. Note that line numbers are always relative to the beginning of the file, not the beginning of the program. The **sdb** program then prompts the user with an *, which signifies waiting for a command.

It is useful to know that **sdb** uses a notion of current function and current line. In this example, they are initially set to **main** and 25, respectively.

In the example shown in this section, **sdb** was called with one argument, **prgm**. In general, it takes two arguments on the command line:

1. The first is the name of the executable file that is to be debugged; it defaults to *a.out* when not specified.

2. The second is the name of the core file, defaulting to *core*. You can prevent **sdb** from opening this file by using the hyphen (-) in place of the core filename.

In the example, the second argument defaulted to the correct value, so only the first was specified.

If the error occurred in a function that was not compiled with the **-Zi** option, **sdb** prints the function name and the address at which the error occurred. The current line and function are set to the first executable line in **main**. If **main** was not compiled with the **-Zi** option, **sdb** will print an error message, but you can continue to debug for those routines that were compiled with the **-Zi** option.

A sample **sdb** session with more examples is shown at the end of this chapter.

### 11.2.3 Printing a Stack Trace

When you are debugging a program, it is often useful to obtain a listing of the function calls that led to the error. You can obtain this listing by typing the **t** command in response to the **sdb** prompt:

```
*t
```

Possible output from the **t** command might be:

```
SampMod (), line 1758
decode (520022), line 1829
main (2,519960), line 2057
_start()
```

This indicates that the program was stopped within the function **SampMod** at line 1758. The **decode** function was called with argument 520022 at line 1829. The **main** function was called with the arguments 2 and 519960 at line 2057. The **main** function is usually called by a startup routine with two arguments referred to as *argc* and *argv*. Note that *argv* is a pointer and that the values of both arguments are printed in the current radix, which is decimal.

### 11.2.4 Examining Variables

The **sdb** program can be used to display variables in the stopped program. Variables are displayed by typing their names and pressing Return. To display the value of variable *errflag*, type the following command:

```
errflag
```

To display the variable as a two-byte decimal value, type its name followed by a slash:

```
errflag/
```

Unless otherwise specified, variables are assumed to be either local to, or accessible from, the current function.

### 11.2.5 Specifying Variable Formats

The **sdb** program normally displays the variable in a format determined by its type as declared in the source program. To request a different format, place a specifier after the slash. The specifier consists of a length option followed by the format. The length specifiers are as follows:

| Specifier | Length |
|---|---|
| b | One byte |
| h | Two bytes (half word) |
| l | Four bytes (long word) |

The length specifiers are effective only with the formats **d**, **o**, **x**, and **u**. If no length is specified, the word length of the host machine is used. A number can be used with the **s** or **a** format to control the number of characters printed. The **s** and **a** formats normally print characters until either a null is reached or 128 characters have been printed. The number specifies exactly how many characters should be printed. The available format specifiers are described as follows:

| Specifier | Format |
|---|---|
| c | character |
| d | decimal |
| u | decimal unsigned |
| o | octal |
| x | hexadecimal |
| f | 32-bit single-precision floating point |
| g | 64-bit double-precision floating point |
| s | assumes variable is a string pointer and prints characters starting at the address pointed to by the variable until a null is reached |
| a | prints characters starting at the variable's address until a null is reached |
| i | interprets as a machine-language instruction |

For example, to display the hexadecimal value of the variable *flag*, type

```
*flag/x
```

The **sdb** program also knows about structures, arrays, and pointers so that all of the following commands work:

```
*array[2][3]/
*sym.id/
*psym->usage/
*xsym[20].p->usage/
```

### 11.2.6 Leaving sdb

To exit **sdb** and return to the system shell, use the **q** or **quit** command.

## 11.3 Displaying and Manipulating Source Files

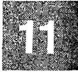
The **sdb** program makes it easy to debug a program without constant reference to a current source listing. Features are provided that perform context searches within the source files of the program being debugged and that display selected portions of the source files. The commands are similar to those of the line editor **ed**. Like **ed**, **sdb** uses a notion of current file and line within the current file. The **sdb** program also knows how the lines of a file are partitioned into functions, so it also uses a notion of current function. As noted in other parts of this document, the current function is used by a number of **sdb** commands.

### 11.3.1 Displaying the Source File

Four commands exist for displaying lines in the source file. They are useful for examining the source program and for determining the context of the current line. The commands are

| Command | Description |
|---|---|
| p | Displays the current line. |
| w | Displays a window of ten lines around the current line. |
| z | Displays ten lines starting at the current line. Advances the current line by ten. |
| Ctrl-D | Scrolls. Displays the next ten lines and advances the current line by ten. This command is used to display cleanly long segments of the program. |

When a line from a file is displayed, it is preceded by its line number. This not only gives an indication of its relative position in the file, but is also used as input by some **sdb** commands.




### 11.3.2 Setting the Current File or Function

You can use the **e** command to change the current source file. For example, you can change the current file to *file.c* by typing:

```
*e file.c
```

In the above example, the current line is also set to the first line in the specified file.

You can also specify that you want a file containing a certain function to become the current file. For example, to change the current file to the file containing *function*, type:

```
*e function
```

This command also causes the first line of the function specified to become the current line.

To display the current function and file, use the **e** command with no arguments.

### 11.3.3 Setting the Current Line

There are several ways to change the current line in the source file. It may be helpful for you to refer to the *XENIX User's Guide* if you are unfamiliar with the concept of current line.

The + and - commands can be used to move the current line forward or backward by a specified number of lines. Typing a new line advances the current line by one, and typing a number causes that line to become the current line in the file. For example, to advance the current line by 15 and then print ten lines, use the following command line:

```
*15+z
```

When you use the **z** or Ctrl-D command to display data, it also sets the current line to the last line displayed.

### 11.3.4 Searching for Regular Expressions

There are two commands for searching for instances of regular expressions in source files. To search forward through a file for a line containing a string that matches a regular expression, type:

```
*/regular expression
```

To search backward through a file for a line containing a string that matches a regular expression, type:

```
*?regular expression
```

To show all variables beginning with *x*, type:

```
*/x*/
```

To show all two letter variables in function *sub*, type:

```
*/sub:??/
```

To show all the variables in the current function, type:

```
*/
```

To display location 1024 in decimal representation, type:

```
*1024/
```

To display the address of the variable *foo*, type:

```
*foo=
```

To redisplay the last variable typed, type:

```
*./
```

## 11.4 Controlling Program Execution



One very useful feature of **sdb** is breakpoint debugging. After you have entered **sdb**, breakpoints can be set at certain lines in the source program. The program is then started with an **sdb** command. Execution of the program proceeds as normal until it is about to execute one of the lines where a breakpoint has been set. The program stops and **sdb** reports the breakpoint where the program stopped. Now, **sdb** commands can be used to display the trace of function calls and the values of variables. If the user is satisfied that the program is working correctly to this point, some breakpoints can be deleted and others set and program execution can be continued from the point where it stopped.

### 11.4.1 Setting and Deleting Breakpoints

You can set breakpoints at any line in any function if your program has been compiled correctly. You will use the **b** command to set breakpoints. To set a breakpoint at line 12 in the current file, type:

```
*12b
```

The line numbers are relative to the beginning of the file as printed by the source-file display commands: **p**, **w**, **z**, and Ctrl-D.

To set a breakpoint at line 12 of function **proc**, type:

```
*proc:12b
```

To set a breakpoint at the first line of function **proc**, type:

```
*proc:b
```

To set a breakpoint at the current line, type:

```
*b
```

You can delete breakpoints in the same way that you set them using the **d** command. To delete a breakpoint at line 12 in the current file, type:

```
*12d
```

To delete a breakpoint at line 12 of function **proc**, type:

```
*proc:12d
```

To delete a breakpoint at the first line of function **proc**, type:

```
*proc:b
```

To delete breakpoints interactively, type the **d** command with no arguments. The **sdb** program prints the location of each breakpoint and waits for a response from the user. If you respond with a **y** or **d**, the breakpoint is deleted.

To print a list of all the current breakpoints, use the **B** command. To delete all the current breakpoints, use the **D** command.

You can also use the breakpoint command to perform automatically a sequence of commands at a breakpoint and then have execution continue. For example, if you want both a stack back trace and the value of *x* to be displayed each time execution gets to line 12, type:

```
*12b t;x/
```

The **a** command also lets **sdb** perform debugging functions. If you want to set a breakpoint and announce the breakpoint by printing the top line of the stack trace, type:

```
*proc:a
```

If you want to set a breakpoint at a given line in a procedure and announce the breakpoint by printing the current line, type:

```
*proc:12a
```

When you are using the **a** command, execution continues after the function name or source line is printed.

### 11.4.2 Single Stepping through a Program

A useful alternative to setting breakpoints is single-stepping through a program. The **sdb** program can be requested to execute the next line of the program and then stop using the **s** command. The command has the following form:

[*count*]**s**

where *count* is the number of lines to execute in each step. This command is useful for slowly executing the program to examine its behavior in detail.

The **S** command is similar to the **s** command, but it steps through the subroutine calls, rather than execute only one line. It is often used when you are confident that the called function works correctly but you are interested in testing the calling routine.

Single-stepping is especially useful for testing new programs, so they can be verified on a statement-by-statement basis. If an attempt is made to single-step through a function that has not been compiled with the **-Zi** option, execution proceeds until a function that has been compiled this way is reached .

You can use the **i** command to run the program one machine-level instruction at a time while ignoring the signal that stopped the program. Its uses are similar to the **s** command. The **I** command is similar to the **i** command except that it steps through the call instructions, rather than execute only one line. These machine-level commands are particularly useful when you have not compiled your program with the necessary options.

For more information on the single-stepping commands, see **sdb**(CP) in the *XENIX Programmer's Reference.*

### 11.4.3 Running the Program

The **r** command is used to begin program execution. This command lets you restart the program as if it were invoked from the shell. For example, to run the current program with given arguments, use the following command form:

> ***r** *args*

If no arguments are specified, then the arguments from the last execution of the program within **sdb** are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as Interrupt or Quit occurs, or the program terminates. In all cases after an appropriate message is printed, control returns to the user.

To continue execution of a stopped program, use the **c** command. You can also use the **c** command to insert a temporary breakpoint during execution. For example, to place a temporary breakpoint at line 12 and resume execution of a stopped program, type:

```
*proc:12c
```

The temporary breakpoint is deleted when the **c** command finishes.

If you want the signal that stopped program execution to be passed back to the program, use the **C** command. This is useful for testing user-written signal handlers.

You can continue execution at a specified line with the **g** command. For example, you can continue program execution at line 17 of the current function with the following command:

```
*17 g
```

This is useful when you want to avoid a section of code that you already know is bad. You should not attempt to continue execution in a function different from that of the breakpoint.

### 11.4.4 Calling Functions and Procedures

It is possible to call any of the functions or procedures in the program from **sdb**. This feature is useful both for testing individual functions with different arguments and for calling a user-supplied function to display structured data. To execute a function or procedure, use a command of the following form:

```
*proc(arg1, arg2, ...)
```

This command will display the return value of the function using the default format for the type of function called. To call a function, display the value that it returns, and specify the format in which the value is to be displayed, type:

```
*proc(arg1, arg2, ...)/m
```

The value is displayed in decimal unless some other format is specified by **m**. Arguments to functions can be integer, character or string constants, or variables that are accessible from the current function.

If a function is called when the program is not stopped at a breakpoint (such as when a core image is being debugged), all variables are initialized before the function is started. This makes it impossible to use a function that formats data from a dump.

## 11.5 Debugging Machine-Language Programs



The **sdb** program has facilities for examining programs at the machine-language level. It is possible to print the machine-language statements associated with a line in the source and to place breakpoints at arbitrary addresses. The **sdb** program can also be used to display or modify the contents of the machine registers.

### 11.5.1 Displaying Machine-Language Statements

To display the machine-language statements associated with line 25 in function **main**, use the following command:

```
*main:25?
```

The **?** command is identical to the / command except that it displays from text space. The default format for printing text space is the **i** format, which interprets the machinelanguage instruction. To print the next ten instructions, press Ctrl-D.

### 11.5.2 Manipulating Registers

The **x** command displays the values of all the registers. For example, to display the value of all the registers, type:

```
*x
```

To display the value of a specific register (*reg*), type:

@*reg*

## 11.6 Using XENIX Shell Commands

The ! command (when used immediately after the * prompt) performs the same function as it does in **ed**(C). It lets you temporarily escape from **sdb** to execute a shell command. After the command executes, control is returned to the **sdb** program.

The **!** can also be used to change the values of variables when the program is stopped at a breakpoint. To set a specified variable to a specified value, use the following command:

**variable* ! *value*

The value can be a number, a character constant, or the name of another variable. If the variable is of type **float** or **double**, the value can also be a floating-point constant (specified according to the standard C language format).

## 11.7 A Sample sdb Session

This section provides a sample **sdb** session to show you how to to debug a program using **sdb**.

1. Before we can begin the **sdb** session, you must create a source file named *sample.c* with the contents as follows:

```
/*
 *    Program usage: Try number [number......]
 *
 *    Program echoes its numerical arguments.....
 */
#include    <stdio.h>

struct node {
      int           val;
      struct node *       next;
};

struct node First;

main(argc,argv)
int   argc;
char  **argv;
{
      register int i;

      First.val = atoi(argv[1]);

      for (i = 2; i < argc; i++) {
            attach( &First, atoi(argv[i]) );
      }
      traverse(&First);
}
```

(continued)



```
attach(leaf, num)
struct node *leaf;
int num;
{
        while (leaf->next != (struct node *) NULL) {
              leaf = leaf->next;
        }
        leaf->next = (struct node *) malloc( sizeof(struct node) );
        leaf = leaf->next;
        leaf->val = num;
        leaf->next = (struct node *) NULL;
        return(num);
}

traverse(leaf)
struct node *leaf;
{
        while (leaf != (struct node *) NULL) {
              printf("%d ",leaf->val);
            leaf = leaf->next;
        }
        putchar('\n');
}
```

2. You must compile the *sample* program using the **-Zi** option so that you can use the full capabilities of the **sdb** commands during this session. To compile the *sample.c* program, type:

   ```
   cc -Zi sample.c
   ```

3. To rename the resulting executable object file from *a.out* to *sample*, type:

   ```
   mv a.out sample
   ```

4. To begin the **sdb** session, type:

   ```
   sdb sample -
   ```

   Please note that since you do not want **sdb** to examine the core file associated with *sample*, a hyphen (-) has been specified here to prevent this. The **sdb** program displays the following message:

   ```
   Core file ignored
   ```

5. It is useful to set breakpoints anywhere in your program that you suspect to be the trouble spot. For example, set a breakpoint at the start of function **attach** by typing:

```
*attach:b
```

The **sdb** program displays the following message:

```
_attach: 30
```

This message means that the function **attach** starts at line 30.

6. Start executing the program by typing:

```
*r 10 20 30 40
```

The **sdb** program displays the following message:

```
sample
Breakpoint at
_attach: 30 struct node *leaf;
```

7. When debugging a program, it is often useful to obtain a listing of the function calls that led to the error. To do this, type:

```
*t
```

The **sdb** program displays the following message:

```
attach (25691632,20), line 30
main (5,25685724), line 24
_start()
```

8. It is useful to peruse the source program to determine the context of the current line. To do this, display the next ten lines of the source file using the following command:

```
*z
```

The **sdb** program displays:

```
30:struct node *leaf;
31:int num;
32:{
33:     register struct node *p;
34:     struct node *q;
35:
36:     while (leaf->next != (struct node *) NULL) {
37:             leaf = leaf->next;
38:     }
39:     leaf->next = (struct node *) malloc( sizeof(struct node));
```

9. It is often useful to step through a program, line by line, while it is executing to examine its behavior in detail. To do this, type:

```
*s
```

The **sdb** program displays:

```
36:   while (leaf->next != (struct node *) NULL) {
```

Type the single step-command again:

```
*s
```

The **sdb** program displays:

```
39:leaf->next = (struct node *) malloc(sizeof(struct node));
```

Continue using this command until you have examined all the lines of the program that are of interest to you:

```
*s
```

The **sdb** program displays the following:

```
40:   leaf = leaf->next;
```

10. An important alternative to the **s** command is the **S** command. This command steps through the program a line at a time but when it encounters a function call, it executes the function. To do this, type:

```
*S
```

The **sdb** program displays the following:

```
41:   leaf->val = num;
```

11. You can single step through the program for a specified number of lines. To step through the next 10 lines of the program, type:

```
*10s
```

The **sdb** program displays the following:

```
41:    leaf->val = num;
42:    leaf->next = (struct node *) NULL;
43:    return(num);
44:}
25:    }
24:         attach( &First, atoi(argv[i]) );
30:struct node *leaf;
36:    while (leaf->next != (struct node *) NULL) {
37:         leaf = leaf->next;
38:    }
```

12. The **sdb** program can display the current value of variables used in the program. For example, you can display the value of the variable *leaf->val* by typing:

```
*leaf->val
```

The **sdb** program displays the following:

```
20
```

13. You can use the **x** command to display the contents of all the registers in your program. To do this, type:

```
*x
```

The **sdb** program displays register information in the following format:

```
eax=01881004 ebx=00000000 ecs=0000001E edx=00000000
esp=0187EE74 ebp=0187EE88 esi=00000003 edo=0187EF74
ds=0047 es=0047 fs=003F gs=003F ss=0047 NV UP EI PL NZ NA PO NC
38:    }
003F:0000011C 55  E9E5FFFFFFFF    jmp start0+86 (00000106)
```

14. You can use **sdb** to change the values of variables when your program is stopped at a breakpoint. To set the value of *leaf->val* to 99, type:

```
*leaf->val!99
```

15. Display the new value of *leaf->val*. To do this, type:

```
*leaf->val
```

The **sdb** program displays the following:

```
99
```

16. Continue executing the program until the next breakpoint is reached. To do this, type:

```
*c
```

The **sdb** program displays the following:

```
Breakpoint at
_attach: 30 struct node *leaf;
```

17. When you have finished debugging your program, you will want to delete all the breakpoints in it. To do this, type:

```
*D
```

The **sdb** program displays the following:

```
All breakpoints deleted
```

18. Continue executing your program until it has finished running. To do this, type:

```
*c
```

When program execution is complete, the **sdb** program displays the following:

```
10 99 30 40

Process terminated normally (5)
```

When you are ready to leave **sdb**, type the following command:

```
*q
```

This officially ends your **sdb** session.

# Appendix A

# XENIX System Calls


A.1 Introduction A-1

A.2 Executable File Format A-1

A.3 Revised System Calls A-2

A.4 Version 7 Additions A-4

A.5 Changes to the ioctl Function A-4

A.6 Pathname Resolution A-4

A.7 Using the mount() and chown() Functions A-5

A.8 Super-Block Format A-5

A.9 Separate Version Libraries A-5

## A.1 Introduction



Note that in this chapter, XENIX 2.3 refers to an older version of XENIX. This is separate from XENIX System V version 2.3. All references to XENIX System V in this chapter include all releases of System V. All references to XENIX 2.3 refer to the pre-System V release numbering.

This appendix lists some of the differences among XENIX 2.3, XENIX 3.0, UNIX V7, UNIX System 3.0 and XENIX System V (all releases). It is intended to aid users who wish to convert system calls in existing application programs for use on other systems.

## A.2 Executable File Format

XENIX 3.0, UNIX System 3.0, and XENIX System V execute only those programs with the *x.out* executable file format. The format is similar to the old *a.out* format, but contains additional information about the executable file such as:

- text and data relocation bases
- target machine identification
- word and byte ordering
- symbol table
- relocation table format
- the revision number of the kernel which is used during execution to control access to system functions

XENIX System V has a segmented *x.out* header which contains segmentation information, as well as relocation information. To execute existing programs in *a.out* format, you must first convert to the *x.out* format. The format is described in detail in **a.out**(F) in the *XENIX User's Reference*.

XENIX System V uses *little-endian* (low order word first in memory) word order for *longs* whereas some XENIX 3.0 systems use *big-endian* (high order word first in memory) word order. XENIX System V checks the *x.out* header for information about the word order. XENIX System V maintains full XENIX 3.0 binary compatibility. XENIX System V executes XENIX 3.0 word-swapped (big-endian) executable files as well as XENIX 3.0 and XENIX System V (little-endian) executables.

## A.3 Revised System Calls

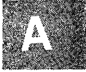
Some system calls in XENIX System V and UNIX System V have been revised and do not perform the same tasks as the corresponding calls in previous systems. To provide compatibility for old programs, XENIX System V and UNIX System V maintain both the new and the old system calls and automatically check the revision information in the *x.out* header to determine which version of a system call should be made.

The following table lists the revised system calls and their previous versions:

| System Call # | XENIX 2.3 function | System 3 function | System V function |
|---|---|---|---|
| 35 | ftime | unused | unused |
| 38 | unused | clocal | clocal |
| 39 | unused | setpgrp | setpgrp |
| 40 | unused | cxenix | cxenix |
| 57 | unused | utssys | utssys |
| 62 | clocal | fcntl | fcntl |
| 63 | cxenix | ulimit | ulimit |

The **cxenix** function provides access to system calls unique to XENIX 3.0 and XENIX System V. The **clocal** function provides access to all calls unique to an OEM.

The new XENIX System V system calls are accessed by means of **cxenix** system calls with their numbers. Note that these numbers are not regular system call numbers, but **cxenix** numbers. To use these calls, make the **cxenix** system call, with the high byte set to the appropriate number listed



below. For example, to call **locking**, take 40, add 256*1 to it, and pass the resulting value in register **ax** when trapping into the kernel. The following table lists the XENIX 3.0 and XENIX System V system calls:

| cxenix Call # | Description | System Call |
|---|---|---|
| 0 | shutdown OS | shutdown |
| 1 | record locking | locking |
| 2 | create semaphore | creatsem |
| 3 | open semaphore | opensem |
| 4 | signal semaphore | sigsem |
| 5 | wait semaphore | waitsem |
| 6 | nonblocking waitsem | nbwaitsem |
| 7 | blocking read check | rdchk |
| 8 | set stack limit | stkgrow |
| 9 | extended ptrace | xptrace |
| 10 | change file size | chsize |
| 11 | XENIX 2.3 ftime call | ftime |
| 12 | sleep for short interval | nap |
| 13 | attach to shared data | sdget |
| 14 | release shared data | sdfree |
| 15 | enter critical region | sdenter |
| 16 | leave critical region | sdleave |
| 17 | get shared data version # | sdgetv |
| 18 | wait for new shared data version | sdwaitv |

The following calls are found in XENIX System V only:

| cxenix Call # | Description | System Call |
|---|---|---|
| 19 | change segment size | brkctl |
| 22 | message control | msgctl |
| 23 | get message queue | msgget |
| 24 | send message | msgsnd |
| 25 | receive message | msgrcv |
| 26 | semaphore control | semctl |
| 27 | get semaphore set | semget |
| 28 | semaphore ops | semop |
| 29 | sysV shared memory control | shmctl |
| 30 | sysV create shared memory | shmget |
| 31 | sysV attach shared memory | shmat |

## A.4 Version 7 Additions

XENIX System V maintains a number of XENIX 3.0 and UNIX V7 features that were dropped from UNIX System 3.0. In particular, XENIX System V continues to support the **dup2**(S) and **ftime**(S) functions. The **ftime** function, used with the **ctime**(S) function, provides the default value for the time zone when the TZ environment variable has not been set. This means a binary configuration program can be used to change the default time zone. No source license is required.

## A.5 Changes to the ioctl Function

XENIX 3.0 and UNIX System 3.0 have full sets of XENIX 2.3-compatible **ioctl** calls. Furthermore, XENIX 3.0 and XENIX System V have resolved problems that previously hindered UNIX System 3.0 compatibility. For convenience, XENIX 2.3-compatible **ioctl** calls can be executed by a UNIX System 3.0 executable. The available XENIX 2.3 **ioctl** calls are:

| | |
|---|---|
| TIOCSETP | TIOCSETN |
| TIOCGETP | TIOCSETC |
| TIOCGETC | TIOCEXCL |
| TIOCNXCL | TIOCHPCL |
| TIOCFLUSH | TIOCGETD |
| TIOCSETD | |

## A.6 Pathname Resolution

If a null pathname is given, XENIX 2.3 interprets the name to be the current directory, but UNIX System 3.0 considers the name to be an error. XENIX 3.0 and XENIX System V use the version number in the *x.out* header to determine what action to take. A XENIX 2.3 header causes null pathnames to be the current directory. Any other version is interpreted as an error.

If the symbol ".." is given as a pathname when in a root directory that has been defined using the **chroot**(S) function, XENIX 2.3 moves to the next higher directory. XENIX 3.0 also lets the ".." symbol move the current working directory to the next higher directory, but restricts its use to the super user. XENIX System V does not let the ".." symbol move the current working directory to the next higher directory.

## A.7 Using the mount() and chown() Functions

XENIX 3.0, and UNIX System 3.0 restrict the use of the **mount**(S) system call to the super user. XENIX System V does not restrict the use of the **mount** system call. However, usually the **mount**(C) function is only executable by the super-user. Also, XENIX System V, XENIX 3.0 and UNIX System 3.0 let the owner of a file use the **chown**(S) function to change the file ownership.

## A.8 Super-Block Format

XENIX System V, UNIX System 3.0 and UNIX System 5.0 have new super block formats. XENIX System V and XENIX 3.0 use the System 5.0 format, but use a different magic number for each revision. The XENIX System V and XENIX 3.0 super blocks each have an additional field at the end that can be used to distinguish between XENIX 2.3, 3.0 and System V super blocks. XENIX System V and XENIX 3.0 check this magic number at boot time and during a mount. If a XENIX 2.3 super block is read, XENIX 3.0 converts it to the new format internally. Similarly, if a XENIX 2.3 super block is written, XENIX 3.0 converts it back to the old format. This permits XENIX 2.3 kernels to be run on file systems also usable by UNIX System 3.0.

However, XENIX System V is word-swapped relative to XENIX 3.0. Even though the super block formats are the same, the order of bytes in long words is different. XENIX System V's **mount**(C) and **fsck**(C) commands cannot be used to mount or check XENIX 3.0 filesystems.

## A.9 Separate Version Libraries

XENIX System V supports the construction of XENIX 3.0 executable files. This systems maintains both the new and the old versions of system calls in separate libraries.

# Appendix B

# Kernel Error Messages

B.1 Introduction B-1

B.2 Informational Messages B-1
  B.2.1 Memory Layout Messages B-1
  B.2.2 Boot Sequence Messages B-2

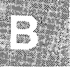
B.3 Warning Messages B-2

B.4 Panic Messages B-5
  B.4.1 Kernel Panic Messages B-5
  B.4.2 286 Panic Messages B-7
  B.4.3 386 Panic Messages B-8

## B.1 Introduction

This appendix lists the messages that the XENIX kernel displays while booting or running the operating system.

The XENIX system error messages fall into three categories:



- *informational messages* that give you information about memory layout and the status of the boot sequence,
- *warning messages* that are informational only, and do not prevent the XENIX kernel from running. These messages inform you of a possible system problem that you may need to correct.
- *panic messages* that indicate a severe problem, one that prevents the XENIX kernel from running. After displaying a message about the panic error, the operating system shuts down and starts the rebooting process.

## B.2 Informational Messages

This section lists memory layout and boot sequence messages.

### B.2.1 Memory Layout Messages

When XENIX is started, it displays information about the memory layout of the system. The total amount of memory allocated for all of the following categories should equal the amount of memory on your system. The memory messages are listed below:

`Buffers =`*x*`K`
: The value of *x* specifies the amount of memory used by the I/O buffers.

`Kernel memory =`*x*`K`
: The value of *x* specifies the amount of memory used by the XENIX kernel.

`Reserved memory =`*x*`K`
: The value of *x* specifies the amount of memory between physical address zero and the load address for the kernel.

`User memory =`*x*`K`
: The value of *x* specifies the amount of memory remaining on the system for user processes.

### B.2.2 Boot Sequence Messages

The XENIX kernel displays messages that let you know the status of the boot sequence:

`Automatic Boot Procedure`
: This message displays if the XENIX system is automatically rebooting after a system shutdown. The system automatically reboots if a manual reboot does not occur within a specified time after the system has been shut down. The specified time varies depending on your target hardware.

`Press Enter to reboot:`
: This message is displayed after the XENIX system is halted using the **shutdown** or **haltsys** command. For more information, see *shutdown*(ADM) and *haltsys*(ADM) in the *XENIX System Administrator's Guide.*

## B.3 Warning Messages

The messages listed in this section indicate potential system problems, but these problems do not prevent the kernel from running. Please refer to your Operating System documentation for information on tuning system parameters.

`Can't allocate message buffer`
: This message appears on the system console during the XENIX boot sequence if memory cannot be allocated for the message buffers used by the interprocess communication (IPC) system calls. This condition indicates an internal system error unless there is an inadequate amount of memory present on the system. Your system must have at least 512 Kbytes of memory to give reasonable performance.

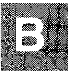
`dscrfree: Cannot free descriptor` *sel-no*
: The **dscrfree** routine is called from the device driver to free segment selectors used by the driver. This message is displayed if the driver cannot free the specified selector (*sel-no*); in other words, the selector is not accessible to the device driver.

`error on dev` *device* (*majordev#/minordev#*), `block=`*bnumber* `cmd=`*command* `status=`*devstatus*
: An error occurred when the kernel tried to access the block device identified by *device*. The *bnumber* indicates what block was being read or written. The *command* indicates the

type of access that failed: read, write, format, or other command. The *devstatus* is the error status read in from the device.

`Inode table overflow`

When the kernel tried to access a file, the inode table was full. The parent process exited with *errno* set to ENFILE.



`Interrupt from unknown device, vec=`*vector-number*

The kernel received an interrupt from a vector that is not assigned to any device.

`Map overflow (`*map-type-indicator*`), shutdown and reboot`

The map specified by the *map-type-indicator* is fragmented to such an extent that there are not enough map elements to keep track of all the resource pieces. The map is either the core map, the swap map, the message map used in interprocess communication, or the semaphore map.

`Maxmem was reduced based on the size of the swap area. Refer to the system documentation for information on the relationship between memory size and swap size.`

*Maxmem* represents the maximum amount of memory available to one user process. This is normally 75 percent of all available user memory. However, the swap area must be at least as big as *maxmem*, since the largest process allowed must be able to swap out if necessary. If, on start-up, the kernel discovers that *maxmem* is larger than the swap area, the kernel reduces *maxmem* to approximately the size of the swap area as stated in this message.

If you receive this warning message when you start your XENIX system, you need to increase the size of the swap area on your file system to make maximum use of the memory available on your machine. If you decide that the reduced size of *maxmem* (approximately the size of your swap area) is acceptable, you do not need to make any changes in response to this message. For more information on how to run processes that are too big to swap, see **runbig**(ADM) in the *XENIX System Administrator's Guide* and **proctl**(S) in the *XENIX Programmer's Reference*.

If you decide you want to increase the size of the swap area, follow these instructions carefully. The first step of this process is to back up the whole disk. You must back up the disk

because the file system will have to be reloaded after you reconfigure your kernel with a bigger swap area.

Next, in the file */usr/sysv/conf/xenixconf* , change the value specified for the swap area . The line looks like this:

```
swap disk 5     6000 3000
```

This line specifies a swap area of 3000 Kbytes beginning at 6000 Kbytes on the disk. If you increase the size of the swap area, increase the value 3000 to the size in kilobytes that you need. Relink the kernel with the new swap area. If your disk is partitioned into separate root and user file system partitions, you will need to increase the size of the root partition to accommodate the additional swap space. This means the user partition size will probably need to be decreased. If you will be distributing a system with a larger swap area size, you should modify the **hdinit** program.

You can change the value of *maxmem* from its default of 75 percent of user memory. The default value of 75 percent is specified by setting the *maxprocmem* parameter to zero in the */usr/sysv/conf/master* file. If you want to change *maxmem*, define a non-zero value for *maxprocmem* in the */usr/sysv/conf/xenixconf* file and relink a new version of XENIX. The *xenixconf* file is used to specify exceptions to the default system configuration and overrides the parameters specified in the *master* file. The *master* file is used to define the default system configuration and should not be changed. The *maxprocmem* parameter specifies the maximum size of a user process in kilobytes. This number can be no larger than 75 percent of the size of your swap area.

`no file`

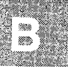
There is no room in the file table structure to make an entry for the file that the kernel is trying to allocate. Since semaphores also use entries in the file table structure, this message can also indicate that a process with active semaphores has been unable to pass those semaphores using a **fork**() routine.

`Out of device descriptors, increase gdt size (NGDT) and relink XENIX`

The number of global descriptor table (GDT) entries reserved for device drivers is inadequate to fill the requests of the drivers on the system. A device driver has requested a GDT entry and has been refused because the table is full. You must increase the value assigned to NGDT in */usr/sysv/h/machdep.h*

and relink the kernel after removing the old configuration file */usr/sysv/conf/c.c*.

`out of text`
There is no room in the text table structure for a process being executed. The process is killed and can be resubmitted at a later time.



`proc on q`
The scheduler tried to add a process to the ready-to-run queue that was already waiting on the queue. This condition represents an internal system error.

`WARNING: SYSTEM MODE TYPE 7`
A trap Type 7 (Processor Extension Not Available Exception) has occurred in system mode on a system that has a 287 chip.

## B.4 Panic Messages

The panic messages indicate a severe error condition, one that forces the XENIX system to shut down. You will probably not see the majority of these messages. After displaying a panic message, the XENIX system immediately terminates and starts the rebooting process. The panic messages fall into three categories: those generated because of kernel problems, those generated by the 286 microprocessor, and those generated by the 386 microprocessor hardware.

### B.4.1 Kernel Panic Messages

This section lists the panic messages generated because of kernel problems.

`panic: blkdev`
During I/O buffer allocation, the kernel detects that the major number of the device specified exceeds the maximum defined. This error condition occurs deep in kernel system call processing and should not happen, since the possibility has been checked in earlier system code.

`panic: devtab`
During I/O buffer allocation, the kernel cannot locate the device specified in its tables. This error condition occurs deep in kernel system call processing and should not happen, since the possibility has been checked in earlier system code.

`panic: iinit`
The superblock of the root file system cannot be read on system startup. This means that a device error has occurred and could mean that the root device needs repair.

`panic: IO err in swap`
The kernel has encountered an error on the swap device while trying to swap a process.

`panic: Kernel buffer crosses 64k boundary, change load address`
This message means that the kernel buffer crossed a 64 Kbyte boundary. When CTRL_16BIT is defined in */usr/sysv/h/machdep.h*, the XENIX kernel cannot have any of its I/O buffers spanning a 64 Kbyte address boundary. You can correct this problem by specifying a different XENIX load address to the boot program when the system is booted.

`panic: memory failure - parity error`
A parity error has occurred on one of the memory boards of the system. You should check the memory boards and replace the faulty one.

If the system contains 1ECC (Error Correcting Code) memory boards, this message is slightly different and could indicate the board that caused the error. The XENIX system only shuts down if the problem cannot be corrected on the 1ECC memory board.

`panic: memory management failure`
One of the memory management routines has encountered an error when doing critical copying or allocating functions. This indicates an internal kernel error and not an equipment malfunction.

`panic: no fs`
The device specified in a command is not currently mounted. This error condition occurs deep in kernel system call processing and should not happen, since the possibility has been checked in earlier system code.

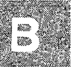
`panic: no imt`
A mounted file system does not have an entry in the mount table. This panic condition should not occur, since the mount table was checked in previous code, but this message is included as a safety check.

`panic: no procs`

This panic condition occurs if a process table entry cannot be found for the process being forked. This should not happen, since code reached prior to this point has already checked for the existence of a process table slot.

`panic: Out of swap`

The kernel ran out of free space on the swap device when the kernel attempted to swap a process. If this error occurs more than once, you should increase the size of the swap area.




`panic: preadi`

An error occurred when the system attempted to read in a process' segments during an **exec** call. This condition represents an internal system failure.

`panic: Small model shared data copy failure`

An error occurred when the system attempted to update a process' shared data segments during a process switch. This condition represents an internal system failure.

`panic: Timeout table overflow`

There is no room in the callout (or timeout) table for a new entry. You can do nothing when this problem occurs except let the system shutdown and reboot.

`panic: unknown interrupt`

The kernel received an interrupt from a vector that is not assigned to any device.

## B.4.2 286 Panic Messages

When the active process causes an exception or trap on the 286 microprocessor that cannot be handled by software, one of the following panic messages is displayed:

`panic: general protection trap`

A trap type 13 (General Protection) occurred while the 286 was in system mode. This condition represents an internal kernel error.

`panic: Invalid TSS`

A trap type 10 (Invalid Task State Segment) occurred.

`panic: Segment Not Present`
A trap type 11 (Segment Not Present) occurred. This condition means that an attempt has been made to access a segment that is marked as not present.

`panic: Trap in system`
This is the default message for system mode traps. The information displayed prior to the list of register contents specifies what kind of trap occurred.

`panic: TRAP` *violation #* `in` *mode*
A trap violation has occurred on the 286 microprocessor. The *violation #* specifies the vector number of the exception. The *mode* specifies whether the 286 was executing user code (user mode) or kernel code (system mode) when the trap occurred. Before this message is given, a list of the contents of all the 286 registers is displayed. See the Intel *iAPX 286 Programmer's Reference* for a description of the 286 trap vectors.

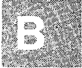
`panic: 287 Exception`
The 286 thinks it received a trap from the 287, but there is no 287 on the system.

`panic: 287 Segment Overrun`
A trap type 9 (Processor Extension Segment Overrun Exception) has occurred on a system that has a 287. This trap is caused when the 287 overruns the limit of a segment while attempting to read or write the second or subsequent words of an operand. If a system without a 287 generates this exception, it will be killed with a segment violation signal (SIGSEGV).

### B.4.3 386 Panic Messages

When the active process causes an exception or trap on the 386 microprocessor that cannot be handled by software, the system displays a 386-specific panic message. You must reboot the system after the kernel displays a panic message. The following list describes the 386-specific panic messages:

`panic:added strange page table`
The system has attempted to handle a page fault in an unexpected region of a process' address space; the system's protection mechanisms prevents this from occurring.




`panic:bad page type for protection fault`
The system detected an abnormal situation while processing a page fault.

`panic:protection fault on read access`
The system detected an abnormal situation while processing a page fault.

`panic:xlcheck: xlink serial mismatch`
The system detected an abnormal situation while processing a page fault.

`panic:swapping intransit page`
The system detected an abnormal situation while processing a page fault.

`panic:buildpt: page directory already used`
While loading a user program, the system attempted to load two sets of data into the process' data address space.

`panic:dfalloc: frame not free at exit`
The system attempted to allocate a region on the swap device from the list of free regions, but discovered that the region was already in use.

`panic:dftodp: bad frameno %x`
The paging subsystem attempted to handle a reference to a non-existent region on the swap device.

`panic:dptodf: bad dp %x`
The paging subsystem attempted to handle a reference to a non-existent region on the swap device.

`panic:dftomf: non-swap page table entry changed`
A locked entry in the page table was altered.

`panic:dftomf: swap disk frame rcnt != 1`
The paging subsystem referenced more than one region on the swap device; only one reference is possible.

`panic:mftodf: swap disk frame rcnt != 1`
The paging subsystem referenced more than one region on the swap device; only one reference is possible.

`panic:dftomf: swap mem frame rcnt != 1`
The paging subsystem referenced more than one region on the swap device; only one reference is possible.

`panic:dftomf: swap memory frame rcnt != 1`
The paging subsystem referenced more than one region on the swap device; only one reference is possible.

`panic:mftodf: swap mem frame rcnt != 1`
The paging subsystem referenced more than one region on the swap device; only one reference is possible.

`panic:impcode(): called to load impure 386 !!`
The system detected an abnormal condition while loading a program.

`panic:impcode(): more than 1 data segment ??`
The system detected an abnormal condition while loading a program.

`panic:preload(): invalid page (%x, %x)`
The system detected an abnormal condition while loading a program.

`panic:lverify: confused 286 segment`
The paging subsystem detected an inconsistency in the structure of a process while processing a page fault.

`panic:mfalloc: page not free`
The system attempted to allocate a region of memory from the list of free regions, but discovered that the region was already in use.

`panic:mfalloc: page not free at exit`
The system attempted to allocate a region of memory from the list of free regions, but discovered that the region was already in use.

`panic:mfcvt: zero ref count`
The system discovered that no references were actually made to the page frame.

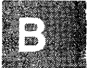
`panic:mffree: page already free`
The system attempted to free a region of free memory.

`panic:mffree: page is locked`
The system attempted to free a region of locked memory.

`panic:mftodf: memory frame marked in transit`
The system attempted to free a region of I/O pending memory.

`panic:mftomp: bad frameno`
The paging subsystem referenced a non-existent region of memory.

`panic:mptomf: bad mp %x`
The paging subsystem referenced a non-existent region of memory.




`panic:not enough contiguous memory`
The system did not satisfy a request for contiguous memory. An extremely large memory request was made, or memory is configured in discontiguous frames.

`panic:not present fault on shared data`
The system detected an abnormal page fault in a shared data segment.

`panic:page table under page table?`
The system detected a page type error.

`panic:page type mismatch`
The system detected an error in page-typing information.

`panic:pgcheck`
The system detected an error in page-typing information.

`panic:pgfind`
The system detected an inconsistency in its memory page cache.

`panic:pghash`
The system detected an inconsistency in its memory page cache.

`panic:pginval: list broken`
The system detected an inconsistency in its memory page cache.

`panic:pginval: not in cache`
The system detected an inconsistency in its memory page cache.

`panic:pgfree: freeing intransit page`
The system detected an abnormal page while freeing memory used by a dead process.

`panic:pgfree: invalid page marked present`
The system detected an abnormal page while freeing memory used by a dead process.

`panic:pgread: no xlink`
After reading a page from swap area, the system discovered that no frames were available to store the data.

`panic:ptdup: TE(S)WAP page rcnt > 1`
The system detected an abnormal condition while duplicating a process with **fork**.

`panic:ptdup: intransit page`
The system detected an abnormal condition while duplicating a process with **fork**.

`panic:ptdup: locked page not present`
The system detected an abnormal condition while duplicating a process with **fork**.

`panic:ptdup: xlinked page has reference`
The system detected an abnormal condition while duplicating a process with **fork**.

`panic:sptmap overflow`
The system attempted to allocate unavailable address space.

`panic:swap io error`
The system detected a hardware failure while reading data to or writing it from the disk-swap area.

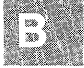
`panic:swapping TE_TABLE page`
The system detected a process trying to swap a page when paging was not enabled at compile time.

`panic:bad boot string`
The default boot string in the */etc/default/boot* file or the string entered in response to the boot prompt was syntactically incorrect.

`panic:bad mapping in copyio`
The system could not detect the direction to copy data to or from a buffer.

`panic:fpsave: no fp_task`
The system attempted to save the current state of the math coprocessor when no process was executing on the coprocessor.

`panic:fp_OVERRUN: coprocessor overrun - with no 287/387`
The system detected a math coprocessor overrun with no math processor installed on the system.




`panic:fp_COPROC: coprocessor error - with no 287/387`
The system detected a math coprocessor error with no math processor installed on the system.

`panic:fp_COPROC: coprocessor error - switched away from fp_task`
The system detected a math coprocessor error with no process running on the coprocessor.

`panic:fp_DNA: called when we have an emulator.`
The system detected a Type 7 (Device Not Available) trap when emulating a floating point processor through software.

`panic:srmount(): cannot cvtv7superb() yet`
The system attempted to mount a Version 7 root filesystem. XENIX does not currently support Version 7 root filesystems.

`panic:physio: bad state`
The system detected an abnormal condition when preparing buffers for a physical I/O operation.

`panic:bad interrupt handler`
An unknown hardware or software source initiated an interrupt.

`panic:u-area not page aligned`
The system detected that the u-area is not page aligned.

`panic:u-area address does not match SPTADDR`
The system u-area address has not been initialized correctly.

`panic:sdfrcm: sdp->sd_inode not found`
The system detected a shared data region with no associated inode.

`panic:lost text`
The system detected an active shared text segment with no associated memory.

`panic:xexpand: no proc refers to text`

The system discovered a text segment not associated with any process.

`panic:non-recoverable kernel page fault`

The system could not process a page fault.

`panic:DNA trap in kernel mode`

The system detected a Type 7 (Device Not Available) trap when in *system mode*.

`panic:trap`

The system detected an unknown trap.

`panic:floating point int in kernel`

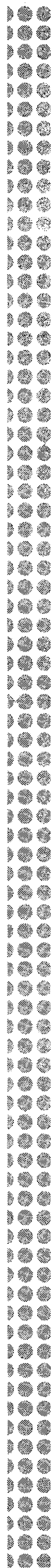
The system detected a software floating point interrupt when in system mode.

`panic:write_sb(): cannot cvts3superb() yet`

The system attempted to write out a Version 7 superblock. XENIX does not support Version 7 filesystems.

Replace this Page
with Tab Marked:

# Index



# Index

## A

-a option
  lint 4-10
adb
  backtrace 8-21
  binary files 8-43
  breakpoints 8-18
  combining commands on a single line 8-37
  computing numbers and
    displaying text 8-40
  core image files 8-2
  creating scripts 8-37
  current address 8-8
  data files 8-3
  data formats 8-11
  decimal integers 8-6
  deleting breakpoints 8-21
  displaying CPU registers 8-22
  displaying external variables 8-23
  displaying instructions and data 8-4
  exiting 8-4
  forming addresses 8-5
  forming expressions 8-5
  hexadecimal integers 8-6
  introduction 8-1
  killing a program 8-21
  leaving 8-4
  locating values in a file 8-43
  making changes to memory 8-45
  memory maps 8-32
  miscellaneous features 8-37
  octal integers 8-6
  operators 8-10
  patching binaries 8-43
  program execution 8-16
  prompt option 8-4
  register names 8-9
  setting default input format 8-39
  setting output width 8-38
  setting the maximum offset 8-39
  single-stepping a program 8-20
  starting and stopping 8-1
  symbols 8-6
  the ? and / commands 8-14
  the = command 8-13
  using XENIX commands 8-40
  validating addresses 8-36
  variables 8-7
adb *(continued)*
  write option 8-3
  writing to a file 8-44
a.out
  link editor 9-1, 9-2, 9-3, 9-4
ar
  description 1-2
Arguments
  macro 10-6
As
  basic tool 1-1
Assembler *See* As

## B

-b option
  lint 4-6
Boot messages B-1, B-2
Boot status B-1

## C

C compiler
  expression
    evaluation order 4-13
  lint directives, effect 4-13
C language
  yacc 6-2
-c option
  lint 4-9
C programming language 1-1
changequote function 10-4
Command
  execution 1-3
  interpretation 1-3
  SCCS commands *See* SCCS
  SCCS *See* SCCS
Conditionals
  m4 macro processor 10-9
Core images
  examining 11-1
CS register 9-8
csh
  description 1-3
C-shell
  command history mechanism 1-3
  command language 1-3

## D

Data segments 9-8
Debugger
  displaying data 11-1
  displaying instructions 11-1
  starting sdb 11-1
  stopping sdb 11-1
Debugging
  adb 8-1
Debugging programs
  sdb 11-1
Debugging tools
  sdb 11-1
define function 10-2
Defining macros 10-2
Delta *See* SCCS
dempdef function 10-11
Desk calculator
  specifications 6-35
divert function 10-8
divnum function 10-9
dnl function 10-11
DS register 9-8
dscrfree( ) routine B-2

## E

Error message file
  creation 1-3
Error messages
  information B-1
  panic B-1, B-6
  panic messages B-5
  system traps B-7, B-8
  warning B-1, B-2, B-3, B-4
eval function 10-7
Executing programs
  controlling 11-1
  monitoring 11-1

## F

File
  archives 1-2
  error message file *See* Error message file
  removal
    SCCS use *See* SCCS
File *(continued)*
  Source Code Control System *See* SCC
fork( ) routine B-4
FORTRAN
  conversion program 5-23

## G

Global
  declaration
    link editor 9-6
  variable
    communal variable allocation 9-5
    uninitialized 9-5
Global descriptor table
  messages B-4

## H

-h option
  lint 4-11
haltsys command B-2
Huge model
  integer size 9-6
  segment structure 9-8
Huge model 7 default names 9-6
Huge model 7 pointer size 9-6

## I

ifdef function 10-5
ifelse function 10-9
include function 10-8
  silent alternate
    sinclude function 10-8
incr function 10-7
Information messages B-1
inode table B-3
Integer
  size in memory model 9-6
Intel 286 microprocessor
  messages B-7
Intel 386 microprocessor
  messages B-8

# K


Kernel
  memory B-1
  panic messages B-5, B-6


# L


Large model
  integer size 9-6
  segment structure 9-8
Large model 7 default names 9-6
Large model 7 pointer size 9-6
ld
  *See* Link editor 9-1
  basic tool 1-1
len function 10-10
lex
  0, end of file notation 5-14
  action
    default 5-10
    description 5-4
    repetition 5-10
    specification 5-10
  alternation 5-8
  ambiguous source rules 5-14
  angle brackets (<>)
    operator character 5-5, 5-27
    start condition referencing 5-18
  a.out file
    contents 5-6
  arbitrary character match 5-7
  array size change 5-26
  asterisk (*)
    operator character 5-5, 5-27
    repeated expression specification 5-8
  automaton interpreter
    initial condition resetting 5-18
  backslash (\)
    C escapes 5-5
    operator character 5-5, 5-27
    operator character escape 5-5, 5-7
  BEGIN
    start condition entry 5-18
  blank character
    quoting 5-5
    rule ending 5-5
  blank, tab line beginning 5-19
  braces ({ })
    expression repetition 5-9
    operator character 5-5, 5-27
lex *(continued)*
  brackets ([ ])
    character class specification 5-6
    character class use 5-2
    operator character 5-5, 5-27
    operator character escape 5-6
  buffer overflow 5-15
  C escapes 5-5
  caret (^)
    character class inclusion 5-7
    context sensitivity 5-9
    operator character 5-5, 5-27
    string complement 5-7
  caret (^) operator
    left context recognizing 5-17
  character
    internal use 5-25
    set table 5-25, 5-26
    translation table *See* set table
  character class
    notation 5-2
    specification 5-6
  character set
    specification 5-25
  context sensitivity 5-9
  copy classes 5-19
  dash (-)
    character class inclusion 5-7
    operator character 5-5, 5-27
    range indicator 5-6
  definitions
    character set table 5-25
    contents 5-20, 5-26
    expansion 5-9
    format 5-20, 5-26
    location 5-20
    placement 5-10
    specification 5-19
  delimiter
    discard 5-20
    rule beginning marking 5-2
    source format 5-4
    third delimiter, copy 5-20
  description 1-2
  dollar sign ($)
    context sensitivity 5-9
    end of line notation 5-2
    operator character 5-5, 5-27
  dollar sign ($) operator
    right context recognizing 5-17
  dot (.) operator *See* period (.)
  double precision constant change 5-23
  ECHO
    format argument, data printing 5-11

lex *(continued)*
  end-of-file
    0 handling 5-14
    yywrap routine 5-14
  environment
    change 5-17
  expression
    new line illegal 5-6
    repetition 5-9
  external character array 5-11
  flag
    environment change 5-17
  FORTRAN conversion program 5-23
  grouping 5-8
  input
    end-of-file, 0 notation 5-14
    ignoring 5-10
    manipulation restriction 5-17
  input () routine 5-13
  input routine
    character I/O handling 5-25
  invocation 5-6
  I/O library *See* library
  I/O routine
    access 5-13
    consistency 5-14
  left context 5-9
    caret (^) operator 5-17
    sensitivity 5-17
  lexical analyzer
    environment change 5-18
  lex.yy.c file 5-6
  library
    access 5-6
    avoidance 5-6
    backup limitation 5-14
    loading 5-21
  line beginning match 5-9
  line end match 5-9
  -ll flag
    library access 5-6
  loader flag *See* -ll flag
  lookahead characteristic 5-12, 5-14
  match count 5-11
  matching
    occurrence counting 5-16
    preferences 5-14
  new line
    illegality 5-6
  newline
    escape 5-25
    matching 5-15
  octal escape 5-7
  operator characters

lex *(continued)*
  operator characters *(continued)*
    *See also* Specific Operator Characte
    designated 5-27
    escape 5-5, 5-6, 5-7
    listing 5-5
    literal meaning 5-5
    quoting 5-5
  optional expression
    specification 5-7
  output (c) routine 5-13
  output routine
    character I/O handling 5-25
  overview 5-1
  parentheses (())
    grouping 5-8
    operator character 5-5, 5-27
  parser generator
    analysis phase 5-2
  percentage sign (%)
    delimiter notation (%%) 5-2
    operator character 5-5
    remainder operator 5-21
    source segment separator 5-10
  period (.)
    arbitrary character match 5-7
    newline no match 5-15
    operator character 5-5
  period (.) operator
    designated 5-27
  plus sign (+)
    operator character 5-5, 5-27
    repeated expression specification 5-
  preprocessor statement entry 5-20
  question mark (?)
    operator character 5-5, 5-27
    optional expression specification 5-
  quotation marks, double ( 5-5, 5-27
  real numbers rule 5-20
  regular expression
    description 5-4
    end indication 5-4
    operators *See* operator characters
    rule component 5-4
  REJECT 5-16
  repeated expression
    specification 5-8
  right context
    dollar sign ($) operator 5-17
  rules
    active 5-19
    components 5-4
    format 5-27
    real number 5-20

lex *(continued)*
semicolon (;)
null statement 5-10
slash (/)
operator character 5-5, 5-27
trailing text 5-9
source
copy into generated program 5-19
description 5-1
format 5-3, 5-19
interception failure 5-19
segment separator 5-10
source definitions
specification 5-19
source file
format 5-26
source program
compilation 5-6
spacing character ignoring 5-10
start
abbreviation 5-18
start condition 5-9
start conditions
entry 5-18
environment change 5-17
format 5-26
location 5-26
statistics gathering 5-22
string
printing 5-4
substitution string
definition *See* definition
tab line beginning *See* blank, tab line beginning
text character
quoting 5-5
trailing text 5-9
unput
REJECT noncompatible 5-17
unput (c) routine 5-13
unput routine
character I/O handling 5-25
unreachable statement 4-6
vertical bar (|)
action repetition 5-10
alternation 5-8
operator character 5-5, 5-27
wrapup *See* yywrap routine
yacc
interface 5-2
library loading 5-21
yacc interface
tokens 5-21
yylex () 5-21

lex *(continued)*
yyleng variable 5-11
yyless ()
text reprocessing 5-13
yyless (n) 5-12
yylex () program
yacc interface 5-21
yylex program
contents 5-1
yymore () 5-12
yytext
external character array 5-11
yywrap () 5-22
yywrap () routine 5-14
Library
conversion 1-2
maintenance 1-2
Link editor
a.out 9-1, 9-2, 9-3, 9-4
_BSS 9-4
command line 9-1
communal variable
allocation of 9-5
_DATA 9-4
error message 9-5
global declaration 9-6
global variable 9-5
introduction 9-1
memory model 9-3
object files 9-4
options 9-1
-A 9-2
-B 9-2
-c 9-2
-D 9-2
-F 9-2
-g 9-2
-i 9-2
-M 9-3
-n 9-3
-o 9-3
-P 9-3
-R 9-3
-s 9-4
-u 9-4
-v 9-4
packing, disable 9-3
relocatable object module 9-3
stack size 9-2
_TEST 9-4
using 9-1
variable allocation rules 9-5
lint
-a option 4-10

lint *(continued)*
ARGSUSED directive 4-14, 4-15
argument number comments turnoff 4-14
assignment
of long to int
check 4-10
operator
operand type balancing 4-8
assignment, implied *See* implied assignment
assignment operator
new form 4-12
old form, check 4-11
-b option 4-6
binary operator, type check 4-8
break statement
unreachable *See* unreachable break statement
-c option 4-9
C program check 4-1
C syntax, old form, check 4-11
cast *See* type cast
conditional operator
operand type balancing 4-8
constant in conditional context 4-11
construction check 4-1, 4-10
control information flow 4-14
degenerate unsigned comparison 4-10
description 4-1
directive
defined 4-13
embedding 4-13
enumeration, type check 4-8
error message, function name 4-7
expression, order 4-13
extern statement 4-4
external declaration, report suppression 4-4
file
library declaration file identification 4-14
function
error message 4-7
return value check 4-7
type check 4-8
unused *See* unused function
-h option 4-11
implied assignment, type check 4-8
initialization, old style check 4-12
library
compatibility check 4-15
suppression 4-15
directive acceptance 4-15
file processing 4-15
LINTLIBRARY directive 4-14, 4-15
loop check 4-6
-ly directive 4-15

lint *(continued)*
-n option 4-15
nonportable
character check 4-9
expression evaluation order check 4-13
NOSTRICT directive 4-14
NOTREACHED directive 4-14
operator
operand types balancing 4-8
precedence 4-11
output turnoff 4-13
-p option 4-15
pointer
agreement 4-8
alignment check 4-12
program flow
control 4-5
relational operator
operand type balancing 4-8
scalar variable check 4-13
source file
library compatibility check 4-15
statement
unlabeled report 4-5
structure selection operator
type check 4-8
syntax 4-1
type cast
check 4-9
comment printing control 4-9
type check
description 4-8
turnoff 4-14
-u option 4-4
unreachable break statement
report suppression 4-6
unused argument
report suppression 4-4
unused function
check 4-4
unused variable
check 4-4
-v option
turnon 4-14
unused variable report suppression 4-4
VARARGS directive 4-14, 4-15
variable
external variable initialization 4-5
inner/outer block conflict 4-11
set/used information 4-4
static variable initialization 4-5
unused *See* unused variable
-x option 4-4
Loader *See* ld

longjmp function 7-11
Loop
  lint use *See* lint

## M

m4
  arithmetic values 10-7
  basic operation 10-1
  changequote function 10-4
  define function 10-2
  dempdef function 10-11
  description 1-2, 10-1
  divert function 10-8
  divnum function 10-9
  dnl function 10-11
  eval function 10-7
  ifdef function 10-5
  ifelse function 10-9
  include function 10-8
  incr function 10-7
  len function 10-10
  maketemp function 10-9
  overview 10-1
  printing function 10-11
  sinclude function 10-8
  substr function 10-10
  syscmd function 10-9
  translit function 10-10
  undefine function 10-5
  undivert function 10-8
  using arguments 10-6
m4 macro processor
  invoking 10-2
m4 *See* Also Macro processor
Macro processor
  arguments 10-6
  arithmetic values 10-7
  built-in function
    changequote 10-4
    define 10-2
    dempdef 10-11
    divert 10-8
    divnum 10-9
    dnl 10-11
    eval 10-7
    ifdef 10-5
    ifelse 10-9
    include 10-8
    incr 10-7
    len 10-10
    maketemp 10-9
Macro processor *(continued)*
  built-in function *(continued)*
    printing 10-11
    sinclude 10-8
    substr 10-10
    syscmd 10-9
    translit 10-10
    undefine 10-5
    undivert 10-8
  conditionals 10-9
  defining macros 10-2
  functions 10-1
  m4 10-1
  manipulating strings 10-10
  printing 10-11
  quoting 10-3
  redirecting input 10-2
  redirecting output 10-2
  removing functions 10-5
  standard input 10-2
  standard output 10-2
  system commands 10-9
Macros 10-1
  built-in 10-1
  preprocessing 1-2
  user defined 10-1
Maintainer *See* Make
make
  argument quoting 2-7
  backslash (\)
    description file continuation 2-2
Make
  basic tool 1-2
  .c suffix 2-11
make
  command
    form 2-1
    location 2-1
Make
  command
    print without execution 2-15
make
  command argument
    macro definition 2-7
  command string
    hyphen (-) start 2-6
  command string substitution 2-6
Make
  -d option 2-15
make
  .Default 2-6
  dependency line
    form 2-1
  dependency line substitution 2-6

make *(continued)*
  description file
    comment convention 2-2
    macro definition 2-7
  description filename
    argument 2-5
  dollar sign ($)
    macro invocation 2-7
  equal sign (=)
    macro definition 2-6
Make
  .f suffix 2-11
  file
    time, date printing 2-15
    updating 2-15
make
  file generation 2-6
  file update 2-1
  hyphen (-)
    command string start 2-6
  .IGNORE 2-6
Make
  .l suffix 2-11
make
  macro
    definition 2-7
    definition override 2-7
    invocation 2-7
    substitution 2-6
    value assignment 2-7
  macro definition
    analysis 2-7
Make
  macro definition
    argument 2-5
make
  macro definition
    description 2-6
  medium sized projects 2-1
  metacharacter expansion 2-1
Make
  -n option 2-15
make
  number sign (#)
    description file comment 2-2
Make
  .o suffix 2-11
  object file
    suffix 2-11
make
  option argument
    use 2-5
  parentheses (( ))
    macro enclosure 2-7
make *(continued)*
  .Precious 2-6
  program maintenance 2-1
Make
  .r suffix 2-11
  .s suffix 2-11
make
  semicolon (;)
    command introduction 2-1
  .Silent 2-6
Make
  source file
    suffixes 2-11
  source grammar
    suffixes 2-11
  suffixes
    list 2-11
    table 2-11
  -t option 2-15
make
  target file
    pseudo-target files 2-6
Make
  target file
    update 2-15
make
  target filename
    argument 2-5
  target name omission 2-4
Make
  touch option *See* -t option
  transformation rules
    table 2-11
  troubleshooting 2-15
  .y suffix 2-11
  .yr suffix 2-11
make command
  arguments 2-5
  syntax 2-5
maketemp function 10-9
Manipulating files
  include function 10-8
  macro processors 10-8
  using m4 to 10-8
Manipulating strings
  m4 macro processor 10-10
Memory
  layout B-1
  requirements B-2
  user B-1, B-3
Memory model 9-3
memory models
  default names 9-6
Memory models

Memory models *(continued)*
integer size 9-6
pointer size 9-6
segment structure 9-8
Messages
386 panic B-8
boot B-1, B-2
error B-1
information B-1
panic B-1, B-5, B-6
system B-1
system traps B-7, B-8
warning B-2, B-3, B-4
Middle model
integer size 9-6
segment structure 9-8
Middle model 7 default names 9-6
Middle model 7 pointer size 9-6

# N

-n option
lint 4-15
Notational conventions 1-4

# P

-p option
lint 4-15
pack option
disable 9-3
Panic messages
386-specific B-8
system traps B-7, B-8, B-1, B-5, B-6
Pipe
SCCS use *See* SCCS
Pointers
size in memory model 9-6
Printing
m4 macro processor 10-11
printing function 10-11
Processes
background 7-12
restoring an execution state 7-11
saving the execution state 7-11
proctl system call B-3
Program
maintainer *See* Make
Program development 1-1

# R

ranlib
description 1-2
Registers
CS 9-8
DS 9-8
segments 9-8
SS 9-8
Relocatable object module 9-3
rm command
SCCS use *See* SCCS
runbig command B-3

# S

XENIX Timesharing system 1-1
SCCS
@(#) string
file information, search 3-36
-a option
login name addition use 3-26
admin command
file administration 3-28
file checking use 3-28
file creation 3-6
use authorization 3-6
administrator
description 3-5
argument
minus sign (-)
use 3-5
types designated 3-4
branch delta
retrieval 3-12
branch number
description 3-3
cdc command
commentary change 3-20
ceiling flag
protection 3-27
checksum
file corruption determination 3-28
command
argument *See* argument
execution control 3-4
explanation 3-30
comments

SCCS *(continued)*
comments *(continued)*
change procedure 3-20
omission, effect 3-31
corrupted file
determination 3-28
processing restrictions 3-28
restoration 3-29
d flag
default specification 3-18
-d flag
flags deletion 3-19
-d option
data specification provision 3-22
flag removal 3-18
data keyword
data specification component 3-23
replacement 3-23
data specification
description 3-23
delta
branch delta *See* branch delta
defined 3-1, 3-2
exclusion 3-32
inclusion 3-32
interference 3-33
latest release retrieval 3-13
level number *See* level number
name *See* SID
printing 3-23, 3-34
range printing 3-24
release number *See* release number
removal 3-35
delta command
comments prompt 3-9
file change procedure 3-9
g-file removal 3-14
p-file reading 3-8, 3-9
delta table
delta removal, effect 3-36
description 3-19
descriptive text
initialization 3-21
modification 3-21
removal 3-22
d-file
temporary g-file 3-4
diagnostic output
-p option effect 3-13
diagnostics
code as help argument 3-14
form 3-14
directory
file argument application 3-4

SCCS *(continued)*
directory *(continued)*
x-file location 3-3
directory use 3-2
-e option
delta range printing 3-24
file editing use 3-8
login name removal 3-27
error message
code use 3-14
form 3-14
exclamation point (!)
MR deletion use 3-21
-f option
flag initialization, modification 3-17
flag, value setting 3-18
file
administration 3-28
change identification 3-33
change, major 3-11
change procedure 3-9
changes *See* delta
checking procedure 3-28
comparison 3-34
composition 3-2, 3-19
corrupted file *See* corrupted file
creation 3-6
data keyword *See* data keyword
descriptive text description 3-19
descriptive text *See* descriptive text
editing, -e option use 3-8
grouping 3-2
identifying information 3-36
link *See* link
multiple concurrent edits 3-24
name arbitrary 3-13
name, s use 3-6
name *See* link
parameter initialization, modificatio
printing 3-22
protection methods 3-25
removal 3-6
retrieval *See* get command
x-file *See* x-file
file argument
description 3-4
processing 3-5
file creation
comment line generation 3-31
commentary 3-31
comments omission, effect 3-31
level number 3-31
release number 3-31
file protection 3-25

SCCS *(continued)*
flags
deletion 3-19
initialization 3-17
modification 3-17
setting, value setting 3-18
use 3-18
floor flag
protection 3-27
-g option
output suppression 3-35
p-file regeneration 3-29
get command
concurrent editing, directory use 3-24
delta inclusion, exclusion check 3-33
-e option use 3-8
file retrieval 3-7
filename creation 3-7
g-file creation 3-3
message 3-7
release number change 3-11
g-file
creation 3-3
creation date, time recordation 3-15
description 3-3
line identification 3-33
line, %M% keyword value 3-34
ownership 3-3
regeneration 3-29
removal, delta command use 3-14
temporary *See* d-file
-h option
file audit use 3-28
help command
argument 3-14
code use 3-14
use 3-30
i flag
file creation, effect 3-16
-i flag
keyword message, error treatment 3-17
-i option
delta inclusion list use 3-32
ID keyword *See* keyword
identification string *See* SID
j flag
multiple concurrent edits
specification 3-24
-k option
g-file regeneration 3-29
keyword
data *See* data keyword
format 3-15
lack, error treatment 3-17

SCCS *(continued)*
keyword *(continued)*
use 3-15
-l option
delta range printing 3-24
l-file creation 3-33
level number
delta component 3-2
new file 3-31
omission, file retrieval, effect 3-10
l-file
contents 3-4
creation 3-33
link
number restriction 3-2
lock file *See* z-file
lock flag
R protection 3-27
%M% keyword
g-file line precedence 3-34
-m option
effective when 3-21
file change identification 3-33
new file creation 3-31
minus sign (-)
argument use 3-5
option argument use 3-4
mode
g-file 3-3
MR
commentary supply 3-19
deletion 3-21
new file creation 3-31
multiple users 3-5
-n option
g-file preservation 3-14
%M% keyword value use 3-34
pipeline use 3-34
option argument
description 3-4
processing order 3-5
output
data specification
*See* data specification
piping 3-31
suppression, -g option 3-35
suppression, -s option 3-31, 3-32
write to standard output 3-13
-p option
delta printing 3-34
output effect 3-13
percentage sign (%)
keyword enclosure 3-15
p-file

SCCS *(continued)*
p-file *(continued)*
contents 3-3, 3-8
creation 3-3
delta command reading 3-9
naming 3-3
ownership 3-3
permissions 3-3
regeneration 3-29
update 3-3
updating 3-4
piping 3-31
-n option use 3-34
prs command
file printing 3-22
purpose 3-1
q file
use 3-4
R
delta removal check 3-35
-r option
delta creation use 3-25
delta printing use 3-23
file retrieval 3-10
release number specification 3-31
release
protection 3-27
release number
change 3-2
change procedure 3-11
delta component 3-2
new file 3-31
-r option, specification 3-31
rm command
file removal 3-6
rmdel command
delta removal 3-35
-s option
output suppression 3-31, 3-32
SID
components 3-2
delta printing use 3-23
sccsdiff command
file comparison 3-34
sequence number
description 3-3
-t option
delta retrieval 3-13
file initialization 3-22
file modification 3-22
tab character
-n option, designation 3-34
user list
empty by default 3-26

SCCS *(continued)*
user list *(continued)*
login name addition 3-26
login name removal 3-27
protection feature 3-25
user name
list 3-25
v flag
new file use 3-18
what command
file information 3-36
write permission
delta removal 3-35
-x option
delta exclusion list use 3-32
XENIX command
use precaution 3-29
x-file
directory, location 3-3
naming procedure 3-3
permissions 3-3
temporary file copy 3-3
use 3-3
-y option
comments prompt response 3-19
new file creation 3-31
-z key
file audit use 3-29
z-file
lock file use 3-4
ownership 3-4
permissions 3-4
SCCS, source code control 1-3
sdb
debugging tools 11-1
description 11-1
displaying data 11-1
displaying instructions 11-1
examining core images 11-1
overview 11-1
quitting 11-1
starting 11-1
stopping 11-1
symbolic names 11-1
using 11-1
Segment
structure
for memory models 9-8
Segments
text data 9-8
setjmp function 7-11
setjmp.h file, described 7-1
shutdown command B-2
signal function 7-1

signal.h file, described 7-1
Signals
  catching 7-4
  default action 7-3
  delaying an action 7-9
  described 7-1
  disabling 7-2
  redefining 7-4
  restoring 7-3, 7-6
  SIG_DFL constant 7-1
  SIGHANG constant 7-2
  SIG_IGN constant 7-1
  SIGINT constant 7-2
  SIGQUIT constant 7-2
  to a child process 7-13
  to background processes 7-12
  with interactive programs 7-10
  with multiple processes 7-12
  with system functions 7-10
sinclude function 10-8
Small model
  integer size 9-6
  segment structure 9-8
Small model 7 default names 9-6
Small model 7 pointer size 9-6
Software development
  described 1-1
Source Code Control System *See* SCCS
SS register 9-8
Stack size 9-2
substr function 10-10
Swap size B-3
syscmd function 10-9
System commands
  m4 macro processor 10-9
System messages
  386-specific B-8
  boot status B-1
  memory layout B-1
System traps B-7, B-8

**T**

Table
  inode B-3
Tags file
  creation 1-3
Text segments 9-8
translit function 10-10

**U**

-u option
  lint 4-4
undefine function 10-5
undivert function 10-8
User memory B-1, B-3

**V**

-v option
  lint 4-4, 4-14

**W**

Warning messages B-1, B-2, B-3

**X**

-x option
  lint 4-4
XENIX file
  identifying information 3-36
xenixconf file
  swap area B-4

**Y**

yacc
  % token keyword
    union member name association 6-32
  0 character
    grammar rules, avoidance 6-5
  %0 keyword
    endmarker token marker 6-11
  accept action *See* parser
  accept simulation 6-30
  action
    0, negative number 6-30
    conflict source 6-17
    defined 6-7

yacc *(continued)*
  action *(continued)*
    error rules 6-24
    form 6-44
    global flag setting 6-29
    input style 6-27
    invocation 6-2
    location 6-8
    nonterminating 6-8
    parser *See* parser
    return value 6-31
    statement 6-7, 6-9
    value in enclosing rules, access 6-30
  ampersand (&)
    bitwise AND operator 6-35
    desk calculator operator 6-35
  arithmetic expression
    desk calculator 6-35
    parsing 6-20
    precedence *See* precedence
  associativity
    arithmetic expression parsing 6-20
    grammar rule association 6-22
    recordation 6-22
    token attachment 6-21
  asterisk (*)
    desk calculator operator 6-35
  backslash (\)
    escape character 6-5
    percentage sign (%) substitution 6-43
  binary operator
    precedence 6-22
  blank character
    restrictions 6-4
  braces ({ })
    action 6-9
    action, dropping 6-44
    action statement enclosure 6-7
    header file enclosure 6-31
  colon (:)
    identifier, effect 6-32
    punctuation 6-5
  comments
    location 6-4
  conflict
    associativity *See* associativity
    disambiguating rules 6-17, 6-18
    message 6-19
    precedence *See* precedence
    reduce/reduce conflict 6-17, 6-23
    resolution, not counted 6-23
    shift/reduce conflict 6-17, 6-19, 6-23
    source 6-17
  declaration

yacc *(continued)*
  declaration *(continued)*
    specification file component 6-4
  declaration section
    header file 6-31
  description 1-2
  desk calculator
    advanced features 6-37
    error recovery 6-38
    floating point interval 6-37
    scalar conversion 6-38
  desk calculator specifications 6-35
  dflag 6-29
  disambiguating rules 6-17, 6-18
  dollar sign ($)
    action significance 6-7
  empty rule 6-28
  enclosing rules, access 6-30
  endmarker
    lookahead token 6-13
    parser input end 6-6
    representation 6-6
    token number 6-11
  environment 6-25
  error
    handling 6-23
    nonassociating implication 6-23
    parser restart 6-23
    simulation 6-30
    yyerrok statement 6-24
  error action *See* parser
  error token
    parser restart 6-23
  escape characters 6-5
  external integer variable 6-26
  flag
    global flag *See* global flag
  floating point intervals *See* desk calcu
  global flag
    lexical analysis 6-29
  grammar rules 6-1, 6-2
    0 character avoidance 6-5
    advanced features 6-37
    ambiguity 6-15
    associativity association 6-22
    C code location 6-44
    empty rule 6-28
    error token 6-23
    format 6-5
    input style 6-27
    left recursion 6-27
    left side repetition 6-5
    names 6-5
    numbers 6-20

yacc *(continued)*
grammar rules 6-1, 6-2 *(continued)*
precedence association 6-22
reduce action 6-12
reduction 6-13
rewrite 6-17
right recursion 6-28
specification file component 6-4
value 6-8
header file, union declaration 6-31
historical features 6-43
identifier
input syntax 6-32
if-else rule 6-18
if-then-else construction 6-18
input
language 6-1
style 6-27
syntax 6-32
input error detection 6-3
keyword 6-21
reservation 6-29
union member name association 6-32
left association 6-16
left associative
reduce implication 6-23
%left keyword 6-21
union member name association 6-32
left recursion 6-27
value type 6-32
%left token
synonym 6-44
lex
interface 5-2
lexical analyzer construction 6-11
lexical analyzer
context dependency 6-28
defined 6-1, 6-9
endmarker return 6-6
floating point constants 6-39
function 6-2
global flag examination 6-29
identifier analysis
lex 6-11
return value 6-31
scope 6-9
specification file component 6-4
terminal symbol *See* terminal symbol
token number agreement 6-9
lexical tie-in 6-28
library 6-26
literal
defined 6-5
delimiting 6-43

yacc *(continued)*
literal *(continued)*
length 6-43
lookahead token 6-11
clearing 6-25
error rules 6-23
LR(2) grammar 6-32
-ly argument, library access 6-26
main program
minus sign (-)
desk calculator operator 6-35
names
composition 6-5
length 6-5
reference 6-4
token name *See* token name
newline character
restrictions 6-4
%nonassoc keyword 6-21
union member name association 6-32
%nonassoc token
synonyms 6-44
nonassociating
error implication 6-23
nonterminal
union member name association 6-32
nonterminal name
input style 6-27
representation 6-5
nonterminal symbol 6-2
empty string match 6-6
location 6-6
name *See* nonterminal name
start symbol *See* start symbol
octal integer
0 beginning 6-35
parser
accept action 6-13
accept simulation 6-30
actions 6-11
arithmetic expression 6-20
conflict *See* conflict
creation 6-20
defined 6-1
description 6-11
error action 6-13
error handling *See* error
goto action 6-12
initial state 6-15
input end 6-6
lookahead token 6-11
movement 6-11
names, yy prefix 6-9
nonterminal symbol *See* nonterminal

yacc *(continued)*
  parser *(continued)*
    production failure 6-3
    reduce action 6-12
    restart 6-23
    shift action 6-11
    start symbol recognition 6-6
    token number agreement 6-9
  percentage sign (%)
    action 6-9
    desk calculator mod operator 6-35
    header file enclosure 6-31
    precedence keyword 6-21
    specification file section separator 6-4
    substitution 6-43
  pipe symbol (|)
    grammar rule repetition 6-5
  plus sign (+)
    desk calculator operator 6-35
  %prec
    synonym 6-44
  %prec keyword 6-22
  precedence
    binary operator 6-22
    change 6-22
    grammar rule association 6-22
    keyword 6-21
    parsing function 6-20
    recordation 6-22
    token attachment 6-21
    unary operator 6-22
  program
    specification file component 6-4
  punctuation 6-5
  quotation marks, double ( 6-43
  quotation marks, single ('')
    literal enclosure 6-5
  reduce action *See* parser
  reduce command
    number reference 6-20
  reduce/reduce conflict 6-17, 6-23
  reduction conflict *See* reduce/reduce conflict
  reduction conflict *See* shift/reduce conflict
  reserved words 6-29
  right association 6-16
  right associative
    shift implication 6-23
  %right keyword 6-21
    union member name association 6-32
  right recursion 6-28
  %right token
    synonym 6-44
  semicolon (;)
    input style 6-27

yacc *(continued)*
  semicolon (;) *(continued)*
    punctuation 6-5
  shift action *See* parser
  shift command
    number reference 6-20
  shift/reduce conflict 6-17, 6-19, 6-23
  simple-if rule 6-18
  slash (/)
    desk calculator operator 6-35
  specification file
    contents 6-4
    lexical analyzer inclusion 6-4
    sections separator 6-4
  specification files 6-3
  start symbol
    description 6-6
    location 6-6
  symbol synonyms 6-44
  tab character
    restrictions 6-4
  terminal symbol 6-2
  token
    associativity 6-21
    defined 6-1
    error token *See* error token
    names 6-4
    organization 6-1
    precedence 6-21
  %token
    synonym 6-44
  token names 6-10
    declaration 6-6
    input style 6-27
  token number 6-9
    agreement 6-9
    assignment 6-10
    endmarker 6-11
  %type keyword 6-32
  unary operator
    precedence 6-22
  underscore sign (_)
    parser 6-14
  union
    copy 6-31
    declaration 6-31
    header file 6-31
    name association 6-31
  unreachable statement 4-6
  -v option
    y.output file 6-13
  value
    typing 6-31
    union *See* union

yacc *(continued)*
value stack 6-31
declaration 6-31
floating point scalars, integers 6-38
vertical bar (|)
bitwise OR operator 6-35
desk calculator operator 6-35
input style 6-27
y.output file 6-14
parser checkup 6-23
y.tab.c file 6-25
y.tab.h file 6-31
YYACCEPT 6-30
yychar 6-26
yyclearin statement 6-25
yydebug 6-26
yyerrok statement 6-24
yyerror 6-26
YYERROR 6-38
yylex 6-25
yyparse 6-25
YYACCEPT effect 6-30
YYSTYPE 6-31

Replace this Page
with Tab Marked:

# MASM
# User's Guide

# XENIX® System V

## Development System

## Macro Assembler User's Guide

Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc. nor Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

Portions © 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988 Microsoft Corporation.
All rights reserved.
Portions © 1983, 1984, 1985, 1986, 1987, 1988 The Santa Cruz Operation, Inc.
All rights reserved.

ALL USE, DUPLICATION, OR DISCLOSURE WHATSOEVER BY THE GOVERNMENT SHALL BE EXPRESSLY SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBDIVISION (b) (3) (ii) FOR RESTRICTED RIGHTS IN COMPUTER SOFTWARE AND SUBDIVISION (b) (2) FOR LIMITED RIGHTS IN TECHNICAL DATA, BOTH AS SET FORTH IN FAR 52.227-7013.

Microsoft, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation.

SCO Document Number: XG-10-31-88-5.0/2.3

# Contents


1 **Getting Started**

1.1 Introduction 1-1
1.2 System Considerations 1-1
1.3 The Program-Development Cycle 1-1
1.4 Developing Programs 1-4

2 **Using masm**

2.1 Introduction 2-1
2.2 Running the Assembler 2-1
2.3 Using masm Options 2-2
2.4 Reading Assembly Listings 2-15

3 **Writing Source Code**

3.1 Introduction 3-1
3.2 Writing Assembly-Language Statements 3-1
3.3 Assigning Names to Symbols 3-4
3.4 Constants 3-6
3.5 Defining Default Assembly Behavior 3-11
3.6 Ending a Source File 3-15

4 **Defining Segment Structure**

4.1 Introduction 4-1
4.2 Simplified Segment Definitions 4-1
4.3 Full Segment Definitions 4-15
4.4 Defining Segment Groups 4-26
4.5 Associating Segments with Registers 4-28
4.6 Initializing Segment Registers 4-31
4.7 Nesting Segments 4-35

5 **Defining Labels and Variables**

5.1 Introduction 5-1
5.2 Using Type Specifiers 5-1
5.3 Defining Code Labels 5-2
5.4 Defining and Initializing Data 5-5
5.5 Setting the Location Counter 5-20
5.6 Aligning Data 5-21

**6 Using Structures and Records**

6.1 Introduction 6-1
6.2 Structures 6-1
6.3 Records 6-6

**7 Creating Programs from Multiple Modules**

7.1 Introduction 7-1
7.2 Declaring Symbols Public 7-1
7.3 Declaring Symbols External 7-3
7.4 Using Multiple Modules 7-6
7.5 Declaring Symbols Communal 7-8

**8 Using Operands and Expressions**

8.1 Introduction 8-1
8.2 Using Operands with Directives 8-1
8.3 Using Operators 8-2
8.4 Using the Location Counter 8-20
8.5 Using Forward References 8-21
8.6 Strong Typing for Memory Operands 8-25

**9 Assembling Conditionally**

9.1 Introduction 9-1
9.2 Using Conditional-Assembly Directives 9-1
9.3 Using Conditional-Error Directives 9-6

**10 Using Equates, Macros, and Repeat Blocks**

10.1 Introduction 10-1
10.2 Using Equates 10-1
10.3 Using Macros 10-5
10.4 Defining Repeat Blocks 10-11
10.5 Using Macro Operators 10-15
10.6 Using Recursive, Nested, and Redefined Macros 10-21
10.7 Managing Macros and Equates 10-25

**11 Controlling Assembly Output**

11.1 Introduction 11-1
11.2 Sending Messages to Standard Output 11-1
11.3 Controlling Page Format in Listings 11-2
11.4 Controlling the Contents of Listings 11-5
11.5 Controlling Cross-Reference Output 11-9

**12 Understanding 8086-Family Processors**

12.1 Introduction 12-1
12.2 Using the 8086-Family Processors 12-1
12.3 Segmented Addresses 12-4
12.4 Using 8086-Family Registers 12-5
12.5 Using the 80386 Processor 12-13

**13 Using Addressing Modes**

13.1 Introduction 13-1
13.2 Using Immediate Operands 13-1
13.3 Using Register Operands 13-2
13.4 Using Memory Operands 13-4

**14 Loading, Storing, and Moving Data**

14.1 Introduction 14-1
14.2 Transferring Data 14-1
14.3 Converting between Data Sizes 14-4
14.4 Loading Pointers 14-7
14.5 Transferring Data to and from the Stack 14-10
14.6 Transferring Data to and from Ports 14-15

**15 Doing Arithmetic and Bit Manipulations**

15.1 Introduction 15-1
15.2 Adding 15-1
15.3 Subtracting 15-3
15.4 Multiplying 15-6
15.5 Dividing 15-9
15.6 Calculating with Binary Coded Decimals 15-10
15.7 Doing Logical Bit Manipulations 15-14
15.8 Scanning for Set Bits 15-19
15.9 Shifting and Rotating Bits 15-20

**16 Controlling Program Flow**

16.1 Introduction 16-1
16.2 Jumping 16-1
16.3 Looping 16-12
16.4 Setting Bytes Conditionally 16-15
16.5 Using Procedures 16-16
16.6 Using Interrupts 16-25
16.7 Checking Memory Ranges 16-27

**17 Processing Strings**

17.1 Introduction 17-1
17.2 Setting Up String Operations 17-1
17.3 Moving Strings 17-5
17.4 Searching Strings 17-7
17.5 Comparing Strings 17-8
17.6 Filling Strings 17-10
17.7 Loading Values from Strings 17-11
17.8 Transferring Strings to and from Ports 17-12

**18 Calculating with a Math Coprocessor**

18.1 Introduction 18-1
18.2 Coprocessor Architecture 18-1
18.3 Emulation 18-4
18.4 Using Coprocessor Instructions 18-4
18.5 Coordinating Memory Access 18-9
18.6 Transferring Data 18-11
18.7 Doing Arithmetic Calculations 18-17
18.8 Controlling Program Flow 18-24
18.9 Using Transcendental Instructions 18-28
18.10 Controlling the Coprocessor 18-30

**19 Controlling the Processor**

19.1 Introduction 19-1
19.2 Controlling Timing and Alignment 19-1
19.3 Controlling the Processor 19-1
19.4 Controlling Protected-Mode Processes 19-2
19.5 Controlling the 80386 19-4

**A New Features**

A.1 Introduction A-1
A.2 Enhancements to masm A-1
A.3 Compatibility with Assemblers and Compilers A-5

**B Instruction Summary**

B.1 Introduction B-1
B.2 8086 Instruction Mnemonics B-2
B.3 8087 Instruction Mnemonics B-8
B.4 80186 Instruction Mnemonics B-13
B.5 80286 Nonprotected Instruction Mnemonics B-14
B.6 80286 Protected Instruction Mnemonics B-15
B.7 80287 Instruction Mnemonics B-15
B.8 80386 Nonprotected Instruction Mnemonics B-16
B.9 80386 Protected Instruction Mnemonics B-19
B.10 80387 Instruction Mnemonics B-20

**C Directive Summary**

C.1 Introduction C-1

**D Segment Names for High-Level Languages**

D.1 Introduction D-1
D.2 Text Segments D-2
D.3 Near Data Segments D-3
D.4 Far Data Segments D-4
D.5 BSS Segments D-5
D.6 Constant Segments D-6

**E Error Messages and Exit Codes**

E.1 Introduction E-1
E.2 Messages and Exit Codes from masm E-1

# Part 1

# Using Assembler Programs

Part 1 of the Macro Assembler User'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

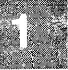
1.1 Introduction 1-1

1.2 System Considerations 1-1

1.3 The Program-Development Cycle 1-1

1.4 Developing Programs 1-4
  1.4.1 Writing and Editing Assembly-Language Programs 1-4
  1.4.2 Assembling Source Files 1-5

## 1.1 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.

## 1.2 System Considerations

Before you start developing assembly-language programs, you need to verify that:

- The current operating system is XENIX System V/286 or System V/386.

  If the current operating system is not XENIX System V/286 or System V/386, 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.

## 1.3 The Program-Development Cycle

The program-development cycle for assembly language is illustrated in Figure 1.1.

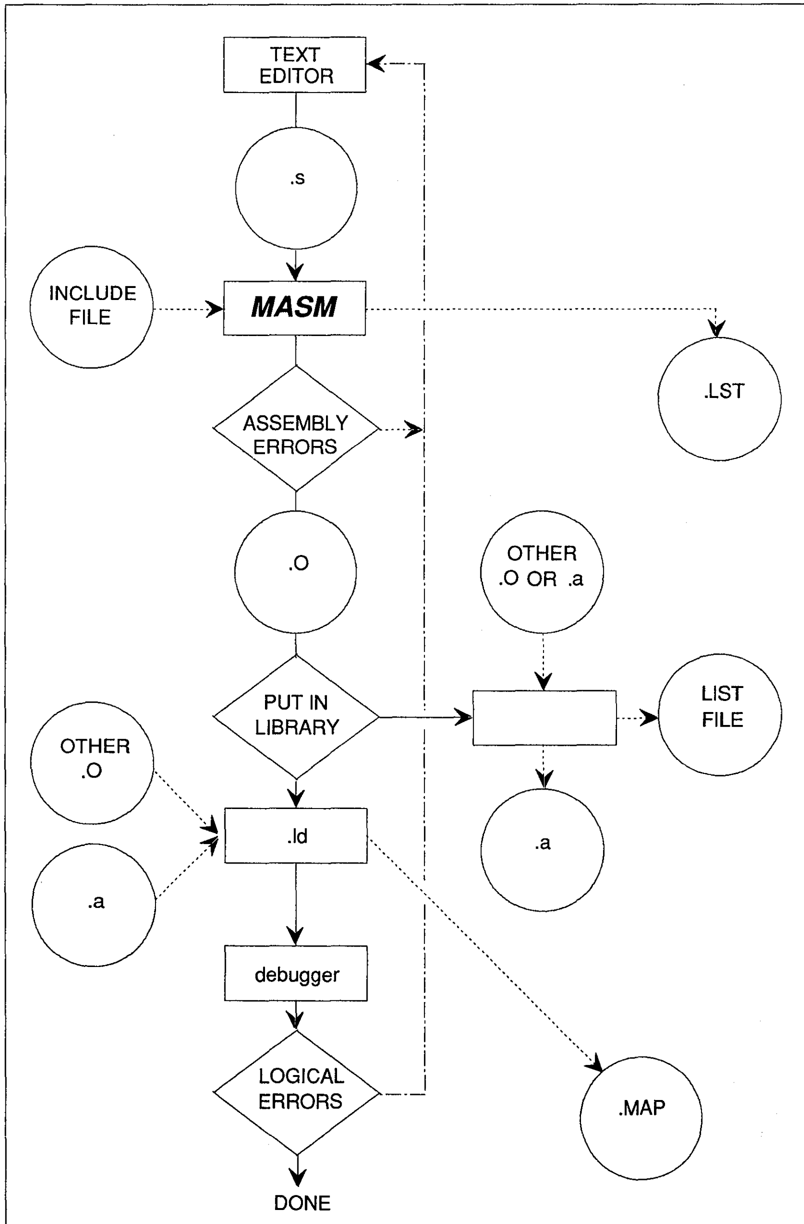



**Figure 1-1** The Program Development Cycle

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. **ld** 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 XENIX debugger, such as **adb**

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

All or part of the program-development cycle can be automated by using **make** with make description files. **make** is most useful for developing complex programs involving numerous source modules.

## 1.4 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.

### 1.4.1 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.

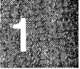
To create assembly-language source files, you need a text editor that is capable of producing ASCII files.

The following example illustrates source code that produces a stand-alone executable program.

**Example 1**

```
        .286
        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
```

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.

### 1.4.2 Assembling Source Files

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.

# Chapter 2

# Using masm


2.1 Introduction 2-1

2.2 Running the Assembler 2-1
  2.2.1 Assembly Using the Command Line 2-1

2.3 Using masm Options 2-2
  2.3.1 Specifying the Segment-Order Method 2-3
  2.3.2 Setting the File-Buffer Size 2-4
  2.3.3 Creating a Pass 1 Listing 2-5
  2.3.4 Defining Assembler Symbols 2-5
  2.3.5 Creating Code for a Floating-Point Emulator 2-6
  2.3.6 Getting Command-Line Help 2-7
  2.3.7 Setting a Search Path for Include Files 2-8
  2.3.8 Specifying Listing Files 2-9
  2.3.9 Specifying Case Sensitivity 2-9
  2.3.10 Suppressing Tables in the Listing File 2-10
  2.3.11 Checking for Impure Code 2-10
  2.3.12 Controlling Display of Assembly Statistics 2-11
  2.3.13 Setting the Warning Level 2-12
  2.3.14 Listing False Conditionals 2-13
  2.3.15 Displaying Error Lines on Standard Error 2-14
  2.3.16 Writing Symbolic Information to the Object File 2-14

2.4 Reading Assembly Listings 2-15
  2.4.1 Reading Code in a Listing 2-15
  2.4.2 Reading a Macro Table 2-18
  2.4.3 Reading a Structure and Record Table 2-18
  2.4.4 Reading a Segment and Group Table 2-20
  2.4.5 Reading a Symbol Table 2-21
  2.4.6 Reading Assembly Statistics 2-23
  2.4.7 Reading a Pass 1 Listing 2-23

## 2.1 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.

## 2.2 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.

### 2.2.1 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 entitled, ''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.

## 2.3 Using masm Options

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

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 "Defining Default Assembly Behavior."

---

The following sections describe each of the **masm** options in more detail.

### 2.3.1 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 "Setting the Segment-Order Method," and "Defining Segment Combinations with Combine Type."

**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.

### 2.3.2 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.

### 2.3.3 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 ''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.

### 2.3.4 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 "Controlling Page Format in Listings").

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
```

## 2.3.5 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.

### 2.3.6 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
```

### 2.3.7 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 "Using Include Files."

### 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:

```
INCLUDE /u/me/macro/asm.inc
INCLUDE ../asm.inc
INCLUDE ./asm.inc
```

### 2.3.8 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 "Reading Assembly Listings."

### 2.3.9 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 "Creating Programs from Multiple Modules").

If you use the **-Zi** or **-Zd** option (see "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.

### 2.3.10 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
```

### 2.3.11 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.''

**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.

### 2.3.12 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.

### 2.3.13 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.

**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.

### 2.3.14 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 "Controlling Listing of Conditional Blocks." 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.

## 2.3.15 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.

## 2.3.16 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.

## 2.4 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.

### 2.4.1 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

                                          INCLUDEasm.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
```

## 2.4.2 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
```

## 2.4.3 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.

2

## 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.

### 2.4.4 Reading a Segment and Group Table

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

**Example**

```
Segments and Groups:

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

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

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

The "Class" column lists the class name of the segment. For a complete explanation of the align, combine, and class types, see "Defining Full Segments."

## 2.4.5 Reading a Symbol Table

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

### Example

```
Symbols:

                Name                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 DWORD0001      _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 "Understanding Memory Models").

### 2.4.6 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
```

### 2.4.7 Reading a Pass 1 Listing

When you specify the **-d** option in the **masm** command line, the assembler puts a Pass 1 listing in the assembly-listing file. The listing file 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


3.1 Introduction 3-1

3.2 Writing Assembly-Language Statements 3-1
- 3.2.1 Using Mnemonics and Operands 3-3
- 3.2.2 Writing Comments 3-3

3.3 Assigning Names to Symbols 3-4

3.4 Constants 3-6
- 3.4.1 Integer Constants 3-6
- 3.4.2 Packed Binary Coded Decimal Constants 3-9
- 3.4.3 Real-Number Constants 3-9
- 3.4.4 String Constants 3-10

3.5 Defining Default Assembly Behavior 3-11

3.6 Ending a Source File 3-15

## 3.1 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 with **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.



## 3.2 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:

```
       .286
       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
```

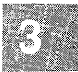
However, the code is much easier to interpret if each field is assigned a specified tab position and a standard convention is used for capitalization. The example program in Chapter 1, "Getting Started," is the same as the example above except for the conventions used.

### 3.2.1 Using Mnemonics and Operands

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

Directives 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. Sections 3.2 and 3.3 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."

### 3.2.2 Writing Comments

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

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

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

**Syntax**

> **COMMENT** *delimiter* [*text*]
> *text*
> *delimiter* [*text*]

All *text* between the first *delimiter* and the line containing a second *delimiter* is ignored by the assembler. The *delimiter* character is the first 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)
```

## 3.3 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 Section 2.2.9, ‘‘Specifying Case Sensitivity.’’

- 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** |

| | | | | |
|---|---|---|---|---|
| **.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 called "Defining Default Assembly Behavior," describes processor directives. Instruction mnemonics for each processor are listed in Appendix B, "Instruction Summary." Register names are listed in "Using Register Operands."

## 3.4 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.

### 3.4.1 Integer Constants

Integer constants represent integer values. They can be used in a variety of contexts in assembly-language source code. For example, they can be used in data declarations 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 "Packed Binary Coded Decimal Constants," and "Binary Coded Decimal Variables."

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 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
```

### 3.4.2 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 ''Binary Coded Decimal Variables,'' for complete information on storage of binary coded decimals.



**Examples**

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

### 3.4.3 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 ''Real-Number Variables,'' 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 "Real-Number Variables," for a description of these formats.

---

## 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
```

## 3.4.4 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.
```

## 3.5 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:

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 "Defining the Memory Model."

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. "Real-Number Variables," 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.<br><br>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 "Controlling Protected- Mode Processes") 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 "Coordinating Memory Access," 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 "Using Predefined Equates," 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
```



## 3.6 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 "Initializing the CS and IP Registers."

# Chapter 4

# Defining Segment Structure

4.1 Introduction 4-1

4.2 Simplified Segment Definitions 4-1
- 4.2.1 Understanding Memory Models 4-2
- 4.2.2 Specifying MS-DOS Segment Order 4-3
- 4.2.3 Defining the Memory Model 4-5
- 4.2.4 Defining Simplified Segments 4-6
- 4.2.5 Using Predefined Equates 4-9
- 4.2.6 Simplified Segment Defaults 4-11
- 4.2.7 Default Segment Names 4-12

4.3 Full Segment Definitions 4-15
- 4.3.1 Setting the Segment-Order Method 4-15
- 4.3.2 Defining Full Segments 4-16

4.4 Defining Segment Groups 4-26

4.5 Associating Segments with Registers 4-28

4.6 Initializing Segment Registers 4-31
- 4.6.1 Initializing the CS and IP Registers 4-31
- 4.6.2 Initializing the DS Register 4-32
- 4.6.3 Initializing the SS and SP Registers 4-33
- 4.6.4 Initializing the ES Register 4-34

4.7 Nesting Segments 4-35

## 4.1 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.

## 4.2 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 Sections 4.3 and 4.4. 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.

### 4.2.1 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.

### 4.2.2 Specifying MS-DOS Segment Order

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 "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.

### 4.2.3 Defining the Memory Model

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

4

**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 ``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 ``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.

## 4.2.4 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 "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.

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 "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 "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 "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 "Default Segment Names," for an equivalent version using full segment definitions.

### 4.2.5 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
```

*@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.

---

4

### 4.2.6 Simplified Segment Defaults

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

- 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 "Procedure Labels," 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 "Defining Segment Groups."

### 4.2.7 Default Segment Names

If you use the simplified segment directives by themselves, you do not need to know the names assigned for each segment. However, it is possible to mix full segment definitions with simplified segment definitions. Therefore, 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 |

| Model | Directive | Name | Align | Combine | Class | Group |
|---|---|---|---|---|---|---|
| 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.**

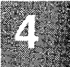
## Example 1

```
            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 ''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 ‘‘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.

## 4.3 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.

### 4.3.1 Setting the Segment-Order Method

The order in which **masm** writes segments to the object file can be either sequential or alphabetical. If the sequential method is specified, segments 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.

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 **DOS-SEG** directive (see "Specifying MS-DOS Segment Order") and class type (see "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 "Specifying the Segment-Order Method"). 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.

### 4.3.2 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 "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 "Defining Segment Combinations with Combine Type."

### 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 "80386 Indirect Memory Operands." 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 Section 4.5.3, ''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. Section 5.4, "Setting the Location Counter," 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.



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
```

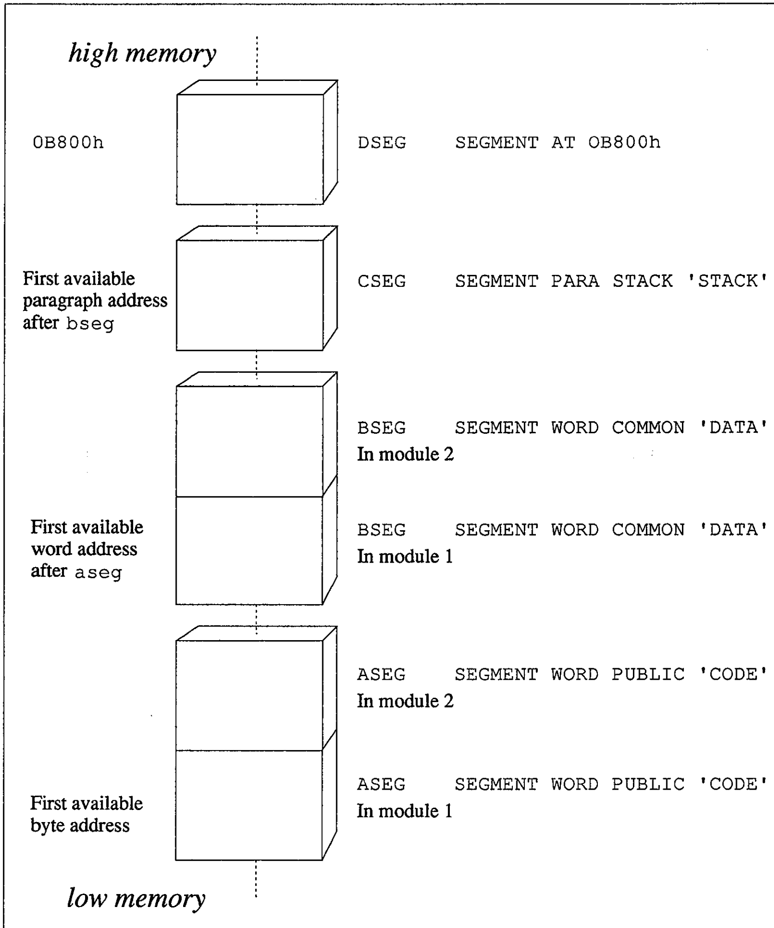



**Figure 4-1** Segment Structure with Combine and Align Types

## 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
```

## 4.4 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 ''SEG Operator'').

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.

---

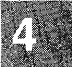
*Note*

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 "Associating Segments with Registers") and as an operand prefix with the segment-override operator (discussed in "Segment-Override Operator").

**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.

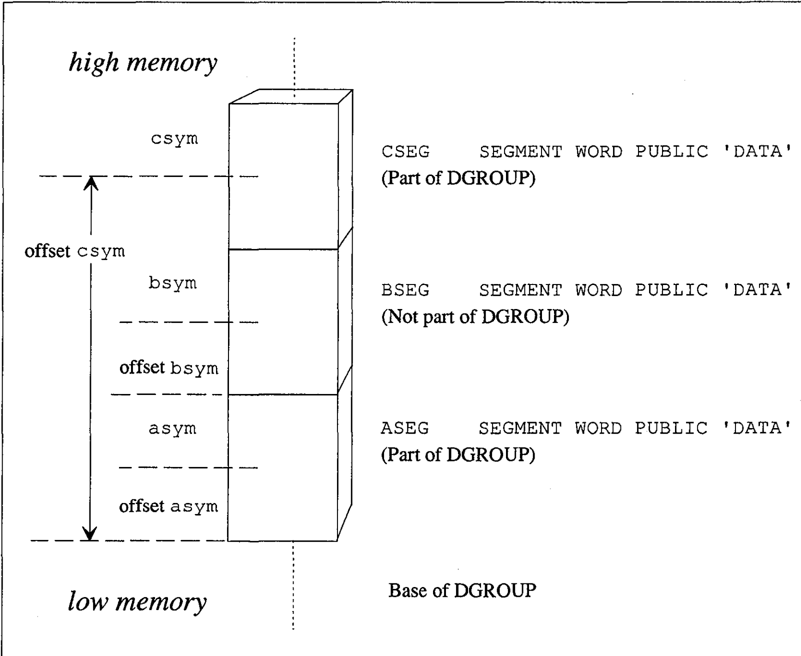



**Figure 4-2** Segment Structure with Groups

## 4.5 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 ''Initializing Segment Registers.''

---

**Syntax**

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

4

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 ''SEG Operator'')
- 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 "Segment-Override Operator"). 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
```

## 4.6 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.

### 4.6.1 Initializing the CS and IP Registers

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



#### Syntax

**END** [*startaddress*]

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

The **CS** segment is initialized to the value of *startaddress*. The **IP** register is normally initialized to 0. You can change the initial value of the **IP** register by using the **ORG** directive (see "Setting the Location Counter") 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.

### 4.6.2 Initializing the DS Register

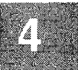
The **DS** register must be initialized to the address of the segment that will be used for data.

The address of the segment or group 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
```

### 4.6.3 Initializing the SS and SP Registers

The **SS** register is automatically initialized to the value of the last segment in the source code having combine type **STACK.** The **SP** register is automatically initialized to the 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**.

### 4.6.4 Initializing the ES Register

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

**Example**

```
ASSUME   es:@fardata        ; Tell the assembler
mov      ax,@fardata        ; Tell the processor
mov      es,ax
```

## 4.7 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.

### 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   segmentword 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   segmentbyte 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**

```
; 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


5.1 Introduction 5-1

5.2 Using Type Specifiers 5-1

5.3 Defining Code Labels 5-2
- 5.3.1 Near Code Labels 5-2
- 5.3.2 Procedure Labels 5-3
- 5.3.3 Code Labels Defined with the LABEL Directive 5-5

5.4 Defining and Initializing Data 5-5
- 5.4.1 Variables 5-5
- 5.4.2 Arrays and Buffers 5-18
- 5.4.3 Labeling Variables 5-20

5.5 Setting the Location Counter 5-20

5.6 Aligning Data 5-21

## 5.1 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.

## 5.2 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 |

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 Section 4.1, "Simplified Segment Definitions") 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**.

## 5.3 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.

### 5.3.1 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*:

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 ''Associating Segments with Registers'').

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.''

### Examples

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

5

## 5.3.2 Procedure Labels

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

### Syntax

*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 ''Using Procedures.''

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.''

### Examples

```
            call    task        ; Call procedure
            .
            .
            .
task        PROC    NEAR        ; Start of procedure
            .
            .
            .
            ret
task        ENDP                ; End of procedure
```

### 5.3.3 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 "Defining Procedures").

#### Example

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

5

## 5.4 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.

### 5.4.1 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 "Arrays and Buffers."

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

### Integer Variables

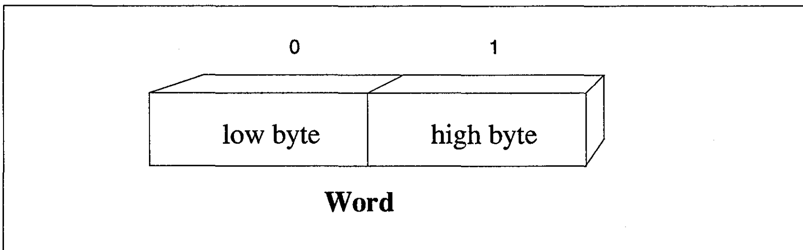
When defining an integer variable, you can specify an initial value as an integer constant or as a constant expression. If you specify an initial value too large for the specified variable, **masm** generates an error.

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:

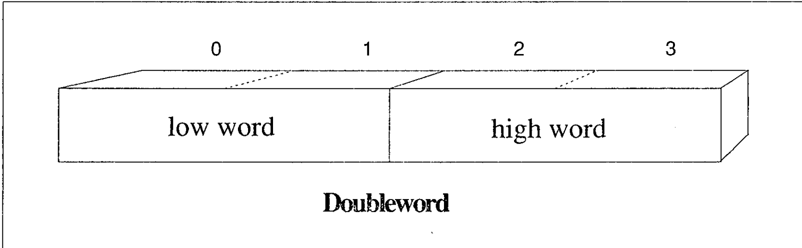
| Directive | Size |
|---|---|
| **DB** (bytes) | Allocates unsigned numbers from 0 to 255 or signed numbers from -128 to 127.<br><br>These values can be used directly in 8086-family instructions. |
| **DW** (words) | Allocates unsigned numbers from 0 to 65,535 or signed numbers from -32,768 to 32,767. The bytes of a word integer are stored in the following format: |




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.

**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:

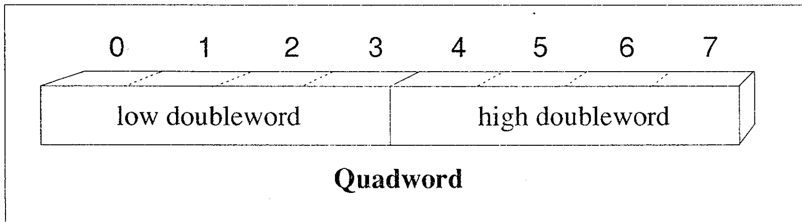


**Doubleword**

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 ''Adding''). 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

**DQ** (quadwords) Allocates 64-bit integers. The doublewords of a quadword integer are stored in the following format:



**Quadword**

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 "Adding").

**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 "Binary Coded Decimal Variables." 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:

```
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 later.

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.

### 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

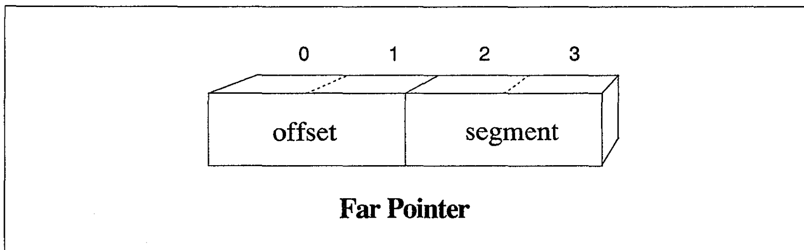
```
function9   DB    'Hello',10,'$'

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:



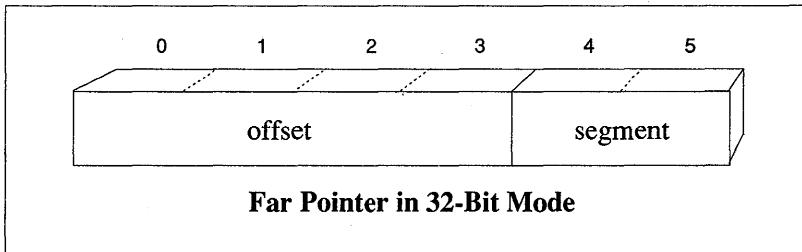

**Far Pointer**

### Examples

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

### 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."

5

## 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
```

The real-number designator (**R**) used to specify encoded numbers is explained in Section 3.3.3, ‘‘Real-Number Constants.’’

### 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 XENIX 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.

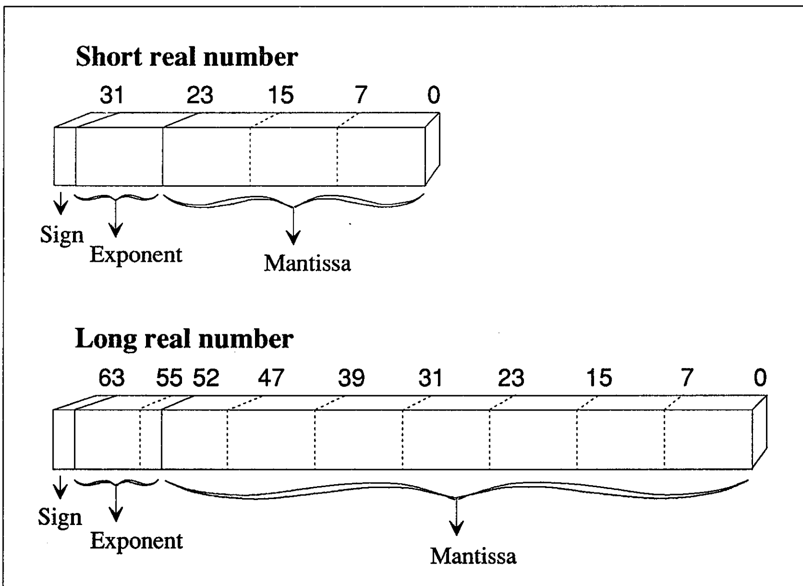



**Figure 5-1** Encoding for Real Numbers in IEEE Format

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.

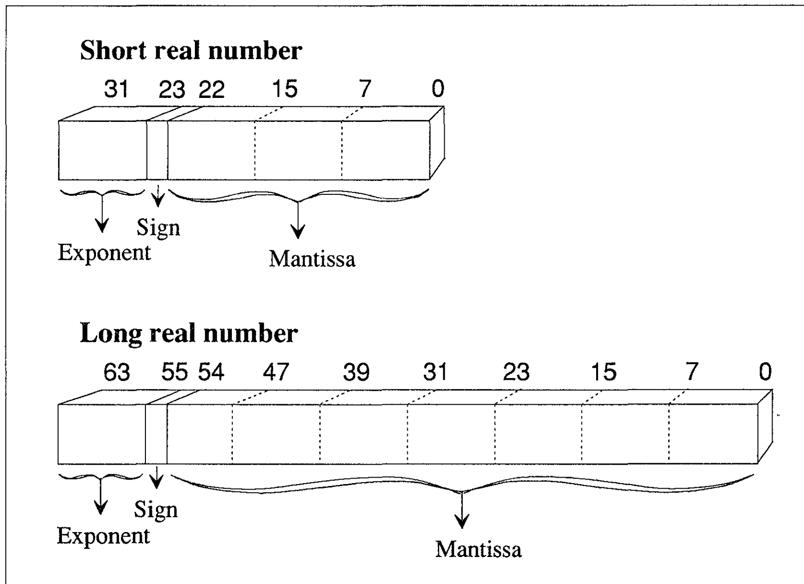



**Figure 5-2** Encoding for Real Numbers in Microsoft Binary Format

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.

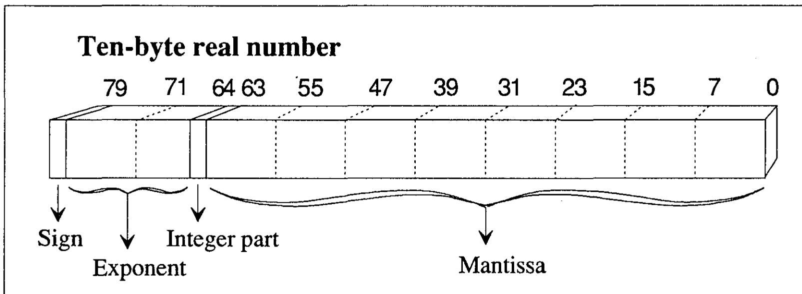



**Figure 5-3** Encoding for Real Numbers in Temporary-Real Format

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.

### 5.4.2 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.

**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.

---

### 5.4.3 Labeling Variables

The **LABEL** directive can be used to define a variable of a given size at a specified location. It is useful if you want to refer to the same data as variables of different sizes.

**Syntax**

*name* **LABEL** *type*

The *name* is the symbol assigned to the variable, and *type* is the variable size. The type can be any one of the following type specifiers: **BYTE**, **WORD**, **DWORD**, **FWORD**, **QWORD**, or **TBYTE**. It can also be the name of a previously defined structure.

**Examples**

```
warray      LABEL   WORD        ; Access array as 50 words
darray      LABEL   DWORD       ; Access same array as 25 doublewords
barray      DB      100 DUP(?)  ; Access same array as 100 bytes
```

## 5.5 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*

The value of the location counter, represented by the dollar sign ($), can be used in *expression*, as described in Section 8.3, ''Using the Location Counter.''

---

### Example 1

```
; 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
```

Example 1 illustrates one way of assigning symbolic names to absolute addresses. This technique is not possible under protected-mode operating systems.

## 5.6 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.

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

6.1 Introduction 6-1

6.2 Structures 6-1
- 6.2.1 Declaring Structure Types 6-2
- 6.2.2 Defining Structure Variables 6-3
- 6.2.3 Using Structure Operands 6-5

6.3 Records 6-6
- 6.3.1 Declaring Record Types 6-6
- 6.3.2 Defining Record Variables 6-8
- 6.3.3 Using Record Operands and Record Variables 6-10
- 6.3.4 Record Operators 6-11
- 6.3.5 Using Record-Field Operands 6-13

## 6.1 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.

## 6.2 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.

### 6.2.1 Declaring Structure Types

The **STRUC** and **ENDS** directives mark the beginning and end of a type declaration for a structure.

**Syntax**

> *name* **STRUC**
> *fielddeclarations*
> *name* **ENDS**

The *name* declares the name of the structure type. It must be unique. The *fielddeclarations* declare the fields of the structure. Any number of field declarations may be given. They must follow the form of data definitions described in "Defining and Initializing Data." 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.

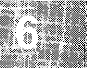
## 6.2.2 Defining Structure Variables

A structure variable is a variable with one or more fields of different sizes. The sizes and initial values of the fields are determined by the structure type with which the variable is defined.

### Syntax

[*name*] *structurename* <[*initialvalue* [,*initialvalue*...]]>

The *name* is the name assigned to the variable. If no *name* is given, the assembler allocates space for the variable, but does not give it a symbolic name. The *structurename* is the name of a structure type previously declared 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 *initialvalues* are given for more than one field, the values must be separated by commas.

If the **DUP** operator (see ''Arrays and Buffers'') 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 first example in "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.

---

### 6.2.3 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 "Structure-Field-Name Operator."

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
            .
            .
            .
```

6

## 6.3 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).

### 6.3.1 Declaring Record Types

The **RECORD** directive declares a record type for an 8- or 16-bit record that contains one or more bit fields. With the 80386, 32-bit records can also be declared.

**Syntax**

*recordname* **RECORD** *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.

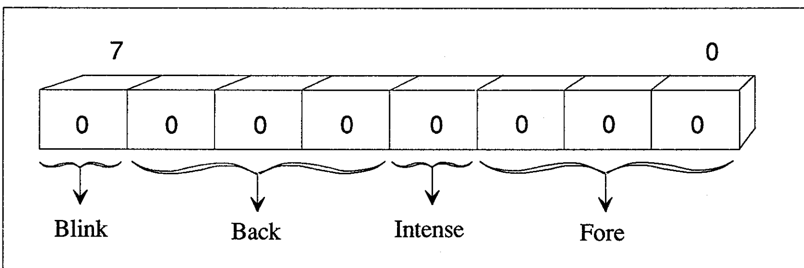
In all cases, the first field you declare goes into the most significant bits of the record. Successively declared fields are placed in the succeeding bits to the right. If the fields you declare do not total exactly 8 bits or exactly 16 bits, the entire record is shifted right so that the last bit of the last field is the lowest bit of the record. Unused bits in the high end of the record are initialized to 0.

6

## 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:

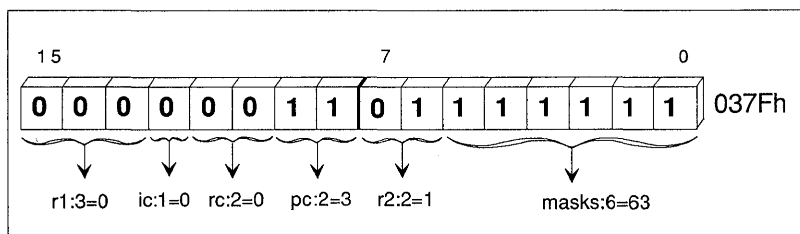


Since no initial values are given, all bits are set to 0. Note that this is only a template maintained by the assembler. No data are created.

### 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.

## 6.3.2 Defining Record Variables

A record variable is an 8-bit or 16-bit variable whose bits are divided into one or more fields. With the 80386, 32-bit variables are also allowed.

### Syntax

[*name*] *recordname* <[*initialvalue* [,*initialvalue*]...]>

The *name* is the symbolic name of the variable. If no *name* is given, the assembler allocates space for the variable, but does not give it a symbolic name. The *recordname* is the name of a record type that was previously declared 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 ''Arrays and Buffers'') 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>)
```

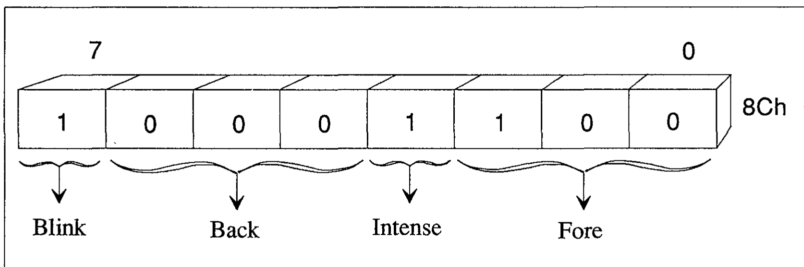
You do not have to initialize all fields in a record. If an initial value is left blank, the assembler automatically uses the default initial value of the field. This is declared by the record type. If there is no default value, each bit in the field is cleared.

"Using Record Operands and Record Variables," and "Record Operators," illustrate ways to use record data after it has been declared.

### Examples

```
color   RECORD  blink:1,back:3,intense:1,fore:3 ; Record declaration
warning color   <1,0,1,4>                       ; Record definition
```

The definition above creates a variable named *warning* whose type is given by the record type *color*. The initial values of the fields in the variable are set to the values given in the record definition. 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

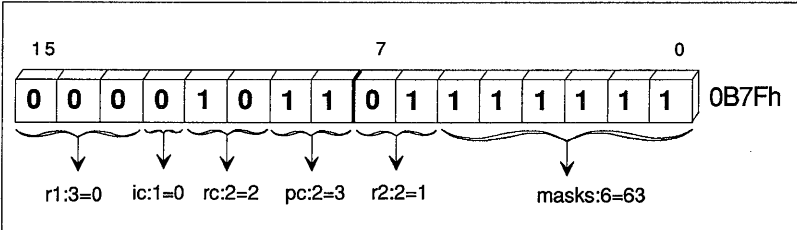
```
color  RECORD  blink:1,back:3,intense:1,fore:3 ; Record declaration
colors color   16 DUP (<>)                      ; Record declaration
```

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:




## 6.3.3 Using Record Operands and Record Variables

A record operand refers to the value of a record type. It should not be confused with a record variable. A record operand is a constant; a record variable is a value stored in memory. A record operand can be used with the following syntax:

### Syntax

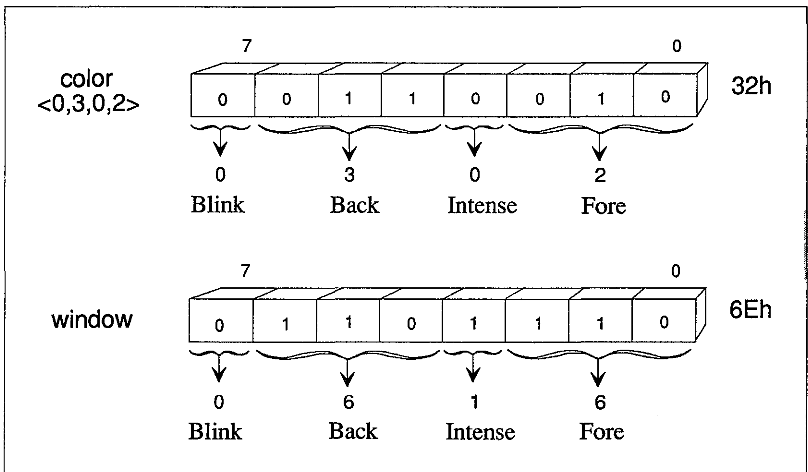
*recordname* <[[*value*][,*value*]...]>

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:




### 6.3.4 Record Operators

The **WIDTH** and **MASK** operators are used exclusively with records to return constant values representing different aspects of previously declared records.

## 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.

**Examples**

```
            .DATA
color       RECORD  blink:1,back:3,intense:1,fore:3

wblink      EQU     WIDTH blink   ; "wblink"   = 1   "blink"   = 7
wback       EQU     WIDTH back    ; "wback"    = 3   "back"    = 4
wintense    EQU     WIDTH intense ; "wintense" = 1   "intense" = 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
```

### 6.3.5 Using Record-Field Operands

Record-field operands represent the location of a field in its corresponding record. The operand evaluates to the bit position of the low-order bit in the field and can be used as a constant operand. The field name must be from a previously declared record.

Record-field operands are often used with the **WIDTH** and **MASK** operators, as described in "The MASK Operator," and "The WIDTH Operator."

## 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.

# Chapter 7

# Creating Programs from Multiple Modules


7.1 Introduction 7-1

7.2 Declaring Symbols Public 7-1

7.3 Declaring Symbols External 7-3

7.4 Using Multiple Modules 7-6

7.5 Declaring Symbols Communal 7-8

## 7.1 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 "Using Local Variables."

---

## 7.2 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.

---

*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
```

## 7.3 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 ''Using Equates'').

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. ''Defining the Memory Model,'' 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
```

Example 2 shows a fragment similar to the one in Example 2, 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.

## 7.4 Using Multiple Modules

The following source files illustrate a program that uses public and external declarations to access instruction labels. The program consists of two modules called *hello* and *display*.

The *hello* module is the program's initializing module. Execution starts at the instruction labeled *start* in the *hello* module. After initializing the data segment, the program calls the procedure *display* in the *display*

module. 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
```

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 examples 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*.

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:

```
ld 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.

## 7.5 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**.

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 XENIX, 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:

```
.DATA
COMM    NEAR nbuffer:BYTE:30
.FARDATA
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:

```
        .286
        .MODEL SMALL

        INCLUDEcommunal.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


8.1 Introduction 8-1

8.2 Using Operands with Directives 8-1

8.3 Using Operators 8-2
- 8.3.1 Calculation Operators 8-3
- 8.3.2 Relational Operators 8-8
- 8.3.3 Segment-Override Operator 8-10
- 8.3.4 Type Operators 8-11
- 8.3.5 Operator Precedence 8-19

8.4 Using the Location Counter 8-20

8.5 Using Forward References 8-21
- 8.5.1 Forward References to Labels 8-22
- 8.5.2 Forward References to Variables 8-24

8.6 Strong Typing for Memory Operands 8-25

## 8.1 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.

## 8.2 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.

## 8.3 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

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, the record operators and the macro operators.

### 8.3.1 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.

---

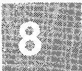
#### Arithmetic Operators

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 Section 8.2.5, "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 ''Structures.''

### Example

```
            .DATA
date        STRUC                          ; Declare structure
month       DB      ?
day         DB      ?
year        DW      ?
date        ENDS
yesterday   date    <12,31,1987>           ; Define structure variables
today       date    <1,1,1988>

            .CODE
            .
            .
            .
            mov     bh,yesterday.day       ; Load structure variable

            mov     bx,OFFSET 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.

8

### 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. "Indirect Memory Operands," 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.

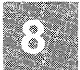


**Table 8.2**

**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
```

### 8.3.2 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**

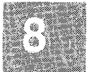
**Relational Operators**

| 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 |

*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)
```

## 8.3.3 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 (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 "SEG Operator."

---

*Note*

When a segment override is given with an indexed operand, the segment must be specified outside the index operators. For example, *es:[di]* is correct, but *[es:di]* 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 "Associating Segments with Registers."

### 8.3.4 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.

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 1

```
            .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 "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.

### Examples

```
            .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.

8

### 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.

### Examples

```
            .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.

## Examples

```
array       DD      100 DUP(0FFFFFFFh)
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
```

## 8.3.5 Operator Precedence

Expressions are evaluated according to the following rules:

- Operations of highest precedence are performed first.
- Operations of equal precedence are performed from left to right.
- The order of evaluation can be overridden by using parentheses. Operations in parentheses are always performed before any adjacent operations.

The order of precedence for all operators is listed in Table 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
```

## 8.4 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.

## 8.5 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.

### 8.5.1 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.) 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
```

## 8.5.2 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.

## 8.6 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:

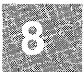
```
            .DATA
string      DB       "A message."
            .CODE
            .
            .
            .
            mov      ax,string[1]
```

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

```
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

9.1 Introduction 9-1

9.2 Using Conditional-Assembly Directives 9-1
- 9.2.1 Testing Expressions with IF and IFE 9-2
- 9.2.2 Testing the Pass with IF1 and IF2 9-3
- 9.2.3 Testing Symbol Definition with IFDEF and IFNDEF 9-3
- 9.2.4 Verifying Macro Parameters with IFB and IFNB 9-4
- 9.2.5 Comparing Macro Arguments with IFIDN and IFDIF 9-5

9.3 Using Conditional-Error Directives 9-6
- 9.3.1 Generating Unconditional Errors with .ERR, .ERR1, and .ERR2 9-7
- 9.3.2 Testing Expressions with .ERRE or .ERRNZ 9-8
- 9.3.3 Verifying Symbol Definition with .ERRDEF and .ERRNDEF 9-9
- 9.3.4 Testing for Macro Parameters with .ERRB and .ERRNB 9-9
- 9.3.5 Comparing Macro Arguments with .ERRIDN and .ERRDIF 9-10

## 9.1 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 ''Using Macro Operators.''

## 9.2 Using Conditional-Assembly Directives

The conditional-assembly directives include the following:

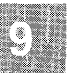
| | | |
|---|---|---|
| **IF** | **IFDEF** | **IFNB** |
| **IF1** | **IFDIF** | **IFNDEF** |
| **IF2** | **IFE** | **ENDIF** |
| **IFB** | **IFIDN** | **ELSE** |

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**.

### 9.2.1 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*.

### 9.2.2 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 Section 2.3.7, "Reading a Pass 1 Listing."

**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
```

### 9.2.3 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**

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

In this example, *buff* is allocated only if *buffer* has been previously defined.

One way to use this conditional block is to leave *buffer* undefined in the source file and define it if needed by using the **-D***symbol* option (see "Defining Assembler Symbols") 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
```

### 9.2.4 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.

## 9.2.5 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**).

## 9.3 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.

### 9.3.1 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 Section 2.2.3, ''Creating a Pass 1 Listing'').

You can place these directives within conditional-assembly blocks or macros to see which blocks are being expanded.

**Example**

```
IFDEF       dos
            .
            .
            .
ELSE
IFDEF       xenix
            .
            .
            .
ELSE
            .ERR
            %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.

### 9.3.2 Testing Expressions with .ERRE or .ERRNZ

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.

### 9.3.3 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.

**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.

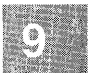
### 9.3.4 Testing for Macro Parameters with .ERRB and .ERRNB

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 **.ERRB** directive generates an error if no argument is passed to the macro. The **.ERRNB** directive generates an error if more than one argument is passed to the macro.

## 9.3.5 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


10.1 Introduction 10-1

10.2 Using Equates 10-1
- 10.2.1 Redefinable Numeric Equates 10-1
- 10.2.2 Nonredefinable Numeric Equates 10-2
- 10.2.3 String Equates 10-4

10.3 Using Macros 10-5
- 10.3.1 Defining Macros 10-6
- 10.3.2 Calling Macros 10-8
- 10.3.3 Using Local Symbols 10-9
- 10.3.4 Exiting from a Macro 10-11

10.4 Defining Repeat Blocks 10-11
- 10.4.1 The REPT Directive 10-12
- 10.4.2 The IRP Directive 10-13
- 10.4.3 The IRPC Directive 10-13

10.5 Using Macro Operators 10-15
- 10.5.1 Substitute Operator 10-15
- 10.5.2 Literal-Text Operator 10-17
- 10.5.3 Literal-Character Operator 10-18
- 10.5.4 Expression Operator 10-19
- 10.5.5 Macro Comments 10-20

10.6 Using Recursive, Nested, and Redefined Macros 10-21
- 10.6.1 Using Recursion 10-21
- 10.6.2 Nesting Macro Definitions 10-21
- 10.6.3 Nesting Macro Calls 10-22
- 10.6.4 Redefining Macros 10-23
- 10.6.5 Avoiding Inadvertent Substitutions 10-24

10.7 Managing Macros and Equates 10-25

10.7.1 Using Include Files 10-25
10.7.2 Purging Macros from Memory 10-26

## 10.1 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.

## 10.2 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)

### 10.2.1 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 "Defining Repeat Blocks," for more information on repeat blocks.

### 10.2.2 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 Section 10.1.3, "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
```

### 10.2.3 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"
parml       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 parml           ; Increment word value of
                                        ;   argument passed on stack
```

## 10.3 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 "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 "Passing Arguments on the Stack," does the same thing as the *addup* macro in "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.

### 10.3.1 Defining Macros

The **MACRO** and **ENDM** directives are used to define macros. **MACRO** designates the beginning of the macro block and **ENDM** designates the end 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 ''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 ''Using Local Symbols,'' and ''Exiting from a Macro.'' Macro operators can also be used in macro definitions, as described in ''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.

### 10.3.2 Calling Macros

A macro call directs **masm** to copy the statements of the macro to the point of the call and to replace any parameters in the macro statements with the corresponding actual arguments.

#### Syntax

*name* [*argument* [,*argument*]...]

The *name* must be the name of a macro defined earlier in the source file. The *arguments* can be any text. For example, symbols, constants, and registers are often given as arguments. Any number of arguments can be given, but they must all fit on one line. Multiple arguments must be separated by commas, spaces, or tabs.

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 ''Controlling Listing of Macros'').

### 10.3.3 Using Local Symbols

The **LOCAL** directive can be used within a macro to define symbols that are available only within the defined macro.

---

*Note*

In this context, the term ''local'' is not related to the public availability of a symbol, as described in Chapter 7, ''Creating Programs from Multiple Modules,'' or to variables that are defined to be local to a procedure, as described in ''Using Local Variables.'' ''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.

### 10.3.4 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.

## 10.4 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.

### 10.4.1 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.

### 10.4.2 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.

### 10.4.3 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 character 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 "Substitute Operator," for more information.

## 10.5 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.

### 10.5.1 Substitute Operator

The substitute operator (**&**) forces **masm** to replace a parameter with its corresponding actual argument value.

#### Syntax

*¶meter*

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
```

---

## 10.5.2 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
```

---

### 10.5.3 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.

## 10.5.4 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 "Sending Messages to Standard Output"; the **IF2** directive is described in "Testing the Pass with IF1 and IF2 Directives."

## 10.5.5 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 "Controlling Page Breaks."

## 10.6 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.

### 10.6.1 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.

### 10.6.2 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

### 10.6.3 Nesting Macro Calls

Macro definitions can contain calls to other macros. Nested macro calls are expanded like any other macro call, but only when the outer macro is called.

### Example

```
ex          MACRO   text,val    ; Inner macro definition
            IF2
            %OUT    The expression (&text) has the value: &val
            ENDIF
            ENDM

express     MACRO   expression ; Outer macro definition
            ex      <expression>,%(expression)
            ENDM
            .
            .
            .
            express <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 "Sending Messages to Standard Output"). 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. "Literal-Text Operator," explains why.

### 10.6.4 Redefining Macros

Macros can be redefined. You do not need to purge the macro before redefining it. The new definition automatically replaces the old definition. If you redefine a macro from within the macro itself, make sure there are no 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.

### 10.6.5 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.

## 10.7 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.

### 10.7.1 Using Include Files

The **INCLUDE** directive inserts source code from a specified file into the source file from which the directive is given.

**Syntax**

**INCLUDE** *filespec*

The *filespec* must specify an existing file containing valid assembler statements. When the assembler encounters an **INCLUDE** directive, it opens the specified source file and begins processing its statements. When all statements have been read, **masm** continues with the statement immediately following the **INCLUDE** directive.

The *filespec* can be given either as a file name, or as a complete 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 Section 2.2.7, "Setting a Search Path for Include Files," 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
```

### 10.7.2 Purging Macros from Memory

The **PURGE** directive can be used to delete a currently defined macro from memory.

**Syntax**

**PURGE** *macroname*[,*macroname*]...

Each *macroname* is deleted from memory when the directive is encountered at assembly time.

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.

**Examples**

```
GetStuff
PURGE   GetStuff
```

These examples call a macro and then purge 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

11.1 Introduction 11-1

11.2 Sending Messages to Standard Output 11-1

11.3 Controlling Page Format in Listings 11-2
  11.3.1 Setting the Listing Title 11-2
  11.3.2 Setting the Listing Subtitle 11-3
  11.3.3 Controlling Page Breaks 11-3

11.4 Controlling the Contents of Listings 11-5
  11.4.1 Suppressing and Restoring Listing Output 11-5
  11.4.2 Controlling Listing of Conditional Blocks 11-6
  11.4.3 Controlling Listing of Macros 11-7

11.5 Controlling Cross-Reference Output 11-9

## 11.1 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.

## 11.2 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 ''Nesting Macro Calls.''

### 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.

## 11.3 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 |

### 11.3.1 Setting the Listing Title

The **TITLE** directive specifies a title to be used on each page of assembly listings.

**Syntax**

**TITLE** *text*

The *text* 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.

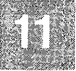
**Example**

```
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
```

### 11.3.2 Setting the Listing Subtitle



The **SUBTTL** directive specifies the subtitle used on each page of assembly listings.

**Syntax**

**SUBTTL** *text*

The *text* can be any combination of characters up to 60 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 Section 11.2.3, ''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
```

### 11.3.3 Controlling Page Breaks

The **PAGE** directive can be used to designate the line length and width for the program listing, to increment the section and adjust the section number accordingly, or to generate a page break in the listing.

**Syntax**

**PAGE** [[*length*],*width*]
PAGE

If *length* and *width* are specified, the **PAGE** directive sets the maximum number of lines per page to *length* and the maximum number of characters per line to *width*. The *length* must be in the range 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 "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.

## 11.4 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 |

### 11.4.1 Suppressing and Restoring Listing Output

The **.LIST** and **.XLIST** directives specify which source lines are included in the program listing.

### Syntax

**.LIST**
**.XLIST**

The **.XLIST** directive suppresses copying of subsequent source lines to the program listing. The **.LIST** directive restores copying. The directives are typically used in pairs to prevent a particular section of a source file from being copied to the program listing.

The **.XLIST** directive overrides other listing directives such as **.SFCOND** or **.LALL**.

**Example**

```
ST
```

### 11.4.2 Controlling Listing of Conditional Blocks

The **.SFCOND**, **.LFCOND**, and **.TFCOND** directives control whether false-conditional blocks should be included in assembly listings.

**Syntax**

> **.SFCOND**
> **.LFCOND**
> **.TFCOND**

The **.SFCOND** directive suppresses the listing of any subsequent conditional blocks whose condition is false. The **.LFCOND** directive restores the listing of these blocks. Like **.LIST** and **.XLIST**, conditional-listing directives can be used to suppress listing of conditional blocks in sections of a program.

The **.TFCOND** directive toggles the current status of listing of conditional blocks. This directive can be used in conjunction with the **-X** option of the assembler. By default, conditional blocks are not listed on start-up. However, they will be listed on start-up if the **-X** 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 Section 2.2.14, ''Listing False Conditionals.''

### 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.

## 11.4.3 Controlling Listing of Macros

The **.LALL**, **.XALL**, and **.SALL** directives control the listing of the expanded macros calls. The assembler always lists the full macro definition. The directives only affect expansion of macro calls.

### Syntax

> **.LALL**
> **.XALL**
> **.SALL**

The **.LALL** directive causes **masm** to list all the source statements in a macro expansion, including normal comments (preceded by a single semicolon) but not macro comments (preceded by a double semicolon).

The **.XALL** directive causes **masm** to list only those source statements in a macro expansion that generate code or data. For 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.

## 11.5 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


12.1 Introduction 12-1

12.2 Using the 8086-Family Processors 12-1
- 12.2.1 Processor Differences 12-1
- 12.2.2 Real and Protected Modes 12-3

12.3 Segmented Addresses 12-4

12.4 Using 8086-Family Registers 12-5
- 12.4.1 Segment Registers 12-8
- 12.4.2 General-Purpose Registers 12-8
- 12.4.3 Other Registers 12-10
- 12.4.4 The Flags Register 12-11
- 12.4.5 8087-Family Registers 12-13

12.5 Using the 80386 Processor 12-13

## 12.1 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.



## 12.2 Using the 8086-Family Processors

The Intel Corporation manufactures the group of processors referred to in this manual as the 8086-family processors. The XENIX 286/386 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.

### 12.2.1 Processor Differences

The main 8086-family processors are discussed below:

| Processor | Description |
|---|---|
| 8088 and 8086 | These processors work in real mode. They are designed to run a single process. No provision is made to protect one part of memory from actions occurring in another part of memory. The processor can address up to one megabyte of memory. Addresses specified in assembly language correspond to physical memory addresses.<br><br>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 ''Aligning Data''). |

| | |
|---|---|
| 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.<br><br>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.<br><br>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 ''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.



### 12.2.2 Real and Protected Modes

Protected mode is the multiple-process mode used in XENIX. 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 XENIX and OS/2, provide extended functions for doing the kinds of tasks that are currently done by using the previously described restricted practices.

## 12.3 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
```

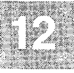
In protected mode, the address *53C2:107A* represents a movable address. The segment portion of the address is a selector assigned a physical address by the operating system. The applications programmer has no control (and needs none) over the physical address represented by the selector.

### 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.

## 12.4 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.

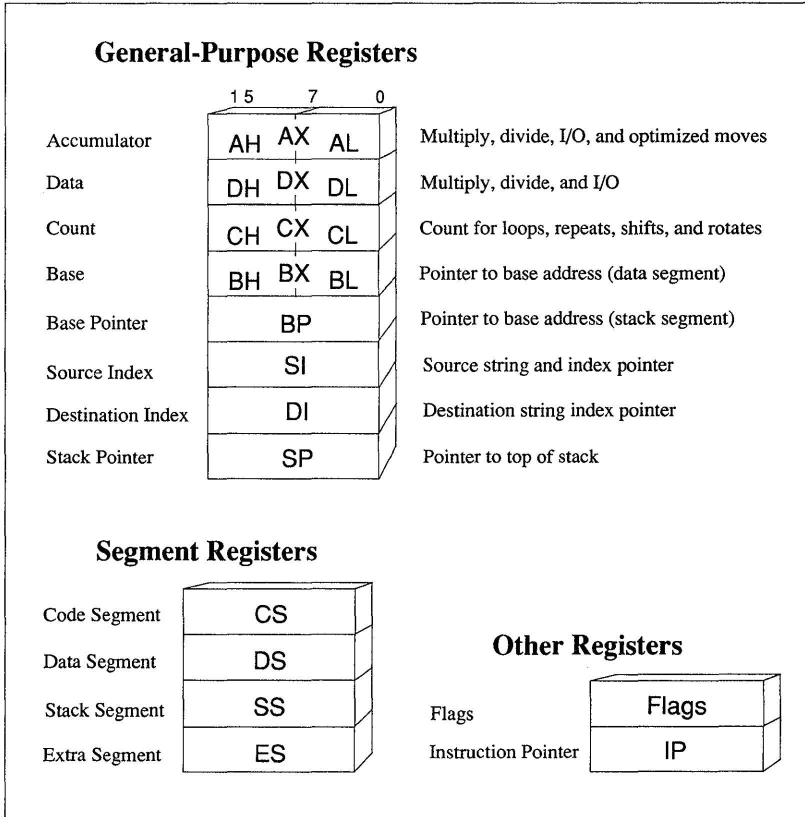



**Figure 12-1** Register for 8088-80286 Processors

## 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.

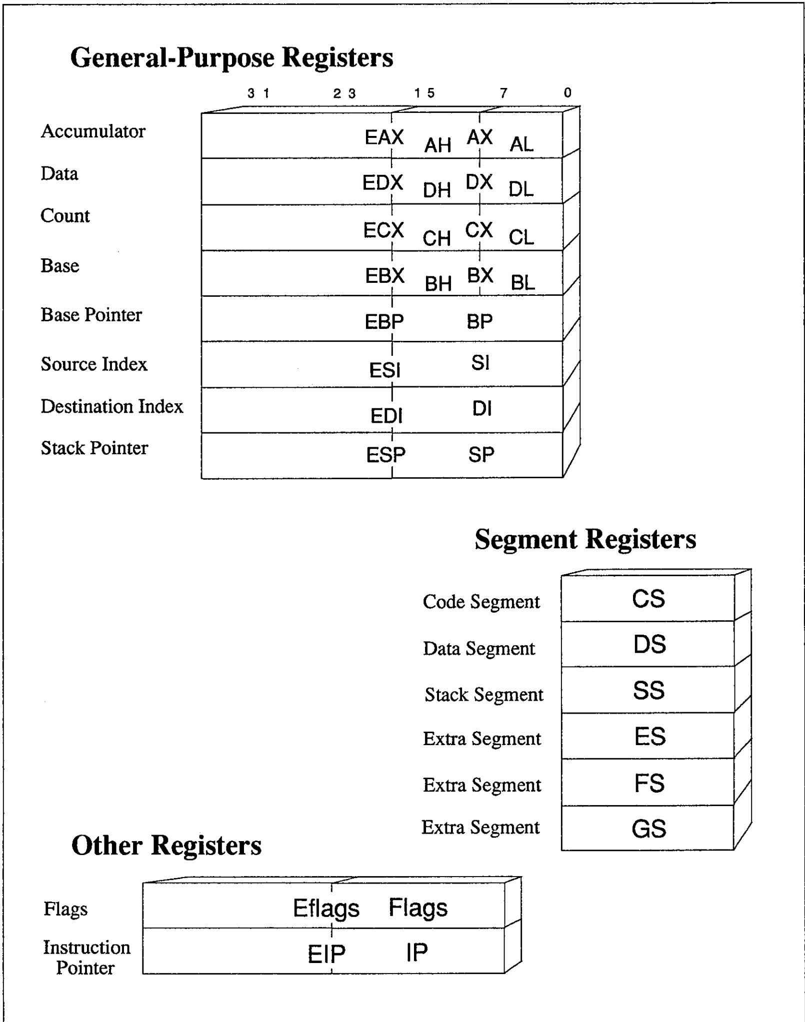



**Figure 12-2** Extended Registers of 80386 Processor

### 12.4.1 Segment Registers

At run time, all addresses are relative to one of four segment registers: **CS**, **DS**, **SS**, or **ES**. These registers and the segments they correspond to are listed below:

| Segment | Purpose |
|---|---|
| Code Segment (**CS**) | Addresses in the segment pointed to by this register contain the encoded instructions and operands specified by the program. |
| Data Segment (**DS**) | Addresses in the segment pointed to by this register normally contain data allocated by the program. |
| Stack Segment (**SS**) | Addresses in the segment pointed to by this register are available for instructions that store data on the program stack. A stack is an area of memory reserved for storing temporary data. For information on using stacks, see "Transferring Data to and from the Stack." |
| 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**. |

### 12.4.2 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**, **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.<br>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.<br>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 "Indirect Memory Operands"). 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 "Passing Arguments on the Stack." 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.

### 12.4.3 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. "Using the Stack," and "Passing Arguments on the Stack," discuss operation of the stack in more detail.<br><br>**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.



### 12.4.4 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.

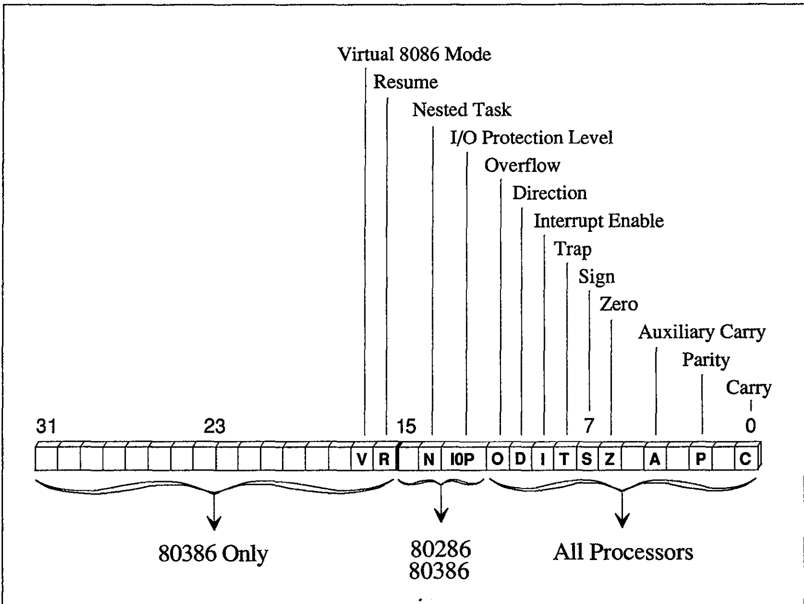



**Figure 12-3** Flags for 8088-80386 Processors

The thirteen 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. |



### 12.4.5 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 "Real-Number Variables."

## 12.5 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


13.1 Introduction 13-1

13.2 Using Immediate Operands 13-1

13.3 Using Register Operands 13-2

13.4 Using Memory Operands 13-4
- 13.4.1 Direct Memory Operands 13-4
- 13.4.2 Indirect Memory Operands 13-6
- 13.4.3 80386 Indirect Memory Operands 13-11

## 13.1 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

```
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.

## 13.2 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
```

## 13.3 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 ‘‘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

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
```

## 13.4 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.

---

### 13.4.1 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 "Segment-Override Operator."

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
```

---

### 13.4.2 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 "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.

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]**<br>**[BP]**<br>**[DI]** | Effective address is contents of register |
| Based or indexed | **[BX]***disp*<br>*displacement***[BP]**<br>*displacement***[DI]**<br>*displacement***[SI]** | Effective address is contents of register and *displacement* |
| Based indexed | **[BX][DI]**<br>**[BP][DI]**<br>**[BX][SI]**<br>**[BP][SI]** | Effective address is contents of base register and contents of index register |
| Based indexed with displacement | *displacement***[BX][DI]**<br>*displacement***[BP][DI]**<br>*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 "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 (**:**).

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

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 "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 "Multiplying and Dividing by Constants."

### 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. "80386 Indirect Memory Operands," shows an enhanced version that uses 80386 instructions and addressing modes.

### 13.4.3 80386 Indirect Memory Operands

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.

---



## 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 ''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 ''Moving and Extending Values'').

- **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 ''Pushing and Popping,'' and ''Multiplying, '' 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


14.1 Introduction 14-1

14.2 Transferring Data 14-1
- 14.2.1 Copying Data 14-1
- 14.2.2 Exchanging Data 14-2
- 14.2.3 Looking Up Data 14-2
- 14.2.4 Transferring Flags 14-3

14.3 Converting between Data Sizes 14-4
- 14.3.1 Extending Signed Values 14-4
- 14.3.2 Extending Unsigned Values 14-6
- 14.3.3 Moving and Extending Values 14-6

14.4 Loading Pointers 14-7
- 14.4.1 Loading Near Pointers 14-7
- 14.4.2 Loading Far Pointers 14-8

14.5 Transferring Data to and from the Stack 14-10
- 14.5.1 Pushing and Popping 14-10
- 14.5.2 Using the Stack 14-13
- 14.5.3 Saving Flags on the Stack 14-14
- 14.5.4 Saving All Registers on the Stack 14-14

14.6 Transferring Data to and from Ports 14-15

## 14.1 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.

## 14.2 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.

### 14.2.1 Copying Data



The **MOV** instruction is the most common method of moving data. This instruction can be thought of as a "copy" instruction, since it always copies the source operand to the destination operand. Immediately after a **MOV** instruction, the source and destination operands both contain the same value. The old value in the destination operand is destroyed.

**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
```

## 14.2.2 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"
```

## 14.2.3 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.

---



### 14.2.4 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 ''Saving Flags on the Stack.''

**SAHF** is often used with a coprocessor to transfer coprocessor control flags to processor control flags. ''Controlling Program Flow,'' explains and illustrates this technique.

## 14.3 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.

### 14.3.1 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
```

### 14.3.2 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)
```

### 14.3.3 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.

**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
```



## 14.4 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.

### 14.4.1 Loading Near Pointers

The **LEA** instruction loads a near pointer into a specified register.

**Syntax**

**LEA** *register,memory*

The destination register may be any general-purpose register. The source operand may be any memory operand. The effective address of the source operand is placed in the destination 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 "80386 Indirect Memory Operands.") 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 "Multiplying and Dividing by Constants."

## 14.4.2 Loading Far Pointers

The **LDS** and **LES** instructions load far pointers. **Syntax**

**LDS** *register,memory*
**LES** *register,memory*

The memory address being pointed to is specified in the source operand, and the register where the offset 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

14

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
```

## 14.5 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 "Using Procedures," and "Using Interrupts"). You can also use the **PUSH** and **POP** instructions and their variations to store values on the stack.

### 14.5.1 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.



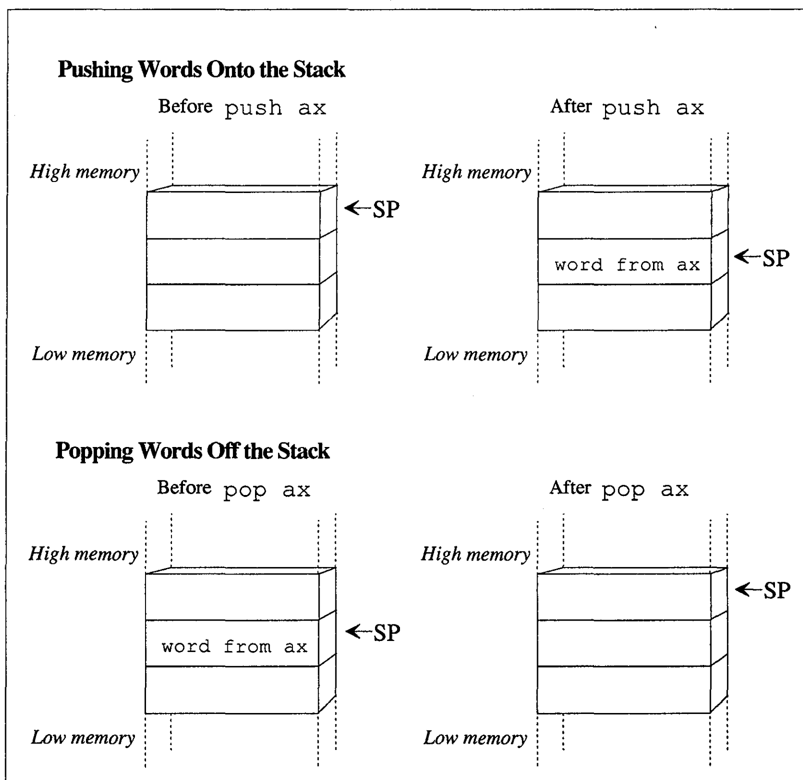



**Figure 14-1** Stack Status after Pushes and Pops

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.

## 14.5.2 Using the Stack

The stack can be used to store temporary data. For example, in the Microsoft calling convention, the stack is used to pass arguments to a procedure. The arguments are pushed 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 "Passing Arguments on the Stack."



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**.

### 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
```

### 14.5.3 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**.

### 14.5.4 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.

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**.



## 14.6 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 XENIX and other protected-mode operating systems, **IN** and **OUT** are privileged instructions and can only be used in privileged mode.

---

### 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 "Transferring Strings to and from Ports."

# Chapter 15

# Doing Arithmetic and Bit Manipulations

15.1 Introduction 15-1

15.2 Adding 15-1
15.2.1 Adding Values Directly 15-1
15.2.2 Adding Values in Multiple Registers 15-3

15.3 Subtracting 15-3
15.3.1 Subtracting Values Directly 15-4
15.3.2 Subtracting with Values in Multiple Registers 15-5

15.4 Multiplying 15-6

15.5 Dividing 15-9

15.6 Calculating with Binary Coded Decimals 15-10
15.6.1 Unpacked BCD Numbers 15-11
15.6.2 Packed BCD Numbers 15-13

15.7 Doing Logical Bit Manipulations 15-14
15.7.1 AND Operations 15-15
15.7.2 OR Operations 15-16
15.7.3 XOR Operations 15-17
15.7.4 NOT Operations 15-18

15.8 Scanning for Set Bits 15-19

15.9 Shifting and Rotating Bits 15-20
15.9.1 Multiplying and Dividing by Constants 15-22
15.9.2 Moving Bits to the Least-Significant Position 15-24
15.9.3 Adjusting Masks 15-24
15.9.4 Shifting Multiword Values 15-24
15.9.5 Shifting Multiple Bits 15-25

## 15.1 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.

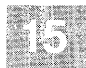
This chapter tells you how to use the instructions that do calculations on numbers and bits.

## 15.2 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 Section 15.5.

### 15.2.1 Adding Values Directly

The **ADD** and **INC** instructions are used for adding to values in registers or memory.

The **INC** instruction takes a single register or memory operand. The value of the operand is 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 ''Testing Bits and Jumping''). You can also use the **INTO** instruction to trigger the overflow interrupt (interrupt 4) if the overflow flag is set.

### Examples

```
            .DATA
mem8        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.

### 15.2.2 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.

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.

15

**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
```

## 15.3 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 Section 15.5.

### 15.3.1 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.

### 15.3.2 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 more-significant portion. This can be done by using the **SUB** instruction for the least-significant portion and the **SBB** instruction for more-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
```

## 15.4 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 "Multiplying and Dividing by Constants") or by using 80386 scaling (see "Loading Near Pointers").

---

**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}*

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
```

## 15.5 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 "Converting between Data Sizes") 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 "Using Interrupts."

---

*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 "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
```

## 15.6 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.

---

### 15.6.1 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
```

**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.

### 15.6.2 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 eight-bit arithmetic calculations on each byte separately. The result should always be in the **AL** register. After each operation, use the corresponding BCD instruction to adjust the result. The decimal adjust instructions do not take an operand. They always work on the value in the **AL** register.

Unlike the ASCII adjust instructions, the decimal adjust instructions never affect **AH**. The auxiliary carry flag is set if the digit in the lower 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.

## 15.7 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 are 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.

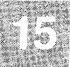
---

### 15.7.1 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*}



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.

## 15.7.2 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.

### 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
```

### 15.7.3 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.

#### 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.

### 15.7.4 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 "Bitwise Logical Operators") 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
```

## 15.8 Scanning for Set Bits

### 80386 Only

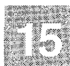
The 80386 processor has instructions for scanning bits to find the first or last set bit in a register value. These instructions can be used to find the position of a set bit in a mask or other value. They can also check to see if a register value is 0.

### 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:     .
```

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 the chapter "Processing Strings" for more information on the string instructions used in this example.

### 15.9 Shifting and Rotating Bits

The 8086-family processors provide a complete set of instructions for shifting and rotating bits. Bits can be moved right (toward the most-significant bits) or left (toward the 0 bit). Values shifted off the end 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.

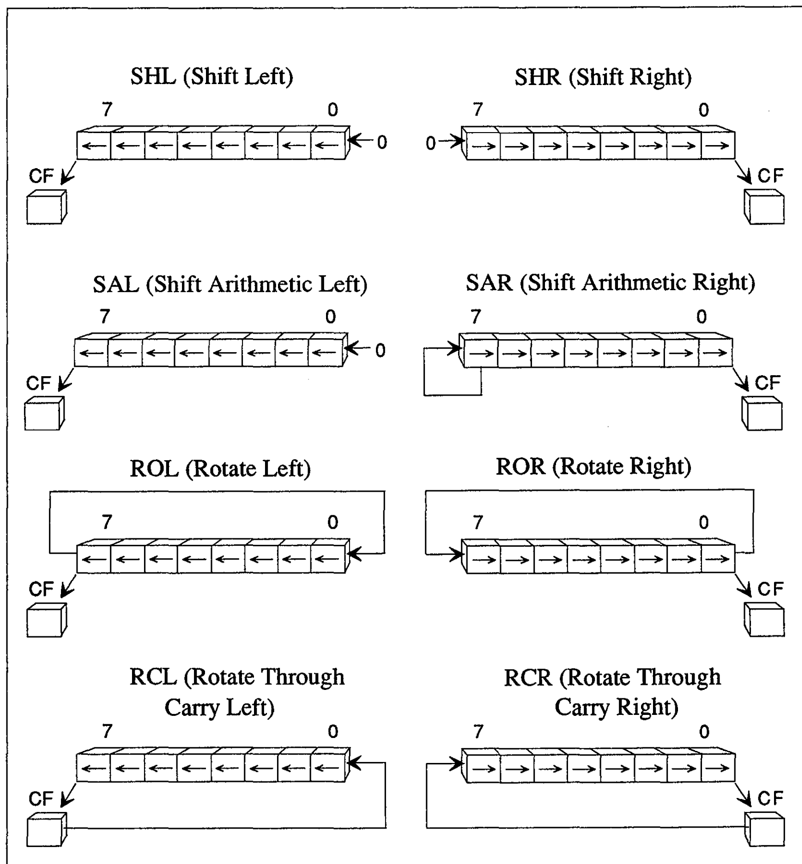





**Figure 15-1** Shifts and Rotates

## 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.

---

*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
```

---

### 15.9.1 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
```

### 15.9.2 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 "Defining and Redefining Interrupt Routines."

### 15.9.3 Adjusting Masks

Masks for logical instructions can be shifted to new bit positions. For example, an operand that masks off a bit or group of bits can be shifted to move the mask to a different position.

**Example**

```
            .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 runtime.

### 15.9.4 Shifting Multiword Values

Sometimes it is necessary to shift a value that is too large to fit in a register. In this case, you can shift each part separately, passing the shifted bits through the carry flag. The **RCR** or **RCL** instructions must be used to move the carry value from the first register to the second.

**RCR** and **RCL** can also be used to initialize the high or low bit of an operand. Since the carry flag is treated as part of the operand (like using a 9-bit operand), the flag value before the operation is crucial. The carry flag may be set by a previous instruction, or you can set it directly using

the **CLC** (Clear Carry Flag), **CMC** (Complement Carry Flag), and **STC** (Set Carry Flag) instructions.

### Example

```
                .DATA
mem32           DD      500000
                .CODE
                .
                .                              ; 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
```

## 15.9.5 Shifting Multiple Bits

### 80386 Only

15

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

16.1 Introduction 16-1

16.2 Jumping 16-1
- 16.2.1 Jumping Unconditionally 16-1
- 16.2.2 Jumping Conditionally 16-3

16.3 Looping 16-12

16.4 Setting Bytes Conditionally 16-15

16.5 Using Procedures 16-16
- 16.5.1 Calling Procedures 16-16
- 16.5.2 Defining Procedures 16-17
- 16.5.3 Passing Arguments on the Stack 16-19
- 16.5.4 Using Local Variables 16-21
- 16.5.5 Setting Up Stack Frames 16-23

16.6 Using Interrupts 16-25
- 16.6.1 Calling Interrupts 16-25
- 16.6.2 Defining and Redefining Interrupt Routines 16-27

16.7 Checking Memory Ranges 16-27

## 16.1 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.

## 16.2 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 "Forward References to Labels."

### 16.2.1 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 "Forward References to Labels."

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.

### 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.

### 16.2.2 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 ``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 ``Forward References to Labels.''

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.

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.

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.

16

**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

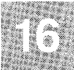
```
; If CX is greater than or equal to -20, then make DX 20, else make DX 30

            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 "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 ''Testing Bits and Jumping'') 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.

## 16.3 Looping

The 8086-family of processors has several instructions specifically designed for creating loops of repeated instructions. In addition, you can create loops using conditional jumps.

### Syntax

**LOOP** *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
```

## 16.4 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 ''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.

## 16.5 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 "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.

### 16.5.1 Calling Procedures

The **CALL** instruction saves the address following the instruction on the stack and passes control to a specified address.

### Syntax

**CALL** {*register* | *memory*}

The address is usually specified as a direct memory 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 ''Comparing and Jumping.''

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
```

## 16.5.2 Defining Procedures

Procedures are defined by labeling the start of the procedure and placing a **RET** instruction at the end. There are several variations on this syntax.

### Syntax 1

*label* **PROC** [NEAR | **FAR**]
*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.

### 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 "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.

### 16.5.3 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.

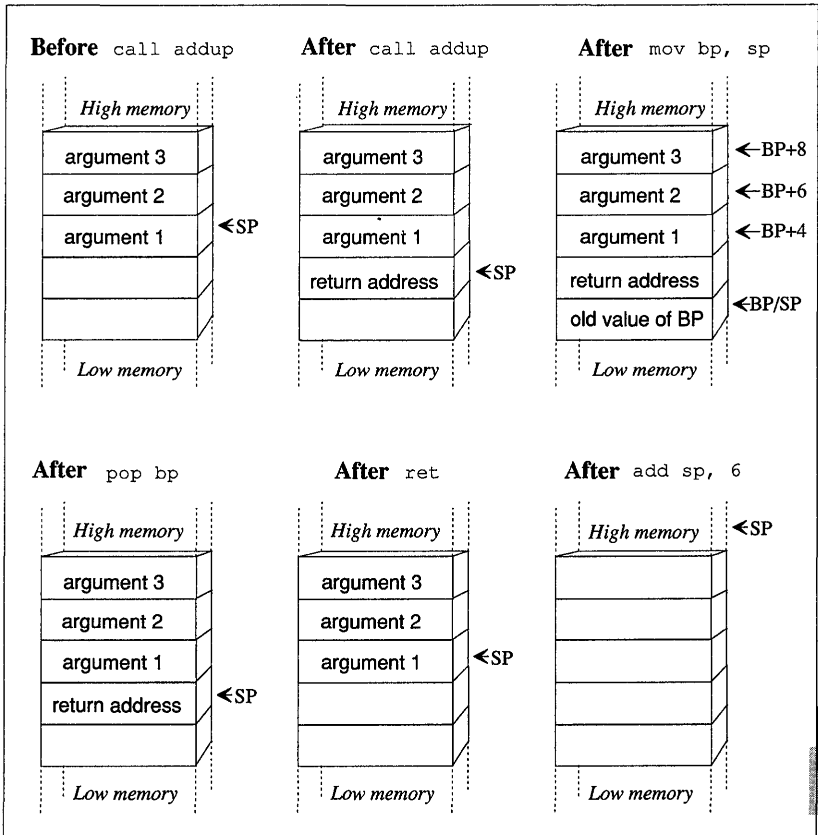



**Figure 16-1** Procedure Arguments on the Stack

## 16.5.4 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 ‘‘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.

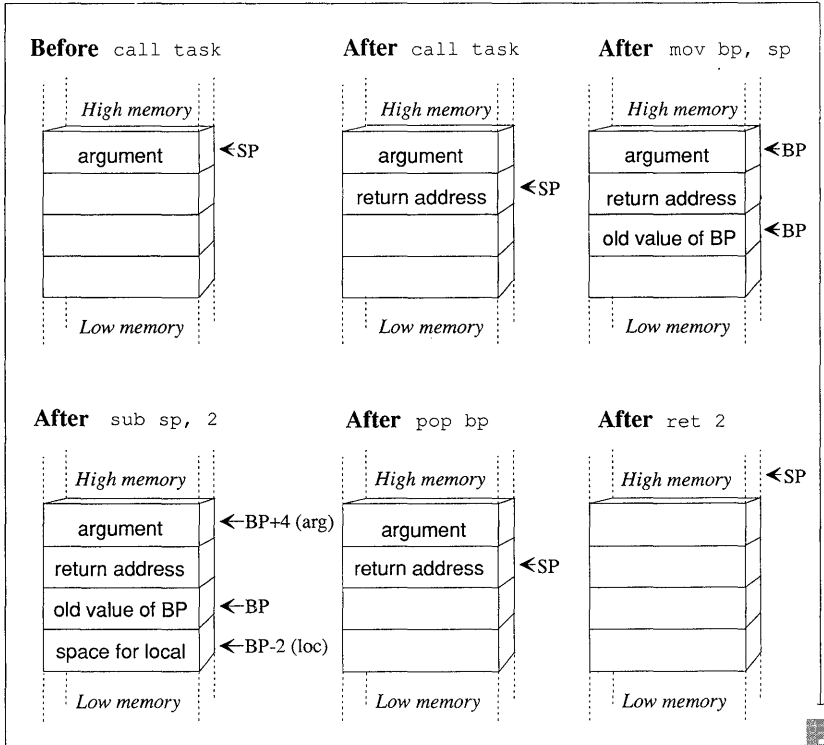



**Figure 16-2** Local Variables on the Stack

## 16.5.5 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 examples in ''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.

## 16.6 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.

### 16.6.1 Calling Interrupts

Interrupts are called with the **INT** instruction. **Syntax**

> **INT** *interruptnumber*
> **INTO**

The **INT** instruction takes an immediate operand with a value between 0 and 255.

When 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.

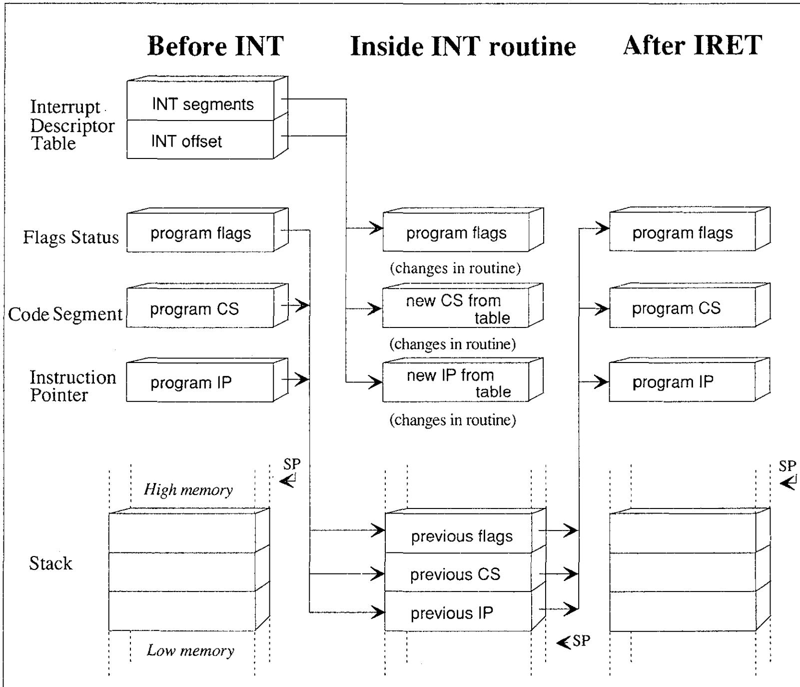



**Figure 16-3** Operation of Interrupts

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. ''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.

### 16.6.2 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**.



## 16.7 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. XENIX does not provide an interrupt routine for interrupt 5, so you must write your own. For more information, see "Using Interrupts."

## 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
```

## 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.

# Chapter 17

# Processing Strings


17.1 Introduction 17-1

17.2 Setting Up String Operations 17-1

17.3 Moving Strings 17-5

17.4 Searching Strings 17-7

17.5 Comparing Strings 17-8

17.6 Filling Strings 17-10

17.7 Loading Values from Strings 17-11

17.8 Transferring Strings to and from Ports 17-12

## 17.1 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 doublewords 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.



## 17.2 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.

---

## 17.3 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 "Setting Up String Operations." Then use the **REP** instruction with the **MOVS** instruction.

17

## 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 double-words 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
```

## 17.4 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 double-words.

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**.

## 17.5 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 "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.

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.

## 17.6 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 "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.

## 17.7 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 1

```
          .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
```

Example 1 loads, processes, and displays each byte in a string of bytes.

## 17.8 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 XENIX 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


18.1 Introduction 18-1

18.2 Coprocessor Architecture 18-1
  18.2.1 Coprocessor Data Registers 18-2
  18.2.2 Coprocessor Control Registers 18-3

18.3 Emulation 18-4

18.4 Using Coprocessor Instructions 18-4
  18.4.1 Using Implied Operands in the Classical-Stack Form 18-5
  18.4.2 Using Memory Operands 18-6
  18.4.3 Specifying Operands in the Register Form 18-7
  18.4.4 Specifying Operands in the Register-Pop Form 18-8

18.5 Coordinating Memory Access 18-9

18.6 Transferring Data 18-11
  18.6.1 Transferring Data to and from Registers 18-11
  18.6.2 Loading Constants 18-15
  18.6.3 Transferring Control Data 18-16

18.7 Doing Arithmetic Calculations 18-17

18.8 Controlling Program Flow 18-24
  18.8.1 Comparing Operands to Control Program Flow 18-25
  18.8.2 Testing Control Flags after Other Instructions 18-29

18.9 Using Transcendental Instructions 18-29

18.10 Controlling the Coprocessor 18-31

## 18.1 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.

---

## 18.2 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 each 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 ''Transferring Data.''

### 18.2.1 Coprocessor Data Registers

The 8087-family coprocessors have eight 80-bit data registers. Unlike 8086-family registers, the coprocessor data registers are organized as a stack. As data is pushed into the top register, previous data items move into higher-numbered registers. Register 0 is the top of the stack; register 7 is the bottom. The syntax for specifying registers is shown below:

**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 ''Real-Number Variables.'' The registers and the register format are shown in Figure 18.1.

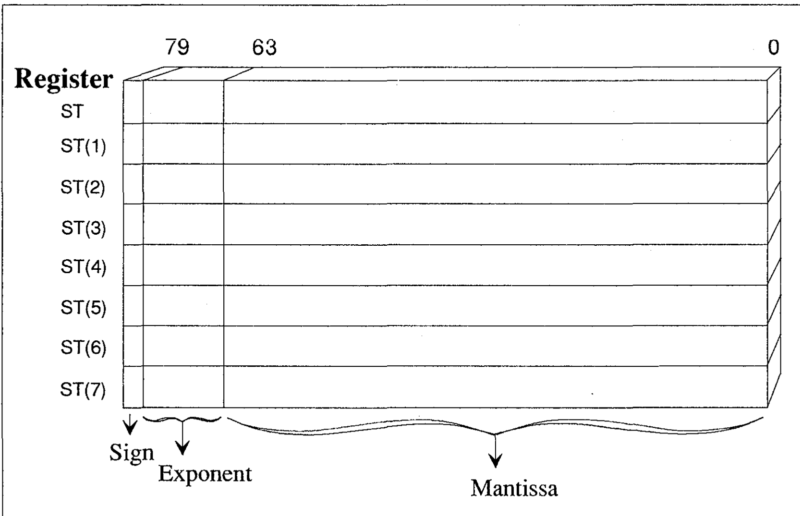



**Figure 18-1** Coprocessor Data Registers

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.

### 18.2.2 Coprocessor Control Registers

The 8087-family coprocessors have seven 16-bit control registers. The most useful control registers are made up of bit fields or flags. Some flags control coprocessor operations, while others maintain the current status of the coprocessor. In this sense, they are much like the 8086-family flags registers.

You do not need to understand these registers to do most coprocessor operations. Control flags are set by default to the values appropriate for most programs. 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 "Controlling Program Flow."

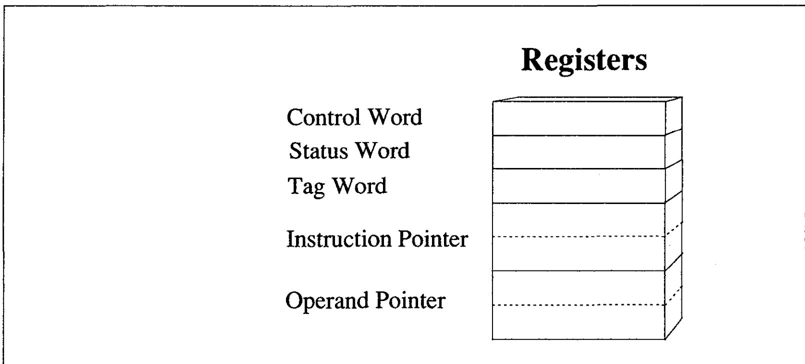



**Figure 18-2** Coprocessor Control Registers

## 18.3 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 ''Creating Code for a Floating-Point Emulator.'' For information on writing assembly-language procedures for high-level languages, see Appendix D, ''Segment Names for High-Level Languages.''

## 18.4 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 ''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**<br>F*action* **ST,ST(***num***)** | | *fadd st(5),st*<br>*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.

### 18.4.1 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, 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 ''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*

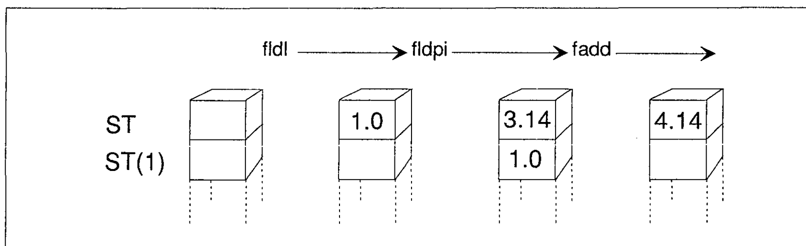
The classical-stack form with its implied operands is similar to the register-pop form, not to the register form. For example, *fadd*, with the implied operands **ST(1),ST**, is equivalent to *faddp st(1),st*, rather than to *fadd st(1),st*.

---

## 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:




## 18.4.2 Using Memory Operands

The memory form treats coprocessor registers like items on a stack. Items are pushed from memory onto the top element of the stack, or popped from the top element to memory. Since only the top item can be accessed on a traditional stack, there is no need to specify the stack operand. The top register (**ST**) is always assumed. However, the memory operand must be specified.

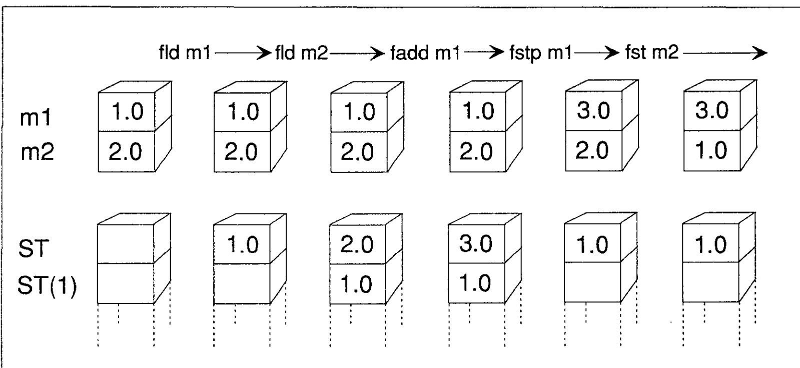
Memory operands can be used in load and store instructions (see "Transferring Data to and from Registers"). Load instructions push source values from memory to an implied destination register (**ST**). Store instructions pop source values from an implied source register (**ST**) to the destination in 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.

Memory operands can also be used in calculation instructions that operate on two values (see "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:




18

### 18.4.3 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**).

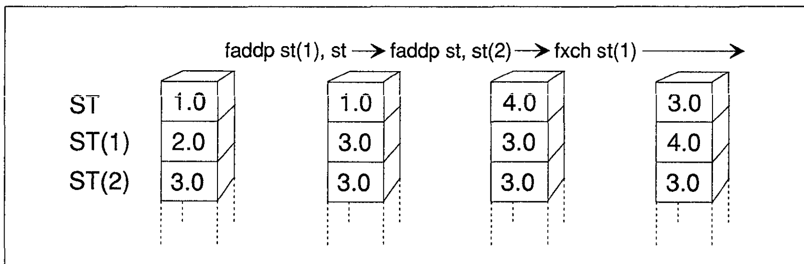
In the register form, operands are specified by name. The second operand is the source; it is not affected by the operation. The first operand is the destination; its value is replaced with the result of the operation. The stack position of the operands does not change.

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:




## 18.4.4 Specifying Operands in the Register-Pop Form

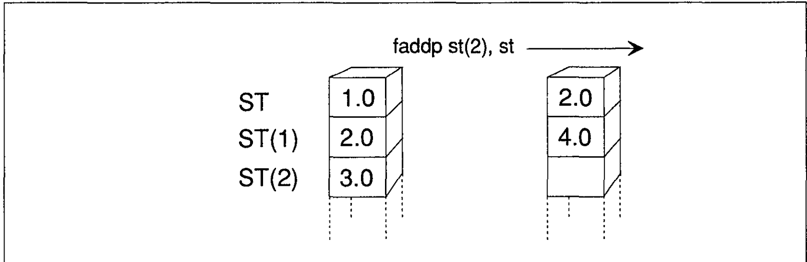
The register-pop form treats coprocessor registers as a modified stack. This form has some of the aspects of both a stack and registers. The destination register can be specified by name, but the source register must always be the stack top.

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:




## 18.5 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
```

## 18.6 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

### 18.6.1 Transferring Data to and from Registers

Data-transfer instructions transfer data between main memory and the coprocessor registers, or between different coprocessor registers. Two basic principles govern data transfers:

- The instruction determines whether a value in memory will be considered an integer, 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 ''Real-Number Variables.''

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. |

### 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

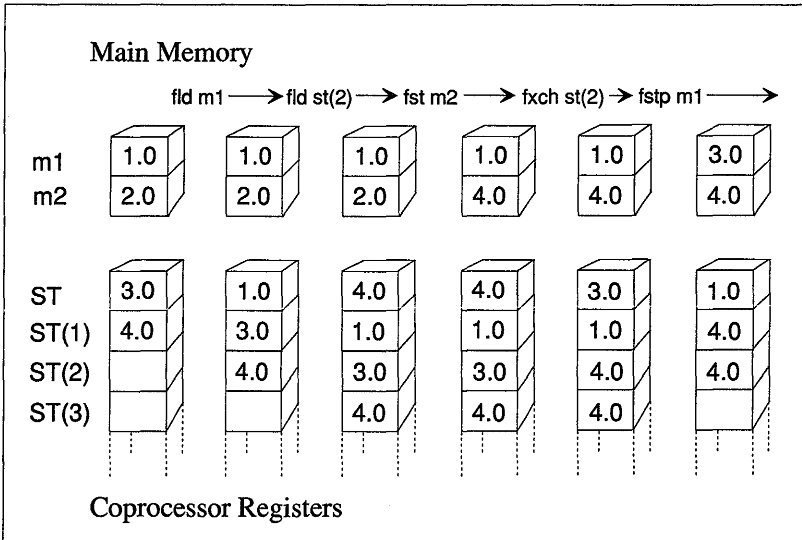
The following instructions are available for transferring BCD integers.

| 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
```

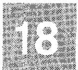
Example 2 illustrates one way of doing run-time type conversions.

### 18.6.2 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 "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** |

### 18.6.3 Transferring Control Data

The coprocessor data area, or parts of it, can be stored to memory and later loaded back. One reason for doing this is to save a snapshot of the coprocessor state before going into a procedure, and restore the same status after the procedure. Another reason is to modify coprocessor behavior by storing certain data to main memory, operating on the data with 8086-family instructions, and then loading it back to the coprocessor data area.

You can choose to transfer the entire coprocessor data area, the control registers, or just the status or control word. Applications programmers 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.

| 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.

## 18.7 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 "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.

| 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.

| 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
```

18

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 "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.

## 18.8 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 "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.

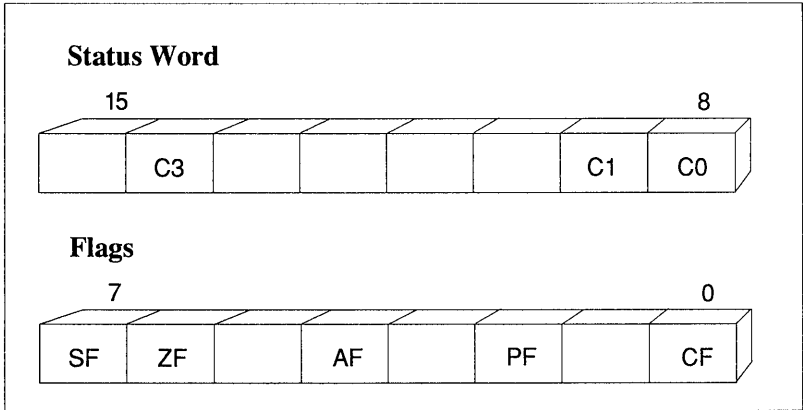



**Figure 18-3** Coprocessor and Processor Control Flags

### 18.8.1 Comparing Operands to Control Program Flow

The 8087-family coprocessors provide several instructions for comparing operands. All these instructions compare the stack top (**ST**) to a source operand, which may either be specified or 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.

**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 |

18

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
```

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 "Defining Assembler Symbols") the code is smaller and faster, but does not run on an 8087.

### 18.8.2 Testing Control Flags after Other Instructions

In addition to the compare instructions, the **FXAM** and **FPREM** instructions affect coprocessor control flags.

The **FXAM** instruction sets the value of the control flags based on the type of the number in the stack top (**ST**). This instruction is used to identify and handle special values such as infinity, zero, unnormal numbers, denormal numbers, and NANs (not a number). Certain math operations are capable of producing these special-format numbers.

**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.

## 18.9 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. |

18

**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)**.

## 18.10 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. |



### 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 "Segmented Addresses."

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


19.1 Introduction 19-1

19.2 Controlling Timing and Alignment 19-1

19.3 Controlling the Processor 19-1

19.4 Controlling Protected-Mode Processes 19-2

19.5 Controlling the 80386 19-4

## 19.1 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.

## 19.2 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 "Aligning Data") to align data or code on a given boundary. The assembler automatically inserts **NOP** instructions for alignment.

## 19.3 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 "Coordinating Memory Access." |
| **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. |

## 19.4 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 XENIX 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 |

## 19.5 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
```

# Appendix A

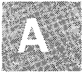
# New Features

A.1 Introduction A-1

A.2 Enhancements to masm A-1
- A.2.1 80386 Support A-1
- A.2.2 Segment Simplification A-2
- A.2.3 Performance Improvements A-3
- A.2.4 Enhanced Error Handling A-3
- A.2.5 New Options A-3
- A.2.6 String Equates A-4
- A.2.7 RETF and RETN Instructions A-4
- A.2.8 Communal Variables A-4
- A.2.9 Flexible Structure Definitions A-5

A.3 Compatibility with Assemblers and Compilers A-5

## A.1 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.

## A.2 Enhancements to masm

This version of **masm** has several important enhancements. The following sections summarize new options, directives, instructions, and other features.

### A.2.1 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 ``Using 8086-Family Registers.'')

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 (``Defining Default Assembly Behavior'')
- the **USE32** and **USE16** segment types for setting the segment word size (``Setting Segment Word Size with Use Type'')
- indirect addressing modes (``80386 Indirect Memory Operands'')

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 |
|---|---|
| Bit Scan Forward | **BSF** |
| Bit Scan Reverse | **BSR** |
| Bit Test | **BT** |
| Bit Test and Complement | **BTC** |
| Bit Test and Reset | **BTR** |
| Bit Test and Set | **BTS** |
| Move with Sign Extend | **MOVSX** |
| Move with Zero Extend | **MOVZX** |
| Set Byte on Condition | **SET***condition* |
| Double Precision Shift Left | **SHLD** |
| Double Precision Shift Right | **SHRD** |
| Move to/from Special Registers | **MOV** |
| Sine | **FSIN** |
| Cosine | **FCOS** |
| Sine Cosine | **FSINCOS** |
| IEEE Partial Remainder | **FPREM1** |
| Unordered Compare Real | **FUCOM** |
| Unordered Compare Real and Pop | **FUCOMP** |
| Unordered Compare Real and Pop Twice | **FUCOMPP** |

### A.2.2 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 "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 "Specifying MS-DOS Segment Order."

### A.2.3 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.

### A.2.4 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 ''Setting the Warning Level.''

- During assembly, messages are output to standard output. In Version 4.0 they were sent to standard error.

### A.2.5 New Options

The following command-line options have been added to Version 5.0:

| Option | Description |
|---|---|
| **-w**[0\|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 ''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 "Writing Symbolic Information to the Object File."

**-h** Displays the **masm** command line and options, as explained in "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 "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 "Setting the Segment-Order Method."

### A.2.6 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 "String Equates."

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 "Expression Operator."

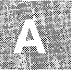
### A.2.7 RETF and RETN Instructions

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. "Defining Procedures," explains these instructions.

### A.2.8 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 "Using Multiple Modules."

### A.2.9 Flexible Structure Definitions



Structure definitions can now include conditional-assembly statements, thus enabling more flexible structures. For more information, see "Declaring Structure Types."

## A.3 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. ''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


B.1 Introduction B-1

B.2 8086 Instruction Mnemonics B-2

B.3 8087 Instruction Mnemonics B-8

B.4 80186 Instruction Mnemonics B-13

B.5 80286 Nonprotected Instruction Mnemonics B-14

B.6 80286 Protected Instruction Mnemonics B-15

B.7 80287 Instruction Mnemonics B-15

B.8 80386 Nonprotected Instruction Mnemonics B-16

B.9 80386 Protected Instruction Mnemonics B-19

B.10 80387 Instruction Mnemonics B-20

## B.1 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 chapter.

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<br>byte: AL, AH, BL, BH, CL, CH, DL, DH<br>word: AX, BX, CX, DX, SI, DI, BP, SP<br>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 |

## B.2 8086 Instruction Mnemonics

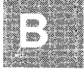
The 8086 instructions are listed below. All 8086 instructions are assembled by default.

**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 |
| CMP *r/m, immed* | Compare immediate with operand |

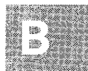
| Syntax | Action |
|---|---|
| CMP *r/m, reg* | Compare register with operand |
| CMP *reg, r/m* | Compare operand with register |
| CMPS *src, dest* | Compare strings |
| CMPSB | Compare strings byte for byte |
| CMPSW | Compare strings word for word |
| CWD | Convert word to doubleword |
| DAA | Decimal 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 6-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 |
| JE *label* | Jump on equal |

| Syntax | Action |
|---|---|
| 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 |
| LEA *r/m* | Load effective address of operand |
| LES *r/m* | Load operand into ES |
| LOCK | Lock bus |




| **Syntax** | **Action** |
|---|---|
| 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 |
| 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) |

| Syntax | Action |
|---|---|
| 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 |
| 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 |



| **Syntax** | **Action** |
|---|---|
| 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 |
| 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
```

## B.3 8087 Instruction Mnemonics

The 8087 instructions are listed below. All 8087 instructions are assembled by default.

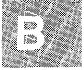
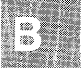



**Table B.3**
**8087 Instruction Mnemonics**

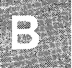
| Syntax | Action |
|---|---|
| F2XM1 | Calculate $2^x$-1 |
| FABS | Take absolute value of top of stack |
| FADD | Add real |
| FADD *mem* | Add real from memory |
| FADD ST, ST(*i*) | Add real from stack |
| FADD ST(*i*), ST | Add real to stack |
| FADDP ST(*i*), ST | Add real and pop stack |
| FBLD *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 |
| 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 |

| Syntax | Action |
|---|---|
| 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) |
| 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 |

| Syntax | Action |
|---|---|
| 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 |
| 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 |



| Syntax | Action |
|---|---|
| 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 $\log_2(x+1)$ |




## B.4 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 |
| SAR *r/m*, *immed* | Shift arithmetic right immediate |
| SHL *r/m*, *immed* | Shift left immediate |
| SHR *r/m*, *immed* | Shift right immediate |

## B.5 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 |

## B.6 80286 Protected Instruction Mnemonics

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.

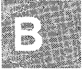


**Table B.6**

**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 |

## B.7 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) |

## B.8 80386 Nonprotected Instruction Mnemonics

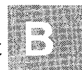
The 80386 nonprotected instruction set consists of all 8086 and 80286 nonprotected instructions plus the following instructions. The **.386** directive must be placed at the beginning of the source file to enable these instructions.

**Table B.8**

**80386 Nonprotected Instruction Mnemonics**

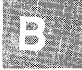
| Syntax | Action |
|---|---|
| BT *reg, reg* | Bit test |
| BT *mem, reg* | Bit test |
| BT *reg, immed* | Bit test |
| BT *mem, immed* | Bit test |
| BT *mem* | Bit test |
| BTC *reg, reg* | Bit test and complement |
| BTC *mem, reg* | Bit test and complement |
| BTC *reg, immed* | Bit test and complement |
| 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 |
| CMPSD | String compare doubleword |
| CWDE | Convert word in AX, doubleword in EAX |

| Syntax | Action |
|---|---|
| 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 |
| 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 |



| Syntax | Action |
|---|---|
| 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 |
| SETNC *r/m* | Set byte if not carry |
| SETNE *r/m* | Set byte if not equal |
| SETNG *r/m* | Set byte if greater |

| Syntax | Action |
|---|---|
| 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 |



## B.9 80386 Protected Instruction Mnemonics

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* |

## B.10 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

C.1 Introduction C-1

## C.1 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.

**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:

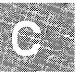
**Table C.2**

**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. |

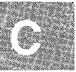
| Directive | Action |
|---|---|
| .286P | Enables assembly of 80286 protected instruction set and is equivalent to the following sequence: `.286` `.PRIV` |
| .287 | Enables assembly of 80287 instruction set. |
| .386 | Enables assembly of 80386 nonprotected instruction set and sets the default segment 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 80287 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. |
| COMMENT *delimiter text delimiter* | Treats all *text* between the given pair of *delimiter* delimiters as a comment. |

| Directive | Action |
|---|---|
| .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*. |
| 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*. |



| Directive | Action |
|---|---|
| *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. |
| *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. |

| Directive | Action |
|---|---|
| .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 a 8- or 16-bit record that contains one or more fields. |
| .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. |



| Directive | Action |
| --- | --- |
| *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

D.1 Introduction D-1

D.2 Text Segments D-2

D.3 Near Data Segments D-3

D.4 Far Data Segments D-4

D.5 BSS Segments D-5

D.6 Constant Segments D-6

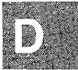
## D.1 Introduction

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 "Understanding Memory Models," and "Defining the Memory Model."

## D.2 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 "Associating Segments with Registers").

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 "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
```

D

## D.3 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.

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 "Associating Segments with Registers."

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 "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
```

## D.4 Far Data Segments

### Syntax

*name*_DATA SEGMENT WORD PUBLIC 'FAR_DATA'
*statements*
*name*_DATA ENDS

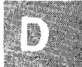
A "far" data segment contains data that is not pointed to by the **DS** segment register when the program starts execution. To access data in a far data segment, an explicit segment value must be given.

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 "Segment-Override Operator," and "Using Memory Operands."

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 "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
```



## D.5 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 "Associating Segments with Registers."

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.

### 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.6 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

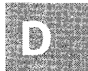
the predefined group **DGROUP**. Therefore, the following statements must appear at the beginning of the segment:

```
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 Section 4.4, "Associating Segments with Registers." 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.

### Example

```
DGROUP      group CONST
      ASSUME ds:DGROUP

CONST segment      word public 'CONST'
seg1  dw     ARRAY_DATA
seg2  dw     MESSAGE_DATA
CONST ends
```

# Appendix E

# Error Messages and Exit Codes


E.1 Introduction E-1

E.2 Messages and Exit Codes from masm E-1
- E.2.1 Assembler Status Messages E-1
- E.2.2 Numbered Assembler Messages E-2
- E.2.3 Unnumbered Error Messages E-18
- E.2.4 Exit Codes from masm E-20

## E.1 Introduction

This appendix lists and explains the messages and exit codes that can be generated by **masm**.

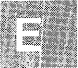
Messages are sent to the standard output device. By default, this device is the screen, but you can redirect the messages to a file or to a device such as a printer.

## E.2 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.

### E.2.1 Assembler Status Messages

After every assembly, **masm** reports on the symbol space, errors, and warnings. A sample display is shown below:



```
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981, 1987.  All rights reserved.


  47904 + 353887 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

The first line indicates how much near and far symbol space was unused during the assembly. This data may help you determine whether increasing the size of your program will exhaust available memory.

The first number indicates near symbol space. There is 64K total. 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.

### E.2.2 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 "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`

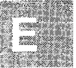
Nested procedures, segments, structures, macros, or repeat blocks were not properly terminated. This error may indicate that you closed an outer level of nesting with inner levels still open.

`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.

`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. Section 7.1, "Declaring Symbols Public," 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.

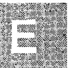
18 `Operand must be register:` *operand*

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.

`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 "Full Segment Definitions."

`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.

`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 "Arithmetic Operators."

`40 Operands must be in same segment, or one must be constant`
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 "Arithmetic Operators."

`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.

`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.

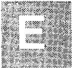
`53 Jump out of range by` *number* `bytes`

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. For more information, see "Forward References to Labels."

`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 "Passing Arguments on the Stack") 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 instruc-

tion, 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 "Strong Typing for Memory Operands."

`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 "Associating Segments with Registers," for information on the **ASSUME** directive.

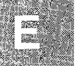
`65 Label cannot have segment override`

A segment override was used incorrectly.

`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 "Associating Segments with Registers," 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. ''Aligning Data,'' 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.

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 ''Arrays and Buffers.''

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 "Defining Structure Variables."

`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.

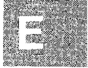
99 `Line too long expanding symbol:` *symbol*

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 "Aligning Data."

`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
```

With the 80386 processor, this 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.

E

`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.2.3 Unnumbered Error Messages

Unnumbered messages appear when an error occurs that cannot be associated with a particular line of code. Generally these errors indicate problems with the command line, memory allocation, or file access.

#### 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`

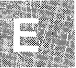
`Write error on object file`

## 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.

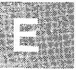
### E.2.4 Exit Codes from masm

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.



Replace this Page
with Tab Marked:

# Index

# Index


- & (ampersand), operator 10-15
- < > (angle brackets), operator 9-4
- * (asterisk), operator 8-3, 13-11
- @ (“at sign”) 3-5
- : (colon), operator
  - definition 8-10
- $ (dollar sign)
  - location counter symbol 5-20
  - symbol names, used in 3-5
- = (equal sign), directive 2-5, 7-4, 10-1
- ! (exclamation point), operator 10-18
- / (forward slash), operator 8-3
- - (minus sign), operator 8-3
- % (percent sign)
  - expression operator 10-19
  - symbol names, used in 3-5
- . (period) 3-5
- + (plus sign), operator 8-3
- ? (question mark) 3-5
- : (Segment-override operator)
  - definition 8-10
  - memory operands, with 13-4, 13-8
  - OFFSET operator, with 8-15
  - String instructions, with 17-2
  - XLAT instructions, with 14-3
- ;; (semicolons), operator 10-20
- _ (underscore) 3-5
- 10-byte temporary-real format 5-16
- 16-bit
  - addressing modes 13-11
  - segments 4-6, 4-18
- .186 directive 3-12
- .286P directive 3-12, 3-13, 19-2
- .287 directive 3-10, 3-13, 5-14, 18-11
- 32-bit
  - addressing modes 12-14, 13-11
  - segments 4-6, 4-18, 12-4, 14-13
- .386P directive 3-13, 4-6, 4-18, 19-2
- .387 directive 3-10, 3-14, 5-14, 18-11
- 80186 processor described 12-2
- 80286 processor described 12-2
- 80287 processor described 12-2
- 8036 processor
  - bytes, setting conditionally 16-15
- 80386 processor 12-13
  - 32-bit
    - addressing modes 12-14, 13-11

- 80386 processor 12-13 *(continued)*
  - 32-bit *(continued)*
    - pointers 5-11
    - registers 12-14
    - segments 4-6, 4-18, 12-4, 14-13
  - .386 directive 3-13, 4-18, 19-2
  - bit scan instructions 15-19
  - bit test instructions 15-18, 16-10
  - BSF instruction 15-19
  - BSR instruction 15-19
  - BT instruction 16-10
  - BTC instruction 16-10
  - BTR instruction 16-10
  - BTS instruction 16-10
  - CDQ instruction 14-5
  - CWDE instruction 14-5
  - data conversion 14-5, 14-6
  - described 12-2
  - double shifts 15-25
  - enhanced instructions 12-13
  - equate names 7-2
  - IMUL instruction 15-8
  - LFS instruction 14-9
  - LGS instruction 14-9
  - loading pointers 14-9
  - LSS instruction 14-9
  - MOVSX instruction 14-6
  - MOVZX instruction 14-6
  - new instructions 12-13
  - PUSHAD and POPAD instructions 14-15
  - PUSHD and POPD instructions 14-14
  - registers 12-5, 19-4
  - scaling 14-8
  - SETcondition instruction 16-15
  - SHLD instruction 15-25
  - SHRD instruction 15-25
  - simplified segment directives, with 4-6
  - special registers 19-4
- 80387 processor, described 12-2
- .8086 directive 3-12
- .8087 directive 3-10, 3-13, 5-14, 18-11
- 8087 processor described 12-2
- 8087/80287/80387 instruction set 2-3
- 8088/8086 processors described 12-1
- -087-family registers 12-13

## A

-a option 2-3, 4-16
AAA instruction 15-11
AAD instruction 15-12
AAM instruction 15-12
AAS instruction 15-12
ABS type 7-3
Absolute segments 4-21
Accumulator registers 12-9
ADC instruction 15-1, 15-3
ADD instruction 15-1, 15-3
Adding 15-1
Addition operator (+) 8-3
Addresses
  assembly listing 2-16
  effective 13-4, 13-8
Addressing modes
  16-bit 13-11
  32-bit 12-14
Adjusting masks 15-24
Advisory warnings 2-12
Aliases 10-4
ALIGN directive 5-21, 12-1
Align type 4-17, 4-22
Alignment, of segments 4-17, 5-21
.ALPHA directive 4-16
Ampersand (&), operator 10-15
AND instruction 15-14, 15-15, 16-9
AND operator 8-7
Angle brackets (< >>), operators 9-4
Arguments
  macros 10-6, 10-8, 10-24
  passing on stack 16-19
  repeat blocks 10-11
Arithmetic operators 8-3
Arrays
  boundary checking 16-27
  defining of 5-17
Assembler *See* masm
Assembly listing
  false conditionals 11-6
  macros 11-7
  page breaks 11-3
  page length 11-3
  page width 11-3
  Pass 1 2-5
  reading 2-15
  subtitle 11-3
  suppressing 11-5
  title 11-2
ASSUME directive 4-27, 4-28, 8-10
Asterisk (*), operator 8-3, 13-11
AT combine type 4-21
Auxiliary-carry flag 12-12
AX register 12-9

## B

-b option 2-4
Base registers 13-6, 13-11
Based operands 13-6
Based-indexed operands 13-6
BASIC
  modules called from 5-14
BASIC language, mentioned 16-2, 16-1
    16-16, 16-24
BCD (binary coded decimal) numbers
  calculations with 15-10, 18-17
  constants 3-9
  coprocessor, with 18-11
  defining of 5-9
  variables initialized 3-6
Binary coded decimals *See* BCD
Binary radix 3-7, 3-8
Binary to decimal conversion 15-13
Bit fields 6-1, 6-6
Bit mask 15-13, 16-9
Bit scan instructions 15-19
Bit test instructions 16-10
Bits, rotating 15-20
Bits, shifting 15-20
Bitwise operators 8-7
Boolean bit operations 15-14
BOUND instruction 16-27
Boundary-checking array 16-27
BP registers 12-9
BSF instruction 15-19
BSR instruction 15-19
BT instruction 16-10
BTC instruction 16-10
BTR instruction 16-10
BTS instruction 16-10
Buffers
  defining 5-17
  file, setting size 2-4
BYTE align type 4-17
BYTE type specifier 5-1

# C


C compiler 5-14
C language 4-2, 4-3
C language, mentioned 16-2, 16-12, 16-13, 16-16, 16-20, 16-24
Calculation operators 8-3
CALL instruction 5-4, 14-10, 16-16
Call tables 16-17
Carry flag 12-12, 15-2, 15-3, 15-5
Case
  emulating Pascal statement 16-2
CBW instruction 14-4
CDQ instruction 14-5
Character constant 3-10
Character set 3-4
Class type 4-23
Classical-stack operands, coprocessor 18-5
CLC instruction 15-3, 15-5
CLD instruction 17-1
CLI instruction 16-26
CMP instruction 16-4, 16-15
CMPS instruction 17-8
Code, assembly listing 2-15
CODE class name 4-4, 4-24
.CODE directive 4-7
@code equate 4-9
Code segments
  defining 4-7
  initializing 4-31
  register 12-8
@CodeSize equate 4-9
Combine type 4-19, 4-22
COMENT object record 4-5
COMM directive 7-1, 7-8
Command lines
  with masm 2-1
Command-line help 2-7
COMMENT directive 3-3
Comments, writing 3-3
COMMON combine type 4-20
Communal symbols 7-1, 7-8
Compact memory model 4-3, 4-5
Compare instructions 18-25
Comparing register to zero 15-16
Comparing strings 17-8
Compatibility
  other assemblers A-5
  upward 12-1
Conditional directives
  assembly directives 2-13, 9-0, 9-1, 10-8
  assembly passes 9-3, 9-7
  error directives 9-0, 10-8
Conditional directives *(continued)*
  macro arguments 9-4, 9-5, 9-9, 9-10
  nesting 9-2
  operators 10-15
  symbol definition 9-3, 9-9
  value of true and false 9-2, 9-8
Conditional-error directives 9-6
Conditional-jump instructions 16-3, 18-25
.CONST directive 4-7
Constants 3-6, 13-1, 15-22
Control data, coprocessor 18-16
Conversion, binary to decimal 15-13
Converting data sizes 14-4
Coprocessor
  8086 family 12-2
  architecture 18-1
  control data 18-16
  directives 3-12
  emulator 2-6
  loading data 18-11
  loading pi 18-15
  no-wait instructions 18-30
  operands 18-4
  -r options 2-3
  registers 12-13
Copying data 14-1
CREF
  directive (.CREF) 11-9
Cross-reference files
  comparing with listing 2-16
CS: override 2-10
CS Register 12-8
CurSeg equate 4-9
CWD instruction 14-4
CWDE instruction 14-5
CX Register 12-9


# D


-d option 2-5, E-3
DAA instruction 15-13
DAS instruction 15-13
Data bus 12-1
Data conversion 14-4
.data directive 1-5
.DATA? directive 4-7
@data equate 4-9
Data segments
  defining 4-7
  developing programs 1-5
  initializing 4-32
  registers 12-8

Data-definition directives 5-5
DataSize equate 4-5, 4-9
@DataSize equate 4-5, 4-9
DB directive 5-6, 5-7, 5-10
DD directive 5-6, 5-7
DEC instruction 15-3, 15-4
Decimal, packed BCD numbers 15-11
Decimal radix 3-7, 3-8
Decrementing 15-3
Defaults
  radix 3-8
  segment names 4-7, 4-12
  segment registers 4-28
  simplified segment 4-11
  types 8-26
Defining symbols from command line 2-5
Destination string 17-2
Development cycle 1-1
DF directive 5-6, 5-7
DGROUP group name
  COMM directive, with 7-10
  simplified segments, with 4-4, 4-8, 4-11
Direction flag 12-12, 17-1
Directives
  .186 3-12
  .286 3-12, 19-2
  .286P 3-13
  .287 3-10, 3-13, 5-14, 18-11
  .386 3-13, 4-6, 4-18, 19-2
  .386P 3-13
  .387 3-10, 3-14, 5-14, 18-11
  .8086 3-12
  .8087 3-10, 3-13, 5-14, 18-11
  ALIGN 5-21, 12-1
  .ALPHA 4-16
  ASSUME 4-27, 4-28, 8-10
  .CODE 4-7
  COMM 7-1, 7-8
  COMMENT 3-3
  .CONST 4-7
  .CREF 11-9
  .DATA 1-5
  .DATA? 4-7
  data definition 5-5
  DB 5-6, 5-7, 5-10
  DD 5-6, 5-7
  defined 3-3
  DF 5-6, 5-7
  DOSSEG 4-3
  DQ 5-6, 5-7, 5-12
  DT 5-6, 5-7, 5-12
  DW 5-6, 5-7, 5-11
  ELSE 9-2
  END 3-15, 4-7, 4-31
Directives *(continued)*
  ENDIF 9-2
  ENDM 10-6, 10-11, 10-12, 10-13
  ENDP 5-4, 16-17, 16-27
  ENDS 4-15, 4-16, 6-2
  EQU 2-16, 7-4, 10-2, 10-4
  equal sign (=) 2-5, 7-4, 10-1
  .ERR 9-7
  .ERR1 9-7
  .ERR2 9-7
  .ERRB 9-9
  .ERRDEF 9-9
  .ERRDIF 9-10
  .ERRE 9-8
  .ERRIDN 9-10
  .ERRNB 9-9
  .ERRNDEF 9-9
  .ERRNZ 9-8
  EVEN 5-21, 12-1
  EXITM 10-11
  EXTRN 5-3, 7-1, 7-3
  .FARDATA? 4-7
  full segment 4-1
  functions C-0
  global 7-0, 7-6
  GROUP 4-2, 4-26, 8-10
  IF 2-13, 9-2
  IF1 9-3, 11-1
  IF2 9-3, 11-1
  IFB 9-4
  IFDEF 9-3
  IFDIF 9-5
  IFE 9-2
  IFIDN 9-5
  IFNB 9-4
  IFNDEF 9-3
  INCLUDE 10-5, 10-25, 10-26
  instruction set 3-12
  IRP 10-13
  IRPC 10-13
  LABEL 5-5, 5-20
  .LALL 10-9, 11-7
  .LFCOND 2-13, 11-6
  .LIST 11-5
  LOCAL 10-9, 10-11
  MACRO 10-6
  .MODEL 3-11, 4-5, 7-4
  .MSFLOAT 3-12, 5-14
  NAME 7-8
  ORG 4-31, 5-20
  %OUT 11-1
  PAGE 11-3
  PROC 4-11, 5-3, 16-16, 16-27
  PUBLIC 5-3, 5-4, 7-1

Directives *(continued)*
PURGE 10-26
.RADIX 3-8
RECORD 6-6
REPT 10-12
.SALL 10-9, 11-7
SEGMENT 4-15, 4-16, 8-10
.SEQ 4-16
.SFCOND 2-13, 11-6
simplified segment 4-1
.STACK 4-7
STRUC 6-2
SUBTTL 11-3
summary C-0
syntax C-0
.TFCOND 2-13, 11-6
TITLE 7-8, 11-2
.XALL 10-9, 11-7
.XCREF 11-9
.XLIST 11-5
Displacement 13-6
DIV instruction 15-9
Dividing 15-9
Dividing by constants 15-22
Division operator (/) 8-3
Do
emulating C statement 16-13
emulating FORTRAN statement 16-12
Dollar sign ($)
location counter symbol 5-20
symbol names, used in 3-5
DOSSEG directive 4-3
-dosseg linker option 4-5
Double shifts, with 80386 processor 15-25
DQ directive 5-6, 5-7, 5-12
DS registers 12-8
-Dsymbol option 2-5
DT directive 5-6, 5-7, 5-12
DT Register 12-10
Dummy parameters
macros 10-6, 10-8, 10-24
repeat blocks 10-11
Dummy segment definitions 4-25
DUP operator 5-17, 6-2, 6-3, 6-8
DW directive 5-6, 5-7, 5-11
DWORD align type 4-18
DWORD type specifier 5-1
DX Registers 12-9

# E

-e option 2-6, 5-14
Effective address 13-4, 13-8
ELSE directive 9-2
Emulator, coprocessor 2-6
Encoded real numbers 3-9, 5-14
Encoding of instructions 13-1
END directive 3-15, 4-7, 4-31
ENDIF directive 9-2
ENDM directive 10-6, 10-11, 10-12, 10-13
ENDP directive 5-4, 16-17, 16-27
ENDS directive 4-15, 4-16, 6-2
ENTER instruction 16-23
EQ operator 8-8
EQU directive 2-16, 7-4, 10-2, 10-4
Equal sign (=), directive 2-5, 7-4, 10-1
Equates
defined 10-0, 10-1
nonredefinable 10-2
predefined 4-9
redefinable 10-1
string 10-4
.ERR directive 9-7
.ERR1 directive 9-7
.ERR2 directive 9-7
.ERRB directive 9-9
.ERRDEF directive 9-9
.ERRDIF directive 9-10
.ERRE directive 9-8
.ERRIDN directive 9-10
.ERRNB directive 9-9
.ERRNDEF directive 9-9
.ERRNZ directive 9-8
Error lines, displaying 2-14
Error messages
assembly listing 2-15, 2-16
masm E-3
ES registers 12-8
ESC instruction 19-2
EVEN directive 5-21, 12-1
Exclamation point (!), operator 10-18
Exit codes
masm E-20
EXITM directive 10-11
Exponent, part of real-number constant 3-9
Exponentiation, with087-family coprocessors 18-28
Expression operator (%) 10-19
Expressions, defined 8-0
External names 2-9

External symbols 7-3
Extra segment 12-8
EXTRN directive 5-3, 7-1, 7-3

## F

F2XM1 instruction 18-29
FABS instruction 18-22
FADD instruction 18-17
FADDP instruction 18-18
False conditionals, listing 2-13, 11-6
Far pointers 5-11, 14-8
FAR type specifier 5-2
.FARDATA? directive 4-7
@fardata? equate 4-9
Fatal errors 9-7
FBLD instruction 18-13
FBSTP instruction 18-13
FCHS instruction 18-22
FCOM instruction 18-26
FCOMP instruction 18-26
FCOMPP instruction 18-27
FCOS instruction 18-29
FDIV instruction 18-20
FDIVP instruction 18-21
FDIVR instruction 18-21
FDIVRP instruction 18-21
FIADD instruction 18-18
FICOM instruction 18-26
FICOMP instruction 18-27
FIDIV instruction 18-21
FIDIVR instruction 18-21
Fields
  assembler statements 3-1
  bit 6-1, 6-6
  records 6-6, 6-10
  structures 6-2, 6-4
FILD instruction 18-13
fileName equate 4-9
Files
  buffer 2-4
  include 2-8, 7-11, 10-25
  listing 2-1, 2-9, 11-2
  source *See* Source files
  specifications 10-25
Filling strings 17-10
FIMUL instruction 18-20
FINIT instruction 18-30
First-in-first-out (FIFO) 14-10
FIST instruction 18-13
FISTP instruction 18-13
FISUB instruction 18-18
FISUBR instruction 18-19
Flags
  loading and storing 14-3
  register 12-11
FLD instruction 18-12
FLD1 instruction 18-15
FLDCW instruction 18-16
FLDL2E instruction 18-15
FLDL2T instruction 18-16
FLDLG2 instruction 18-16
FLDLN2 instruction 18-16
FLDPI instruction 18-15
FLDZ instruction 18-15
Floating-point format
  compatibility A-5
Floating-point numbers 2-3, 2-6
FMUL instruction 18-20
FMULP instruction 18-20
For, emulating high-level-language st
    16-12
FORTRAN compiler 5-14
FORTRAN language, mentioned 16-1
    16-16, 16-24
Forward references
  defined 8-21
  during a pass 2-23
  labels 8-22
  variables 8-24
Forward slash (/), operator 8-3
FPATAN instruction 18-29
FPREM instruction 18-22, 18-28
FPTAN instruction 18-29
Fraction 3-9
FRNDINT instruction 18-22
FS registers 12-8
FSCALE instruction 18-22
FSIN instruction 18-30
FSINCOS instruction 18-29
FSQRT instruction 18-22
FST instruction 18-12
FSTCW instruction 18-16
FSTP instruction 18-12
FSTSW instruction 18-16, 18-17
FSUB instruction 18-18
FSUBP instruction 18-19
FSUBR instruction 18-19
FSUBRP instruction 18-19
FTST instruction 18-25, 18-26
Full segment directives 4-1
Functions
  C 16-16
  Pascal 16-16
FWAIT instruction 18-10
FWORD type specifier 5-1

FXAM instruction 18-28
FXCH instruction 18-12
FXTRACT instruction 18-22
FYL2X instruction 18-29
FYL2XP1 instruction 18-29

# G

GE operator 8-8
General-purpose registers 12-8
Getting strings from ports 17-12
Global directives
  defined 7-0
  illustrated 7-6
Global scope 7-1
Global symbols 7-1, 7-3
GROUP directive 4-2, 4-26, 8-10
Group-relative segments 4-27
Groups
  assembly listing 2-20
  defined 4-26
  illustrated 4-27
  size restriction 4-26
GS Registers 12-8
GT operator 8-8

# H

-h option 2-7
Hardware interrupts 16-26
Help 2-7
Hexadecimal radix 3-7, 3-8
HIGH operator 8-13
High-level languages, memory model 4-2, 4-5
HLT instruction 19-2
Huge memory model 4-3, 4-5

# I

-I option 2-8, 10-25
IDIV instruction 15-9
IEEE format 3-10, 5-13, 5-14, 18-11
IF directives 2-13, 9-2
IF1 directive 9-3, 11-1
IF2 directive 9-3, 11-1
IFB directive 9-4
IFDEF directive 9-3
IFDIF directive 9-5
IFE directive 9-2
IFIDN directive 9-5
IFNB directive 9-4
IFNDEF directive 9-3
Immediate operands 13-1
Implied operands 18-5
Impure code, checking for 2-10
IMUL instruction 15-6, 15-7, 15-8
IN instruction 14-15
INC instruction 15-1
INCLUDE directive 10-5, 10-25, 10-26
Include files 10-25
  assembly listings 2-16
  communal variables 7-11
  setting search paths 2-8
  using 10-25
Incrementing 15-1
Indeterminate operand 5-19
Index checking 16-27
Index operator 8-5
Index registers 13-6, 13-11
Indexed operands 13-6
Initializing
  segment registers 4-31
  variables 5-6
INS instruction 17-12
Instruction sets
  80186 processor B-13
  80286 processor B-14, B-15
  80287 coprocessor B-15
  80386 processor B-16, B-19
  8086 processor B-2
  8087 coprocessor B-8
  assembly-language programs, used with 1-1
  Intel family B-1
Instruction-pointer register (IP) 12-11, 16-1
Instructions
  AAA 15-11
  AAD 15-12
  AAM 15-12
  AAS 15-12
  ADC 15-1, 15-3
  ADD 15-1, 15-3
  AND 15-14, 15-15, 16-9
  bit scan 15-19
  bit test 15-18, 16-10
  BOUND 16-27
  BSF 15-19
  BSR 15-19
  BT 16-10
  BTC 16-10
  BTR 16-10
  BTS 16-10

Instructions *(continued)*
- CALL 5-4, 14-10, 16-16
- CBW 14-4
- CDQ 14-5
- CLC 15-3, 15-5
- CLD 17-1
- CLI 16-26
- CMP 16-4, 16-15
- CMPS 17-8
- compare 18-25
- conditional jump 16-1, 18-25
- CWD 14-4
- CWDE 14-5
- DAA 15-13
- DAS 15-13
- DEC 15-3, 15-4
- defined 3-3
- DIV 15-9
- ENTER 16-23
- ESC 19-2
- F2XM1 18-29
- FABS 18-22
- FADD 18-17
- FADDP 18-18
- FBLD 18-13
- FBSTP 18-13
- FCHS 18-22
- FCOM 18-26
- FCOMP 18-26
- FCOMPP 18-27
- FCOS 18-29
- FDIV 18-20
- FDIVP 18-21
- FDIVR 18-21
- FDIVRP 18-21
- FIADD 18-18
- FICOM 18-26
- FICOMP 18-27
- FIDIV 18-21
- FIDIVR 18-21
- FILD 18-13
- FIMUL 18-20
- FINIT 18-30
- FIST 18-13
- FISTP 18-13
- FISUB 18-18
- FISUBR 18-19
- FLD 18-12
- FLD1 18-15
- FLDCW 18-16
- FLDL2E 18-15
- FLDL2T 18-16
- FLDLG2 18-16
- FLDLN2 18-16

Instructions *(continued)*
- FLDPI 18-15
- FLDZ 18-15
- FMUL 18-20
- FMULP 18-20
- FPATAN 18-29
- FPREM 18-22, 18-28
- FPTAN 18-29
- FRNDINT 18-22
- FSCALE 18-22
- FSIN 18-30
- FSINCOS 18-29
- FSQRT 18-22
- FST 18-12
- FSTCW 18-16
- FSTP 18-12
- FSTSW 18-16, 18-17
- FSUB 18-18
- FSUBP 18-19
- FSUBR 18-19
- FSUBRP 18-19
- FTST 18-25, 18-26
- FWAIT 18-10
- FXAM 18-28
- FXCH 18-12
- FXTRACT 18-22
- FYL2X 18-29
- FYL2XP1 18-29
- HLT 19-2
- IDIV 15-9
- IMUL 15-6, 15-7, 15-8
- IN 14-15
- INC 15-1
- INS 17-12
- INT 13-2, 14-10, 16-25, 16-27
- INTO 16-25, 16-26
- IRET 14-10, 16-26, 16-27
- IRETD 16-27
- JC 15-2, 15-5
- Jcondition 16-5, 16-7, 16-9, 16-26
- JCXZ 16-4, 16-14, 17-8, 17-9
- JECXZ 16-13
- JMP 4-28, 8-22, 16-1
- LAHF 14-3
- LDS 14-8
- LEA 14-7
- LEAVE 16-23
- LES 14-8, 17-8
- LFS 14-9
- LGS 14-9
- LOCK 19-2
- LODS 17-11
- logical 15-15
- LOOP 16-12

Instructions *(continued)*
LOOPE 16-13
LOOPNE 16-13
LOOPNZ 16-13
LOOPZ 16-13
LSS 14-9
MOV 4-28, 14-1, 19-4
MOVS 17-5
MOVSX 14-6
MOVZX 14-6
MUL 15-6
NEG 15-3, 15-4
NOP 8-22, 19-1
NOT 15-18
OR 15-14, 15-16
OUT 14-15
OUTS 17-12
POP 4-28, 14-10
POPA 14-14
POPAD 14-15
POPD 14-14
POPF 14-14
POPFD 14-14
program-flow 16-0
protected mode 19-3
PUSH 4-28, 14-10
PUSHA 14-14
PUSHAD 14-15
PUSHD 14-14
PUSHF 14-14
PUSHFD 14-14
RCL 15-21
RCR 15-21
REP 17-3, 17-10, 17-12
REPE 17-3, 17-8, 17-9
REPNE 17-3, 17-7, 17-9
REPNZ 17-3, 17-7, 17-9
REPZ 17-3, 17-8, 17-9
RET 5-4, 13-2, 14-10, 16-19
RETF 16-18
RETN 16-18
ROL 15-21
ROR 15-21
SAHF 14-3
SAL 15-21
SAR 15-21
SBB 15-3, 15-5
SCAS 17-7
SETcondition 16-15
SHL 15-21
SHLD 15-25
SHR 15-21
SHRD 15-25
STD 17-1

Instructions *(continued)*
STI 16-26
STOS 17-10
SUB 15-3, 15-4, 15-5, 16-6
TEST 16-4, 16-9, 16-15
timing of 13-1
WAIT 18-10, 19-2
XCHG 14-2
XLAT 14-2
XOR 15-14, 15-17
Instruction-set directives 3-12
INT instruction 13-2, 14-10, 16-25, 16-27
Integers 3-6, 18-17
Integers, with coprocessor 18-11
Interrupt-enable flag 12-12, 16-25
Interrupts 16-25
INTO instruction 16-25, 16-26
I/O protection level flag 12-12
IP Registers 12-11
IRET instruction 14-10, 16-26, 16-27
IRETD instruction 16-27
IRP directive 10-13
IRPC directive 10-13

## J

JC instruction 15-2, 15-5, 16-7
Jcondition instruction 16-5, 16-7, 16-9, 16-26
JCXZ instruction 16-4, 16-14, 17-8, 17-9
JECXZ instruction 16-13
JMP instruction 4-28, 8-22, 16-1
JO 15-2, 16-7, 16-26
Jump tables 16-2
Jumping conditionally 16-3

## L

-l option 2-9
LABEL directive 5-5, 5-20
Labels
defined 5-2
macros, in 10-10
near code 5-2
procedures 5-3
LAHF instruction 14-3
.LALL directive 10-9, 11-7
Large memory model 4-3, 4-5
ld
development cycle, in 1-3

LDS instruction 14-8
LE operator 8-8
LEA instruction 14-7
LEAVE instruction 16-23
LENGTH operator 8-17
LES instruction 14-8, 17-8
.LFCOND directive 2-13, 11-6
LFS instruction 14-9
LGS Instruction 14-9
Line number data 2-14
.LIST directive 11-5
Listing
  false conditionals 11-6
  files 2-1, 2-9, 11-2
  format
    addresses 2-16
    code 2-15
    described 2-15
    EQU directive 2-16
    errors 2-15, 2-16
    groups 2-20
    include files 2-16
    LOCK directive 2-16
    macro expansions 2-16
    macros 2-18
    Pass 1, reading 2-23
    records 2-18
    REP directive 2-16
    segment override 2-16
    segments 2-20
    structures 2-18
    symbols 2-21
  macros 11-7
  Pass 1,creating 2-5
  subtitles in 11-3
  suppressing output 11-5
  suppressing tables 2-10
  tables, suppressing 2-10
Literal-character operator (!) 10-18
Literal-text operator (< >>) 9-4
Loading constants to coprocessor 18-15
Loading coprocessor data 18-11
Loading pointers 14-9
Loading values from strings 17-11
LOCAL directive 10-9, 10-11
Local scope 7-1
Local symbols in macros 10-9
Local variables, in procedures 16-21
Location counter 5-1, 5-20, 5-21, 8-20
Location counter symbol 5-20
LOCK directive, assembly listing 2-16
LOCK instruction 19-2
LODS instruction 17-11
Logarithms 18-28
Logical bit operations 15-14
Logical instructions 15-15
Logical operators 15-15
Loop
  while equal 16-13
  while not equal 16-13
LOOP instruction 16-12
LOOPE instruction 16-13
LOOPNE instruction 16-13
LOOPNZ instruction 16-13
LOOPZ instruction 16-13
LOW operator 8-13
LSS instruction 14-9
LT operator 8-8

# M

Macro Assembler *See* masm
Macro comment operator (;;) 10-20
MACRO directive 10-6
Macro expansions, assembly listings 2-1
Macros
  argument testing 9-5, 9-10
  arguments 10-6, 10-8, 10-24
  assembly listing 2-18
  calling 10-8
  communal variables 7-11
  compared to procedures 10-6
  defined 10-0, 10-5
  efficiency penalty 10-1
  exiting early 10-11
  expansions in listing 11-7
  local symbols 10-9
  nested 10-16, 10-21
  operators 10-15
  parameters 10-6, 10-8, 10-24
  recursive 9-5, 10-21
  redefining 10-23, 10-26
  removing from memory 10-26
  text 10-4
make, in development cycle 1-3
MASK operator 6-12
Masking bits 15-14, 16-9
masm
  command line 2-1
  described 2-0
  development cycle, in 1-3
  error messages E-3
  executable files 1-1
  exit codes E-20
  invoking 2-1
  summary 1-5

Math coprocessors 2-3, 12-2, 18-1
Medium memory model 4-2, 4-5
Memory access, coordinating 18-9
MEMORY combine type 4-20
Memory models 4-2
Memory operands 13-4
Memory operands, coprocessor 18-6
Messages
  status E-1
  suppressing 2-11
Messages to Standard Output 11-1
Microsoft Binary format 5-13, 5-14
Microsoft Binary Real format 3-10, 18-11
Minus operator (-) 8-3
Mixed-languages programs 4-1, 4-16
-Ml option 2-9
-Ml option, masm 4-23
Mnemonics
  defined 3-3
  reserved names, as 3-6
MOD operator 8-3
.MODEL directive 3-11, 4-5, 7-4
Modes, addressing *See* Addressing modes
Modular programming 7-0
Modulo division 18-22
Modulo division operator 8-3
MOV instruction 4-28, 14-1, 19-4
Moving strings 17-5
MOVS instruction 17-5
MOVSX instruction 14-6
MOVZX instruction 14-6
.MSFLOAT directive 3-12, 5-14
-Mu option 2-9
MUL instruction 15-6
Multiple modules 7-6
Multiplication operator (*) 13-11
Multiplication operators 8-3
Multiplying 15-6
Multiplying by constants 15-22
Multiword values, shifting 15-24
-Mx option 2-9
-Mx option, masm 4-23

## N

-n option 2-10
NAME directive 11-2
Names
  Assigning 3-4
  external 2-9
  public 2-9
  reserved 3-5, 10-24, 10-26
NE operator 8-8
Near pointers 5-11, 14-7
NEAR type specifier 5-2
NEG instruction 15-3, 15-4
Negating 15-4
Nested-task flag 12-13
Nesting
  conditionals 9-2
  DUP operators 5-18
  include files 10-25
  macros 10-16, 10-21
  procedures for Pascal 16-24
  segments >, 4-35"
New features A-0
Nonredefinable equates 10-2
NOP instruction 8-22, 19-1
NOT instruction 15-18
NOT operator 8-7
NOTHING, ASSUME 4-29
No-wait coprocessor instructions 18-30
Null class type 4-24
Null string 10-8
Numbers *See* Real numbers, signed numbers, etc.

## O

Object records 4-3
Octal radix 3-7, 3-8
OFFSET operator 4-11, 8-14
OFFSET operator, with group-relative segmen 4-27
ON GOSUB, emulating BASIC statement 16-2
Opcode *See* Instructions
Operands
  based 13-6
  based indexed 13-6
  based indexed with displacement 13-6
  classical stack 18-5
  coprocessor 18-4, 18-5
  defined 3-3, 8-0, 13-0
  immediate 13-1
  implied 18-5
  indeterminate 5-19
  indexed 13-6
  indirect memory 13-1, 13-4, 13-6
  location counter 8-20
  memory 13-1, 13-4, 13-6
  record field 6-13
  records 6-10
  register 12-5, 13-1, 13-2
  register indirect 13-6

Operands *(continued)*
relocatable 13-4
strong typing 8-25
structures 6-5
undefined 5-19
Operating system 1-1
Operators
addition 8-3
AND 8-7
arithmetic 8-3
bitwise 8-7
calculation 8-3
defined 8-1
division (/) 8-3
DUP 5-17, 6-2, 6-3, 6-8
EQ 8-8
expression (%) 10-19
GE 8-8
GT 8-8
HIGH 8-13
index 8-5
LE 8-8
LENGTH 8-17
literal character (!) 10-18
logical 15-15
LOW 8-13
LT 8-8
macro comment (;;) 10-20
MASK 6-12
minus (.) 8-3
MOD 8-3
multiplication (*) 8-3
NE 8-8
NOT 8-7
OFFSET 4-11, 8-14
OR 8-7
plus (+) 8-3
precedence 8-19
PTR 8-11, 8-23
relational 8-8
SEG 4-26, 7-10, 8-14
segment override (:) 2-16, 13-8
segment override (:) *See* : (Segment-override operator)
shift 8-7
SHL 8-7
SHORT 8-12, 8-22
SHR 8-7
SIZE 8-18
structure-field name 8-5
substitute (&) 10-15
subtraction 8-3
THIS 8-13
.TYPE 8-15
Operators *(continued)*
TYPE 8-16
WIDTH 6-12
XOR 8-7
Options
-a 2-3, 4-16
-b 2-4
-d 2-5, E-3
-dosseg linker 4-5
-Dsymbol 2-5
-e 2-6, 5-14
-h 2-7
-I 2-8, 10-25
-l 2-9
-Ml 2-9
-Ml, masm 4-23
-Mu 2-9
-Mx 2-9
-Mx, masm 4-23
-n 2-10
-p 2-10
-r 2-3
-s 2-3, 4-16
summary 2-2
-t 2-11, E-2
using 2-1
-v 2-11, E-1
-w 2-12, 8-26
-X 2-13
-x 2-14, 11-6
-z 2-14
-Zd 2-14
-Zi 2-14
OR instruction 15-14, 15-16
OR operator 8-7
ORG directive 4-31, 5-20
%OUT directive 11-1
OUT instruction 14-15
Output messages to Standard Output 11-
OUTS instruction 17-12
Overflow flag 12-12, 15-2
Override
CS: 2-10

# P

-p option 2-10
Packed BCD numbers 5-10, 15-11, 15-1
Packed decimal integers 3-6
Packed decimal numbers 3-9
PAGE align type 4-18
Page breaks in assembly listings 11-3

PAGE directive 11-3
Page format of listing files 11-2
PARA align type 4-18
Parameters
  defining in procedures 16-19
  macros 10-6, 10-8, 10-24
  repeat blocks 10-11
Parity flag 12-12
Partial remainder 18-22
Pascal compiler 5-14
Pascal language, mentioned 16-2, 16-12, 16-13, 16-16, 16-24
Pass 1 listing 2-5, 2-23
Percent sign (%)
  expression operator 10-19
  symbol names, used in 3-5
Period (.) 3-5
Phase errors 2-5, 2-23
Pi, loading to coprocessor 18-15
Plus sign (+), operator 8-3
Pointers
  defining 5-11
  loading 14-7
POP instruction 4-28, 14-10
POPA instruction 14-14
POPAD instruction 14-15
POPD instruction 14-14
POPF instruction 14-14
POPFD instruction 14-14
Ports
  defined 14-15
  getting strings from 17-12
  sending strings to 17-12
Precedence of operators 8-19
Preserving case sensitivity 2-9
PRIVATE combine type 4-21
PROC directive 4-11, 5-3, 16-17, 16-27
PROC type specifier 5-2, 7-4
Procedures
  compared to macros 10-6
  defining labels 5-3
  Pascal 16-16
  using 16-16
Processor directives 3-12
Processors *See* Coprocessors
Program-development cycle 1-1
Program-flow instructions 16-0
Protected mode 12-2, 12-3, 18-31
Protected-mode instructions 19-3
Pseudo-op *See* Directives
PTR operator 8-11, 8-23
PUBLIC combine type 4-19
PUBLIC directive 5-3, 7-1
Public names 2-9
Public symbols 7-1
PURGE directive 10-26
PUSH instruction 4-28, 14-10
PUSHA instruction 14-14
PUSHAD instruction 14-15
PUSHD instruction 14-14
PUSHF instruction 14-14
PUSHFD instruction 14-14

# Q

Question mark (?) 3-5
QWORD type specifier 5-1

# R

-r option 2-3
.RADIX directive 3-8
Radixes
  binary 3-7, 3-8
  default 3-8
  specifiers 3-7
RCL instruction 15-21
RCR instruction 15-21
Real mode 12-1, 12-3, 19-1
Real numbers
  arithmetic calculations 18-17
  coprocessor 18-11
  designator (R) 5-12
  encoding 3-9, 5-14
  format 2-3, 2-6, 3-9
  format, compatibility A-5
RECORD directive 6-6
Record type 6-6
Records
  assembly listing 2-18
  declarations 6-6
  defining 6-1, 6-8
  field operands 6-13
  fields 6-10
  initializing 6-6, 6-8, 6-10
  MASK operator 6-12
  object 4-3
  operands 6-10
  variables 6-8
  WIDTH operator 6-12
Recursive macros 9-5, 10-21
Redefinable equates 10-1
Redefining interrupts 16-27

Redefining macros 10-23
Registers
  80386 12-5
  80386, special 19-4
  8087 family 12-13
  accumulator 12-9
  AX 12-9
  base 13-6, 13-11
  BP 12-9
  BX 12-9
  coprocessor 12-13, 18-2, 18-3
  CS 12-8
  CX 12-9
  DI 12-10
  DS 12-8
  DX 12-9
  ES 12-8
  flags 12-11
  FS 12-8
  general purpose 12-8
  GS 12-8
  index 13-6, 13-11
  IP 12-11, 16-1
  mixing 16-bit and , 32-bit , 13-12
  operands 12-5, 13-1, 13-2
  operands, coprocessor 18-7
  register-pop operands, coprocessor 18-8
  reserved names, as 3-6
  segment 4-31, 12-8
  SI 12-10
  SP 12-10
  special 19-4
  SS 12-8
Relational operators 8-8
Relocatable operands *See* Memory operands
REP directive, assembly listing 2-16
REP instruction 17-3, 17-10, 17-12
REPE instruction 17-3, 17-8, 17-9
Repeat blocks
  arguments 10-11
  defined 10-0, 10-11
  parameters 10-11
  repeat for each argument 10-13
  repeat for each character of string 10-13
  repeat for specified count 10-12
Repeat, emulating Pascal statement 16-13
Repeat, using086-family string functions 17-0
REPNE instruction 17-3, 17-7, 17-9
REPNZ instruction 17-3, 17-7, 17-9
REPT directive 10-12
REPZ instruction 17-3, 17-8, 17-9
Reserved names 3-5, 10-24, 10-26
Resume flag 12-13
RET instruction 5-4, 13-2, 14-10, 16-16
RETF instruction 16-18
RETN instruction 16-18
ROL instruction 15-21
ROR instruction 15-21
Rotating bits 15-20
Routines, FORTRAN 16-16


# S


-s option 2-3, 4-16
ASCII
  format for text files 1-4
  name for unpacked BCD numbers 15-
MS-DOS
  80386 under 12-13
  segment-order convention 4-3
MS-DOS compatibility
  -l 2-9
  -Ml 2-9
  pathnames, with (backslash) 10-25
  -s 2-4
XENIX 12-3
SAHF instruction 14-3
SAL instruction 15-21
.SALL directive 10-9, 11-7
SAR instruction 15-21
SBB instruction 15-3, 15-5
Scaling 14-8
Scaling by powers of two 18-22
Scaling factor 13-11
SCAS instruction 17-7
Search paths
  include files 10-25
  setting 2-8
Searching strings 17-7
Sections in assembly listings 11-3, 11-4
SEG operator 4-26, 7-10, 8-14
SEGMENT directive 4-15, 4-16, 8-10
Segment-order method 4-15
Segments
  16-bit 4-6, 4-18
  32-bit 4-6, 4-18
  32-bit 12-4
  32-bit 14-13
  absolute 4-21
  alignment 4-17, 5-21
  assembly listing 2-20
  combine types 4-19
  defined 4-1
  definition 4-15
  extra 12-8
  group-relative offset 4-27

Segments *(continued)*
  groups 4-26
  MEMORY 4-20
  nesting 4-35
  ordering 2-3, 4-24
  override, assembly listings 2-16
  override operator (:) 13-8
  override operator (:) *See* : (Segment-override operator)
  registers 12-8
  selectors 12-4
  size 4-18
  types 4-17
Selectors, segment 12-4
Semicolons (;;), operator 10-20
Sending strings to ports 17-12
.SEQ directive 4-16
Serious warnings 2-12
SET condition instruction 16-15
Setting file buffer size 2-4
Setting register to zero 15-17
Severe errors 2-12, 9-7
.SFCOND directive 2-13, 11-6
Shift operators 8-7
Shifting bits 15-20
Shifting multiword values 15-24
SHL instruction 15-21
SHL operator 8-7
SHLD instruction 15-25
SHORT operator 8-12, 8-22
SHR instruction 15-21
SHR operator 8-7
SHRD instruction 15-25
SI registers 12-10
Sign flag 12-12, 15-5
Signed numbers 5-6, 14-4, 15-2, 15-4
Sign-extending 14-6
Simplified segment defaults 4-11
Simplified segment directives 4-1
SIZE operator 8-18
Small memory model 4-2, 4-5
Source files
  compatibility
    high-level languages D-0
    memory models D-0
  defined 1-4
  format 3-1
  illustrated 1-4
  include 10-25
  segments D-0
Source modules 1-3, 7-0
Source string 17-2
SP registers 12-10
Special registers 19-4
Square root 18-22
SS registers 12-8
Stack
  defined 14-10
  frame 16-23
  operands, coprocessor 18-5
  registers 18-4
  segment 4-7, 4-20, 12-8
  segment, initializing 4-33
  use of 14-13
STACK combine type 4-20
.STACK directive 4-7
Standard output device 11-1, E-1
Statement fields 3-1
Statements, defined 3-1
Statistics 2-11, E-1
Status messages E-1
STD instruction 17-1
STI instruction 16-26
Storing coprocessor data 18-11
STOS instruction 17-10
Strict type checking A-5
Strings
  comparing 17-8
  constants 3-10, 13-1
  defined 17-0
  destination strings 17-2
  equates 10-4
  filling 17-10
  getting from ports 17-12
  loading values from 17-11
  moving 17-5
  null 10-8
  ports, transfer from and to 17-12
  searching 17-7
  source 17-2
  structures, in 6-2
  variables 5-10
Strong typing 8-25
STRUC directive 6-2
Structure type 6-2
Structure-field-name operator 8-5
Structures
  assembly listing 2-18
  declarations 6-2
  definitions 6-1, 6-3
  fields 6-5
  initializing 6-2, 6-3, 6-4
  operands 6-5
  overview 6-1, 6-6
  variables 6-3
SUB instruction 15-3, 16-6
Subprograms, BASIC 16-16
Subroutines, BASIC 16-16

Substitute operator (&) 10-15
Subtitles in listings 11-3
Subtracting values 15-3
Subtraction operator 8-3
SUBTTL Directive 11-3
Summary
  masm 1-5
  options 2-2
Switch, emulating C statement 16-2
Symbol space E-1
Symbolic information 2-14
Symbols
  assembly listing 2-21
  communal 7-1, 7-8
  defined 3-4
  defining from command line 2-5
  external 7-3
  global 7-1, 7-3
  location counter 5-20
  public 7-1
  relocatable operands 13-4

# T

-t option 2-11, E-2
TBYTE type specifier 5-1
Temporary real format 5-16
TEST instruction 16-4, 16-9, 16-15
Testing bits 16-10
Text editor 1-1, 1-3, 1-4
Text equates *See* String equates
Text Macros 10-4
.TFCOND directive 2-13, 11-6
THIS operator 8-13
Timing of instructions 13-1
Tiny memory model 4-2
TITLE directive 11-2
Transcendental calculations 18-28
Trap flag 12-12, 16-25
Trigonometric functions 18-28
Two's complement 5-6
Type
  ABS 7-3
  align 4-17, 4-22
  checking, strict A-5
  class 4-23
  combine 4-19, 4-21
  data 2-14
  null class 4-24
  operand matching 8-25
  operators 8-11
  PROC 7-4
Type *(continued)*
  record 6-6
  specifiers 7-4
  structure 6-2
  use 4-18
  USE 13-11
.TYPE operator 8-15
TYPE operator 8-16
Type specifiers 5-1

# U

Unary minus 8-3
Unary plus 8-3
Undefined operand 5-19
Underscore (_) 3-5
Unpacked BCD numbers 5-9, 15-11
Unsigned numbers 5-6, 14-4, 15-2, 15-
Uppercase *See* Case
Upward compatibility 12-1
Use type 4-18
USE type 13-11

# V

-v option 2-11, E-1
Variables
  communal 7-8
  defined 5-5
  external 7-3
  floating point 5-12
  initializing 5-6
  integer 5-6
  local 16-21
  pointer 5-11
  public 7-1
  real number 5-12
  record 6-8
  string 5-10
  structure 6-3
Virtual 8086 Mode flag 12-13

# W

-w option 2-12, 8-26
WAIT instruction 18-10, 19-2
Warning levels 2-12, 8-26

Weak typing in other assemblers 8-26
While, emulating high-level-language statement 16-13
WIDTH operator 6-12
Width, structures 6-7
WORD align type 4-17
WORD type specifier 5-1

## X

-X option 2-13
-x option 2-14, 11-6
.XALL directive 10-9, 11-7
XCHG instruction 14-2
.XCREF directive 11-9
XLAT instruction 14-2
.XLIST directive 11-5
XOR instruction 15-14, 15-17
XOR operator 8-7

## Z

-z option 2-14
-Zd option 2-14
Zero flag 12-12
Zero-extending 14-6
-Zi option 2-14