

XENIX[®] System V

Development System

C Language 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.

Replace this Page
with Tab Marked:

C User's Guide



XENIX[®] System V

Development System

C 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.

Contents

1 Introduction

- 1.1 Overview 1-1
- 1.2 About This Guide 1-1
- 1.3 New Features 1-3
- 1.4 Notational Conventions 1-5
- 1.5 Books about C 1-7

2 Compiling with the cc Command

- 2.1 Introduction 2-1
- 2.2 The Basics: Compiling and Linking C Programs 2-2
- 2.3 Using cc Options 2-4

3 Linking with the cc Command

- 3.1 Introduction 3-1
- 3.2 The Default Linking Process 3-1
- 3.3 Passing Linker Information: The -link Option 3-1

4 Running C Programs on XENIX

- 4.1 Introduction 4-1
- 4.2 Passing Command-Line Data to a Program 4-1

5 Working with Memory Models

- 5.1 Introduction 5-1
- 5.2 Near, Far, and Huge Addressing 5-3
- 5.3 Using the Standard Memory Models 5-4
- 5.4 Using the **near**, **far**, and **huge** Keywords 5-12
- 5.5 Creating Customized Memory Models 5-22
- 5.6 Setting the Data Threshold 5-27
- 5.7 Naming Modules and Segments 5-28
- 5.8 Specifying Text and Data Segments 5-30

6 Improving Program Speed

- 6.1 Introduction 6-1
- 6.2 Using Register Variables 6-1
- 6.3 Optimization Options and Pragmas 6-2
- 6.4 Choosing the Function-Calling Convention 6-5
- 6.5 Efficiency in Large Data Models 6-6
- 6.6 Efficiency in Large Code Models 6-7

7 Object and Executable File Formats

- 7.1 Introduction 7-1
- 7.2 iAPX 286, 386 System Architecture 7-1
- 7.3 The Intel Object Module Format 7-2
- 7.4 Definition of Terms 7-4
- 7.5 Module Identification and Attributes 7-6
- 7.6 Segment Definition 7-7
- 7.7 Segment Addressing 7-7
- 7.8 Symbol Definition 7-8
- 7.9 Indices 7-8
- 7.10 Conceptual Framework for Fixups 7-8
- 7.11 Self-Relative Fixups 7-13
- 7.12 Segment-Relative Fixups 7-14
- 7.13 Record Order 7-15
- 7.14 Introduction to the Record Formats 7-16
- 7.15 Numeric List of Record Types 7-42
- 7.16 Type Representations for Communal Variables 7-43
- 7.17 The Segmented x.out Format 7-45

8 C Language Compatibility with Assembly Language

- 8.1 Introduction 8-1
- 8.2 C Calling Sequence for 8086/80286 8-1
- 8.3 Entering an 8086/80286 Assembly Routine 8-2
- 8.4 8086/80286 Return Values 8-2
- 8.5 Exiting an 8086/80286 Routine 8-2
- 8.6 8086/80286 Program Example 8-3
- 8.7 80386 C Language Calling Sequence 8-4
- 8.8 Entering an 80386 Assembly-Language Routine 8-4
- 8.9 80386 Return Values 8-5
- 8.10 Exiting a 80386 Routine 8-7
- 8.11 80386 Program Example 8-7

9 Error Processing

- 9.1 Introduction 9-1
- 9.2 Using the Standard Error File 9-1
- 9.3 Using the errno Variable 9-2
- 9.4 Printing Error Messages 9-2
- 9.5 Using Error Signals 9-3
- 9.6 Encountering System Errors 9-4

A Converting from Previous Versions of the Compiler

- A.1 Introduction A-1
- A.2 Differences between Versions 5.0 and 4.0 A-1
- A.3 Differences between Versions 4.0 and 3.0 A-5

B Writing Portable Programs

- B.1 Introduction B-1
- B.2 Program Portability B-2
- B.3 Machine Hardware B-2
- B.4 Compiler Differences B-9
- B.5 Environment Differences B-13
- B.6 Portability of Data B-14
- B.7 Type-Size Summary B-14
- B.8 Byte-Ordering Summary B-16

C Writing Programs for Read-Only Memory

- C.1 Introduction C-1
- C.2 XENIX-Dependent Library Routines C-1

D C Error Messages and Exit Codes

- D.1 Introduction D-1
- D.2 Command-Line Error Messages D-1
- D.3 Compiler Error Messages D-5
- D.4 Compiler Exit Codes D-41

Chapter 1

Introduction

- 1.1 Overview 1-1
- 1.2 About This Guide 1-1
- 1.3 New Features 1-3
- 1.4 Notational Conventions 1-5
- 1.5 Books about C 1-7

1.1 Overview

The C language is a powerful general-purpose programming language that can generate efficient, compact, and portable code. The Microsoft® C Compiler (cc) for the XENIX® operating system is a full implementation of the C language as defined by its authors, Brian W. Kernighan and Dennis M. Ritchie, in *The C Programming Language*.

XENIX C offers several important features to help you increase the efficiency of your C programs. You can choose among five standard memory models (small, medium, compact, large, and huge) to set up the combination of data and code storage that best suits your program. For flexibility and even greater efficiency, the XENIX C Compiler allows you to “mix” memory models by using special declarations in your program.

The C language itself does not provide such standard features as input and output capabilities and string-manipulation features. These capabilities are provided as part of the run-time library of functions that accompanies the XENIX C Compiler. Because the functions that require interaction with the operating system (for example, input and output) are logically separate from the language itself, the C language is especially suited for producing portable code.

The portability of your XENIX C programs is increased by the use of a common run-time library for XENIX and MS-DOS® installations. Using the routines in this library, you can transport programs easily from a XENIX development environment to an MS-DOS machine, or vice versa. For more information on the common library for XENIX and MS-DOS, see the *XENIX C Library Guide*.

Compared with other programming languages, C is extremely flexible concerning data conversions and nonstandard constructions. The XENIX C Compiler offers several levels of warnings to help you control this flexibility; programs in an early stage of development can be processed using the full warning capabilities of the compiler to catch mistakes and unintentional data conversions. An experienced C programmer can use a lower warning level for programs that contain intentionally nonstandard constructions. For more information about this feature, see Chapter 2, “Compiling with the cc Command.”

1.2 About This Guide

This guide explains how to use the XENIX C Compiler to compile, link, and run C programs on your XENIX system. The guide assumes that you are familiar with the C language and with XENIX, and that you know how to create and edit a C-language source file on your system. All examples

XENIX C User's Guide

in this guide were generated with the 286 C compiler.

If you have questions about the C language, turn to the *XENIX C Language Reference* included in this package. The *XENIX C Library Guide* documents the run-time library routines you can use in your C programs.

The following describes the remaining chapters of the *XENIX C User's Guide*:

Chapter 2, "Compiling with the cc Command," describes how to compile a program using the cc compiler driver. This chapter describes the options most commonly used to control preprocessing, compiling, and output of files.

Chapter 3, "Linking with the cc Command," describes how to link object files using the cc command. This chapter explains how the linker searches for libraries, shows how to specify libraries for linking, and describes the linker options that can be used for C programs.

Chapter 4, "Running C Programs on XENIX," explains how to run your executable program file, and discusses features specific to the XENIX implementation of C. The chapter tells how to pass data from XENIX to a program at execution time, and how to return an exit code from your program to XENIX.

Chapter 5, "Working with Memory Models," describes methods of managing memory models. These methods are useful for writing programs that use more than 64K (kilobytes) of code or data. This chapter also discusses "mixed-model" programming (combining features from the five standard memory models).

Chapter 6, "Improving Program Speed," gives suggestions and hints for maximizing program speed.

Chapter 7, "Object and Executable File Formats," describes the system architecture of the 80x86 microprocessor family, the object module format that the C compiler follows, and the format of the **x.out** file in a segmented environment.

Chapter 8, "C Language Compatibility with Assembly Language," describes how you can embed assembly language subroutines within C language programs.

Chapter 9, "Error Processing," describes how to process errors detected in calls to the C library routines and explains the functions and variables a program may use to respond to these errors.

Appendix A, “Converting from Previous Versions of the Compiler,” summarizes the differences between Version 5.0 of the XENIX C Compiler and previous versions. This appendix gives instructions for converting programs written for versions prior to 5.0 to the format accepted by Version 5.0.



Appendix B, “Writing Portable Programs,” lists some of the C language features that are implementation-dependent, and offers suggestions for increasing program portability.

Appendix C, “Writing Programs for Read-Only Memory,” gives information about modifying start-up code and initializing floating-point support for programs that will be put in read-only memory.

Appendix D, “C Error Messages and Exit Codes,” lists and describes the error messages and exit codes generated by the XENIX C Compiler and by the `cc` command. It also lists and explains run-time error messages produced by executable programs written in C.

1.3 New Features

Several useful features have been added to Version 5.0 of the XENIX C Compiler. This section summarizes features added since Version 4.0. For information about differences between Version 5.0 and versions prior to 4.0, see “Converting from Previous Versions of the Compiler.”

New features include the following:

Feature	Description	
New <code>cc</code> options	Option	Action
	-Oi	Generates intrinsic forms for certain library functions
	-Ol	Enables loop optimizations
	-Op	Forces consistent precision in the results of floating-point math operations
	-Sp	Specifies lines per page for source listings

XENIX C User's Guide

-Ss	Specifies subtitles for source listings
-St	Specifies titles for source listings
-Tc	Specifies C source files for files without extensions

New pragmas

Pragma	Action
---------------	---------------

alloc_text	Names the code segment used to allocate specified functions
function	Disables intrinsic-function generation for particular functions
intrinsic	Specifies functions that will have intrinsic forms generated
loop_opt	Controls program loop optimization on a local basis
pack	Specifies byte boundaries for structure packing
same_seg	Provides information about far data allocation that the compiler uses to perform optimizations

const keyword

Declares that a value will not change during program execution.

Language changes

The C language syntax and semantics have been modified in certain cases to correspond with recent updates to the Draft Proposed American National Standard—Programming Language C (hereinafter referred to as the “ANSI C standard”). Consult “Converting from Previous Versions of the Compiler,” and the *XENIX C Language Reference* for more information.

New library functions

All library functions defined in the ANSI C standard are supported, except the functions added for international-language support. Some existing functions have been modified and enhanced.

1.4 Notational Conventions

The following notational conventions are used throughout this guide:

Example of Convention	Description of Convention
-----------------------	---------------------------

Examples

The typeface shown in the left column is used to simulate the appearance of information that would be printed on the screen or by a printer. For example, the following command line is printed in this special typeface:

```
cc -Foout.o -DTRUE=1 file.c
```

When this command line is discussed in text, items appearing on the command line, such as *out.o*, also appear in the special typeface.

Language elements

Bold type indicates elements of the C language that must appear in source programs as shown. Text that is normally shown in bold type includes operators, keywords, library functions, commands, options, and preprocessor directives.

Examples are shown below:

```
+=    #if defined()    int
if    -Fa              fopen
main  sizeof
```

**ENVIRONMENT
VARIABLES,
and MACROS**

Bold capital letters are used for environment variables, symbolic constants, and macros.

placeholders

Words in italics are placeholders that you must supply in command-line and option specifications and in the text for types of information. Consider the following option:

```
-H number
```

Note that *number* is italicized to indicate that it represents a general form for the **-H** option. In an actual command, you would supply a partic-

ular number for the placeholder *number*.

Occasionally, italics are also used to emphasize particular words in the text.

Missing code

Vertical ellipses are used in program examples to indicate that a portion of the program is omitted. For instance, in the following excerpt, the ellipses between the statements indicate that intervening program lines occur but are not shown:

```
count = 0;  
.  
.  
.  
*pc++;
```

[*optional items*]

Brackets enclose optional fields in command-line and option specifications. Consider the following option specification:

-D*identifier*[=*string*]

The placeholder *identifier* indicates that you must supply an identifier when you use the **-D** option. The outer brackets indicate that you are not required to supply an equal sign (=) and a string following the identifier. The inner brackets indicate that you are not required to enter a string following the equal sign, but if you do supply a string, you must also supply the equal sign.

Single brackets are used in C-language array declarations and subscript expressions. For instance, *a[10]* is an example of brackets in a C subscript expression.

Repeating elements...

Horizontal ellipses are used in syntax examples to indicate that more items having the same form may be entered. For example, in the Bourne shell, several paths can be specified in the **PATH** command, as shown in the following syntax:

PATH[=*path*;*path*]...

`{choice1|choice2}`

Braces and a vertical bar indicate that you have a choice of two or more items. Braces enclose the choices, and vertical bars separate them. You must choose one of them items unless all of them are also enclosed in double square brackets.

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

```
-W {0 | 1 | 2 | 3}
```

You can use **-W1**, **-W2**, or **-W3** to display different levels of warning messages or **-W0** to suppress all warning messages.

“Defined terms”

Quotation marks set off terms defined in the text. For example, the term “far” appears in quotation marks the first time it is defined.

Some C constructs require quotation marks. Quotation marks required by the language have the form " " rather than “ ”. For example, a C string used in an example would be shown in the following form:

```
"abc"
```

KEY+KEY

Small capital letters are used for the names of keys and key sequences, such as Enter and Ctrl-C. Key sequences to be pressed simultaneously are indicated by the key names in small caps separated by a plus sign (Ctrl-C).

1.5 Books about C

The manuals in this documentation package provide a complete programmer’s reference for XENIX C. They do not, however, teach you how to program in C. If you are new to C or to programming, you may want to familiarize yourself with the language by reading one or more of the following books:

Hancock, Les, and Morris Krieger. *The C Primer*. New York: McGraw-Hill Book Co., Inc., 1982.

XENIX C User's Guide

Hansen, Augie. *Proficient C*. Bellevue, Washington: Microsoft Press, 1986.

Harbison, Samuel P., and Greg L. Steele. *C: A Reference Manual*. Englewood Cliffs, New Jersey: Prentice-Hall Software Series, 1987.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.

Kochan, Stephen. *Programming in C*. Hasbrouck Heights, New Jersey: Hayden Book Company, Inc., 1983.

Plum, Thomas. *Learning to Program in C*. Cardiff, New Jersey: Plum Hall, Inc., 1983.

Schildt, Herbert. *C Made Easy*. Berkeley, California: Osborne McGraw Hill, 1985.

Schustack, Steve. *Variations in C*. Bellevue, Washington: Microsoft Press, 1985.

These books are listed for your convenience only.

Chapter 2

Compiling with the cc Command

- 2.1 Introduction 2-1
- 2.2 The Basics: Compiling and Linking C Programs 2-2
 - 2.2.1 The cc Command 2-2
- 2.3 Using cc Options 2-4
 - 2.3.1 Setting Processor and Memory Model (-M) 2-4
 - 2.3.2 Specifying Help (-help) 2-6
 - 2.3.3 Specifying Source Files (-Tc) 2-6
 - 2.3.4 Compiling without Linking (-c) 2-7
 - 2.3.5 Naming the Object File (-Fo) 2-7
 - 2.3.6 Naming the Executable File (-Fe) (-o) 2-8
 - 2.3.7 Creating Listing Files 2-9
 - 2.3.8 Controlling the Preprocessor 2-24
 - 2.3.9 Checking for Program Errors 2-31
 - 2.3.10 Preparing for Debugging (-Zi, -Od) 2-36
 - 2.3.11 Optimizing 2-37
 - 2.3.12 Enabling/Disabling Language Extensions (-Ze, -Za) 2-49
 - 2.3.13 Packing Structure Members (-Zp) 2-50
 - 2.3.14 Setting the Stack Size (-F) 2-52
 - 2.3.15 Restricting the Length of External Names (-nl) 2-53
 - 2.3.16 Labeling the Object File (-V) 2-53
 - 2.3.17 Suppressing Default-Library Selection (-ZI) 2-54
 - 2.3.18 Changing the Default char Type (-J) 2-55
 - 2.3.19 Controlling the Calling Convention (-Gc) 2-55
 - 2.3.20 Compiling Programs for DOS Environment (-dos, -FP) 2-57
 - 2.3.21 Displaying Compiler Passes (-d, -z) 2-58

2.1 Introduction

This chapter explains how to compile and link using the `cc` command and discusses commonly used `cc` options. The `cc` command is the only command you need to compile and link your C source files. The `cc` command executes the three compiler passes, then automatically invokes `ld`, the Linker, to link your files.

Using the `cc` options described in this chapter, you can control and modify the tasks performed by the command. For example, you can direct `cc` to create an object-listing file or a preprocessed listing. Options also let you give information that applies to the compilation process; you can specify the definitions for manifest (symbolic) constants and macros, and the kinds of warning messages you want to see.

For a quick overview of the more commonly used options, enter:

```
cc -help
```

“The Basics: Compiling and Linking C Programs” explains the basic use of the `cc` command to produce an executable program.

“Using `cc` Options,” describes the `cc` options.

The `-help` option is described in greater detail in the “Using `cc` Options” section of this chapter.

The `cc` command automatically optimizes your program. You never have to give an optimizing instruction unless you want to change the way `cc` optimizes, request more sophisticated optimizations, or disable optimization altogether. For more information on these choices, see the “Optimizing” subsection of the “Using `cc` Options” section later in this chapter.

For information about linking object files and libraries using the `cc` command, see the “Linking with the `cc` Command” chapter of this guide.

For a discussion of the `cc` options that control memory models, see the “Working with Memory Models” chapter in this guide.

For a summary of the `cc` command and its options, see the *XENIX C Language Reference*.



2.2 The Basics: Compiling and Linking C Programs

This section explains how to use **cc** to compile and link C programs and discusses the rules and conventions that apply to file names and options used with **cc**.

2.2.1 The **cc** Command

The **cc** command has the following form:

```
cc [option]... file... [option... file...] [-link[link-libinfo]]
```

Each *option* is one of the command-line options described in the “Using **cc** Options” section of this chapter, in the “Working with Memory Models” chapter, and in the “Improving Program Speed” chapter of this guide.

Each *file* names a source or object file to be processed or a library to be searched at link time. See the section “Specifying Source and Object Files” for information about specifying source and object files.

The **cc** command automatically specifies the appropriate library to be used during linking; however, you can use the **-link** option with the optional *link-libinfo* argument to specify additional or different libraries, library search paths, and options to be used during linking. You can also specify linker options in the *linkoptions* argument. For information about specifying different libraries and linker options, see the “Linking with the **cc** Command” chapter of this guide.

You can give any number of options, file names, and library names on the command line, provided that the command line does not exceed 128 characters.

Specifying Source and Object Files

The **cc** command can process source files, object files, library files, or any combination of these. It uses the file-name extension (the period plus any letters that follow it) to determine what kind of processing the file needs, as shown in the following list:

- If the file has a **.c** extension, **cc** compiles the file.
- If the file has a **.o** extension, **cc** processes the file by invoking the linker.

- If the file has a `.a` extension, `cc` passes the file to the linker to be searched, unless the `-c` option is given to suppress linking. For a description of the `-c` option, see the section on “Compiling without Linking.”
- If the extension is omitted, `cc` assumes an extension of `.o`. If the extension is anything other than `.c`, `.o`, or `.a`, `cc` assumes the file is an object file unless the file name is specified in association with the `-Tc` option. If the file name is specified with the `-Tc` option, `cc` assumes the file is a C source file. For a description of the `-Tc` option, see the section on “Specifying Source Files.”



Examples

```
cc a.c b.c c.o d.o
```

This command line compiles the files `a.c` and `b.c`, creating object files named `a.o` and `b.o`. These object files are then linked with `c.o` and `d.o` to form an executable file named `a.out`.

```
cc a.c b.c c.o -Tcd.src
```

This command performs the same operations as the preceding command line, except that the `-Tc` option indicates that `d.src` is a source file, not an object file. Thus, the files `a.c`, `b.c`, and `d.src` are compiled, creating object files named `a.o`, `b.o`, and `d.o`. These object files are then linked with `c.o` to form an executable file named `a.out`.

Creating Executable Files

When `cc` compiles source files, it creates object files. By default, these object files have the same base names as the corresponding source files, but with the extension `.o` instead of `.c`. (The base name of a file extension is the portion of the name preceding the period, but excluding the path specification, if any.) You can use the `-Fo` option to give a different name to an object file.

Unless the `-c` option is given, `cc` links these object files, along with any `.o` files you give on the command line, to form an executable file. If only `.o` files are given on the command line, `cc` skips the compilation stage and simply links the files.

2.3 Using cc Options

The `cc` command offers a large number of command options to control and modify the compiler's operation. Options begin with a dash (-) and contain one or more letters.

Options can appear anywhere on the `cc` command line. In general, an option applies to all files that follow it on the command line, and it does not affect files preceding it there. However, not all options follow this rule; see the discussion of a particular option for information on its behavior. Keep in mind that most `cc` options apply only to the compilation process. Unless specifically noted, options do not affect any object files given on the command line.

2.3.1 Setting Processor and Memory Model (-M)

The `-M` option sets the program configuration. This configuration defines the program's memory model, word order, and data threshold. It also enables C language enhancements such as the use of the full 286 instruction set and special keywords.

```
cc -Mstring special.c
```

The *string* contains the argument that defines the configuration. It may be any combination of the following (though **s**, **m**, **c**, **l**, **h** are mutually exclusive):

- s** Create a small model program. This is the default.
- m** Create a middle model program.
- c** Create a compact model program.
- l** Create a large model program.
- h** Create a huge model program.
- e** Enable the keywords: **far**, **near**, **huge**, **pascal** and **fortran**. Also enables certain non-ANSI extensions necessary to ensure compatibility with existing versions of the C compiler. (This applies only to compiler versions that support features of ANSI C.)
- 0** Use only 8086 instructions for code generation. This is the default on 8086/80186/80286 systems.

- 1** Use the extended 80186 instruction set.
- 2** Use the extended 80286 instruction set.
- 3** Use the extended 80386 instruction set. This is the default on 80386 systems.
- b** Reverse the word order for **long** types, putting the high order word first. The default is the low order word first.

t*num* Causes all static and global data items whose size is greater than *num* bytes to be allocated to a new data segment. *Num*, the data “threshold” defaults to 32,767. This option can only be used in large model programs (**-Ml**). Its main use is to move data out of the near data segment to allow room for the stack.

```
cc -Ml -Mt12 recursive.c
```

- d** Do not assume (during compilation) that the registers `SS` and `DS` will have the same contents at run-time. *Warning:* This option has no library or runtime support on XENIX. It will **not** cause the stack to be put in a separate segment. It may be of use for DOS cross-development.

-M3 is the default on 80386 systems. Although it is usually advantageous to enable the appropriate instruction set, you are not required to do so. If you have an 80286 processor, for example, but you want your code to be able to run on an 8086, you should not use the 80186/80188 or 80286 instruction set.

Note

The **m**, **c**, **l**, **h**, **b**, **t**, and **d** arguments are compatible only with the **-M0**, **-M1**, or **-M2** option. The **s** and **e** arguments are compatible with **-M0**, **-M1**, **-M2**, or **-M3**.

For a complete description of memory models and segment options, see the “Working with Memory Models” chapter in this guide.

The memory-model option you choose determines the name of the standard libraries that `cc` places in the object file it creates. These libraries are then considered the default libraries, since the linker searches for them by default.

Table 2.1 shows each memory-model option and the corresponding library name that `cc` embeds in the object file.

Table 2.1
cc Options and Default Libraries

Memory-Model Option	Default Libraries
-Ms	Slibc.a Slibcfp.a
-Mm	Mlibc.a Mlibcfp.a
-Mc	Clibc.a Clibcfp.a
-Ml or -Mh	Llibc.a Llibfp.a

2.3.2 Specifying Help (-help)

Option

-help

This option displays a list of the most commonly used compiler options.

2.3.3 Specifying Source Files (-Tc)

Option

-Tc *sourcefile*

The **-Tc** option tells the `cc` command that the given file is a C source file. Zero or more spaces can appear between **-Tc** and the source-file name.

If this option does not appear, `cc` assumes that files with the extension `.c` are C source files, files with the extension `.a` are libraries, and files with

any other extension or with no extension are object files. If you use the `-Tc` option, `cc` treats the given file as a C source file, regardless of its extension, if any. A separate `-Tc` option must appear for each source file that has an extension other than `.c`.

Example

```
cc main.c -Tc test.prg -Tc collate.prg print.prg
```

In this example, the `cc` command compiles the three source files `main.c`, `test.prg`, and `collate.prg`. Since the file `print.prg` is given without a `-Tc` option, `cc` treats it as an object file. Thus, after compiling the three source files, `cc` links the object files `main.o`, `test.o`, `collate.o`, and `print.prg`.

2.3.4 Compiling without Linking (`-c`)

Option

`-c`

The `-c` (for “compile-only”) option suppresses linking. Source files given on the command line are compiled, but the resulting object files are not linked, no executable file is created, and any object files specified on the command line are ignored. This option is useful when you are compiling individual source files that do not make up a complete program.

The `-c` option applies to the entire `cc` command line, regardless of the option’s position in the command line.

Example

```
cc -c *.c
```

This command line compiles, but does not link, all files with the extension `.c` in the current working directory.

2.3.5 Naming the Object File (`-Fo`)

Option

`-Foobjfile`

By default, `cc` gives each object file it creates the base name of the corresponding source file plus the extension `.o`. The `-Fo` option lets you give different names to object files or create them in a different directory.



XENIX C User's Guide

If you are compiling more than one source file, you can give the **-Fo** option for each source file to rename the corresponding object file.

Keep the following rules in mind when using this option:

- The *objfile* argument must appear immediately after the option, with no intervening spaces.
- Each **-Fo** option applies to the next source file that appears on the command line after the option.

You are free to supply any name and any extension you like for the *objfile*. However, it is recommended that you use the conventional **.o** extension because the linker uses **.o** as the default extension when processing object files.

If you do not give a complete object file name with the **-Fo** option (that is, if you do not give an object file name with a base and an extension), **cc** names the object files according to the following rule:

- If you give only a directory specification following the **-Fo** option, **cc** creates the object file in the given directory and uses the default file name (the base name of the source file plus **.o**).

When you give a directory specification, it must end with a forward slash (/) so that **cc** can distinguish between a directory specification and a file name.

Example

```
cc -Fo/object1/this.c that.c -Fo/src/newthose.o those.c
```

In this example, the first **-Fo** option tells the compiler to create, in the */object1* directory, the object files *this.o* (created as a result of compiling *this.c*) and *that.o* (created as a result of compiling *that.c*). The second **-Fo** option tells the compiler to create the object file named *newthose.o* (created as a result of compiling *those.c*) in the */src* directory.

2.3.6 Naming the Executable File (-Fe) (-o)

Option

-Fe*exe*file
-o *exe*file

By default, `cc` gives the name *a.out* to the executable file. In XENIX, `-Fe` and `-o` are the same, except that, syntactically, the file name must come immediately after `-Fe`, whereas blanks can be between the `-o` and the file name. The `-Fe` option lets you give the executable file a different name or create it in a different directory.

Since `cc` creates only one executable file, you can give the `-Fe` option anywhere on the command line. If more than one `-Fe` option appears, `cc` gives the executable file the name specified in the last `-Fe` option on the command line.

The `-Fe` option applies only in the linking stage. If you specify the `-c` option to suppress linking, `-Fe` has no effect.

Examples

```
cc -Fe/bin/process *.c
cc -o /bin/process *.c
```

These examples compile and link all source files with the extension `.c` in the current working directory. The resulting executable file is named *process* and is created in the directory */bin*.

2.3.7 Creating Listing Files

A number of listing options are available with the `cc` command. You can create a source listing, a map listing, or one of several kinds of object listings. You can also set title and subtitle of the source listing from the command line and control the length of source-listing lines and pages.

The options available for producing listings and controlling their appearances are described in this section.

Note

Listings produced by the `cc` command may contain names that begin with more than one underscore (for example, `__chkstk`) or that end with the suffix `QQ`. Names that use these conventions are reserved for internal use by the compiler, and should not be used in your programs, except for those documented in the *XENIX C Library Guide*. Moreover, you should avoid creating global names that begin with an underscore. Since the compiler automatically adds another leading underscore, these names will have two leading underscores and might conflict with the names reserved by the compiler.

Types of Listings (-Fs, -Fl, -Fa, -Fc, -Fm)

Options

-Fs <i>[listfiles]</i>	Source listing
-Fl <i>[listfile]</i>	Object listing
-Fa <i>[listfile]</i>	Assembly listing
-Fc <i>[listfile]</i>	Combined source and object listing
-Fm <i>[mapfile]</i>	Map file that lists segments, in order

This section describes how to use command-line options to create listings. For an example of each type of listing and a description of the information it contains, see "Formats for Listings" later in this section.

When using an option described in this section, the *listfile* argument, if given, must follow the option immediately, with no intervening spaces. The *listfile* can be a file specification or a path specification. It can also be omitted.

Note

When you give just a path specification as the *listfile* argument, the path specification must end with a forward slash (/) so that **cc** can distinguish it from an ordinary file name.

When you give a path specification as the argument to a listing option, or if you omit the argument altogether, **cc** uses the default file name for the listing type. Table 2.2 gives the default names used for each type of listing. The table also shows the default extensions, which are used when you give a file-name argument that lacks an extension.

Table 2.2
Default File Names and Extensions

Option	Listing Type	Default File Name¹	Default Extension²
-Fs	Source	Base name of source file plus .S	.S
-Fl	Object	Base name of source file plus .L	.L
-Fa	Assembly	Base name of source file plus .s	.s
-Fc	Combined source-object	Base name of source file plus .L	.L
-Fm	Map	Base name of first object file on the command line plus .map	.map

Notes:

- 1 The default file name is used when the option is given with no argument or with a path specification as the argument.
- 2 The default extension is used when a file name lacking an extension is given.

Since you can process more than one file at a time with the `cc` command, the order in which you give listing options and the kind of argument you give for each option (file specification or path specification) affect the result. Table 2.3 summarizes the effects of each option with each type of argument.

Table 2.3
Arguments to Listing Options

Option	File-Name Argument	Path Argument ¹	No Argument
-Fa, -Fc, -Fl, -Fs	Creates a listing for next source file on command line; uses default extension if no extension is supplied	Creates listings in the given location for every source file listed after the option on the command line; uses default names	Creates listings in the current directory for every source file listed after the option on the command line; uses default names
-Fm	Uses given file name for the map file; uses default extension if no extension is supplied	Creates map file in the given directory; uses default name	Uses default name

Notes:

¹ When you give just a path specification as the argument, the path specification must end with a forward slash (/) so that `cc` can distinguish it from an ordinary file name.

Only one type of object or assembly listing can be produced for each source file. The **-Fc** option overrides the **-Fa** and **-Fl** options; whenever you use **-Fc**, a combined listing is produced. If you apply both the **-Fa** and the **-Fl** options to one source file, only the last listing specified on the command line is produced. If you specify both the **-Fa** and the **-Fs** options to one source file, a combined listing is produced.

Note

The `cc` command optimizes by default, so listing files reflect the optimized code. Since optimization may involve rearrangement of code, the correspondence between your source file and the machine instructions may not be clear, especially when you use the **-Fc** option to mingle the source and assembly codes. To produce a listing without optimizing, use the **-Od** option (discussed in the "Preparing for Debugging" section) with the listing option.

The map file is produced during the linking stage. If linking is suppressed with the `-c` option, the `-Fm` option has no effect.

Examples

```
cc -Fshello.src -Fchello.cmb hello.c
```

In this example, `cc` creates a source listing called *hello.src* and a combined source and assembly listing called *hello.cmb*. The object file has the default name *hello.o*.

```
cc -Fshello.src -Fshello.lst -Fchello.cod hello.c
```

This command produces a source listing called *hello.lst* rather than *hello.src*, since the last name provided has precedence. This example also produces an object-listing file named *hello.cod*. The object file in this example has the default name *hello.o*.

Setting Titles (`-St`) and Subtitles (`-Ss`)

Options

```
-St "title"  
-Ss "subtitle"
```

The `-St` and `-Ss` options set the title and subtitle, respectively, for source listings. The quotation marks (" ") around the *title* or *subtitle* argument can be omitted if the title or subtitle does not contain space or tab characters. The space between `-St` or `-Ss` and its arguments is optional.

The title appears in the upper left corner of each page of the source listing. The subtitle appears below the title.

The `-St` or `-Ss` option applies to the remainder of the command line or until the next occurrence of `-St` or `-Ss` on the command line. These options do not cause source listings to be created. They take effect only when the `-Fs` option is also used to create a source listing.

Examples

```
cc -St "Income Tax" -Ss 4-14 -Fs tax*.c
```

This command compiles and links all source files beginning with *tax* and ending with the default extension (*.c*) in the current working directory.

XENIX C User's Guide

Each page of the source listing contains the title *Income Tax* in the upper left corner. The subtitle *4-14* appears below the title on each page.

```
cc -c -Fs -St"Calc Prog" -Ss"count" ct.c -Ss"sort" srt.c
```

In this command, **cc** compiles two source files and creates two source listings. Each source listing has a unique subtitle, but both listings have the title *Calc Prog*.

Formats for Listings

The following sections describe and show examples of the five types of listings available with the **cc** command. For information on how to create these listings, see "Types of Listings" earlier in this chapter.

Source Listing

Source listings are helpful in debugging programs as they are being developed. These listings are also useful for documenting the structure of a finished program.

The source listing contains the numbered source-code lines of each procedure in the source file, along with any diagnostic messages that were generated. If the source file compiles with no errors more serious than warning errors, the source listing also includes tables of local symbols, global symbols, and parameter symbols for each function. If the compiler is unable to finish compilation, it does not generate symbol tables.

At the end of the source listing is a summary of the segment sizes in your program. This summary is useful for analyzing the program's memory requirements.

Any error messages that occurred during compilation appear in the listing after the line that caused the error, as shown in the following example:

```
1 char hexvalue[10];
2
3 main()
4 {
5     long htoi();
6     printf("Please enter the hex value you want to convert:\n");
7     scanf("%s", hexvalue);
8     printf("The integer value of the hex value is %ld\n", htoi(hexvalue));
9 }
10
11 long htoi(hexvalue)
12 char *hexvalue;
13 {
14     register char *ptr=hexvalue;
15     int i=0;
16     long n=0;
17     long expl6();
18     while (*ptr != '\0') {
19         if (*ptr >= 'a' && *ptr <= 'f')
20             *ptr -= 87;
21         else if (*ptr >= 'A' && *ptr <= 'F')
22             *ptr -= 55;
23         else
24             *ptr -= 48;
25         ptr++;
26     }
****bomb.c(25) : error 59: syntax error : ';'
}
```

2

The line number given in the error message corresponds to the number of the source line immediately above the message in the source listing.

The following example shows the source listing for a simple C program. The command used to obtain the output would be:

```
cc -St"Hex to ASCII" -St"2/25/87" HextoASCII.c
```


XENIX C User's Guide

Hex to ASCII
2/25/87

PAGE 1
02-25-87
10:44:23

```

Line# Source Line                                XENIX C Compiler Version 3.00.17
1 char hexvalue[10];
2
3 main()
4 {
5     long htoi();
6     printf("Please enter the hex value you want to convert:0);
7     scanf("%s", hexvalue);
8     printf("The integer value of the hex value is %ld0, htoi(hexvalue));
9 }
10
11 long htoi(hexvalue)
12 char *hexvalue;
13 {
14     register char *ptr=hexvalue;
15     int i=0;
16     long n=0;
17     long exp16();
18     while (*ptr != ' ') {
19         if (*ptr >= 'a' && *ptr <= 'f')
20             *ptr -= 87;
21         else if (*ptr >= 'A' && *ptr <= 'F')
22             *ptr -= 55; 23
24             *ptr -= 48;
25         ptr++;
26     }
27     ptr -= 1;
28     while (ptr>=hexvalue)
29     {
30         n+= (*ptr*exp16(i));
31         i++;
32         ptr--; 33     }
34     return(n);
35 }

```

htoi	Local Symbols	CRName	Class	Type	Size	Offset	Register	i
i	auto				-0008		
ptr	auto				***	si	
n	auto				-0004		
hexvalue.	param				0004		

```

36
37 long exp16(exp)
38 int exp;
39 {
40     long result=1;
41     int j;
42     for (j=1; j<=exp; j++)
43         result *= 16;
44     return(result);
45 }

```

Compiling with the cc Command

```
Hex to A
2/25/87                                02-25-87
                                         10:44:23

                                         XENIX C Compiler Version 3.00.17

expl6 Local Symbols

Name          Class  Type      Size  Offset  Register
j . . . . . auto          -0006
result. . . . . auto      -0004
exp . . . . . param       0004

Global Symbols

Name          Class  Type      Size  Offset
expl6 . . . . . global near function  ***  00ae
hexvalue. . . . . common struct/array  10   ***
htoi. . . . . global near function  ***  0038
main. . . . . global near function  ***  0000
printf. . . . . extern near function  ***   ***
scanf. . . . . extern near function  ***   ***

Code size = 00e8 (232)
Data size = 005f (95)
Bss size = 0000 (0)

No errors detected
```

2

At the end of each function, a table of local symbols is given, as shown in the following example for the function *htoi*:

```
htoi Local Symbols

Name          Class  Type      Size  Offset  Register
i . . . . . auto          -0008
ptr . . . . . auto          ***    si
n . . . . . auto      -0004
hexvalue. . . . . param       0004
```

XENIX C User's Guide

The following list shows the contents of each column in the symbol table:

Column Contents

- Name* The name of each local symbol in the function.
- Class* Either *auto* if the symbol is a nonstatic local variable, or *param* if the symbol is a formal parameter.
- Type* Not used for local symbols.
- Size* Not used for local symbols.
- Offset* The symbol's offset address relative to the frame pointer (that is, the **BP** register). The *Offset* number is positive for *param* symbols and negative for *auto* symbols with **auto** storage class.
- Register* Blank unless the variable is stored in a register, in which case, this column indicates the register (**SI** or **DI**).

At the end of the source code, a table of global symbols is given, as shown in the following example:

Name	Class	Type	Size	Offset
exp16	global	near function	***	00ae
hexvalue.	common	struct/array	10	***
htoi.	global	near function	***	0038
main.	global	near function	***	0000
printf.	extern	near function	***	***
scanf	extern	near function	***	***

The following list shows the contents of each column:

Column Contents

- Name* Each global symbol, external symbol, and statically allocated variable declared in the source file.
- Class* Either *global*, *common*, *extern*, or *static*, depending on how the symbol was defined in the source file.
- Type* A simplified version of the symbol's type as declared in the source file.

For functions, this entry is either *near function* or *far function*, depending on which memory model was used and how the function was declared. For a pointer, this entry is *near pointer*, *far pointer*, or *huge pointer*. For enumeration variables, this entry is *int*. For structures, unions, and arrays, this entry is *struct/array*.

Size Used only for variables. Specifies the number of bytes of storage allocated for the variable. Since the amount of storage allocated for an external array may not be known, its *Size* entry may be undefined.

Offset Used only for symbols with an entry of *global* or *static* in the *Class* column.

For variables, this entry gives the relative offset of the variable's storage in the logical data segment for the program file being compiled. Since the linker usually combines several logical data segments into a physical segment, this number is useful only for determining the relative position of storage of variables. For functions, this entry gives the relative offset of the start of the function in the logical code segment. For small-model programs, the linker combines logical code into a single physical segment, so this entry is useful for determining the relative positions of different functions defined in the same source file. However, for medium-, large-, and huge-model programs, each logical code segment becomes a unique physical segment. In these cases, this entry gives the actual offset of the function in its run-time code segment.

The last table in the source listing shows the segments used and their size, as in the following example:

```
Code size = 0103 (259)
Data size = 005f (95)
Bss size  = 0000 (0)
```

The number of bytes in each segment is given first in hexadecimal, and then in decimal (in parentheses).

Object Listing

The `-Fl` option produces an object listing. The object listing contains the instruction encoding and assembly code for your program. The line numbers are shown in the listing as comments. The instruction-encoding

XENIX C User's Guide

is on the left and the assembly code on the right, as shown in the following 286 example:

```
; Line 4
PUBLIC _main
_main PROC NEAR
*** 000000      55                push bp
*** 000001      8b ec            mov  bp,sp
*** 000003      33 c0           xor  ax,ax
*** 000005      e8 00 00           call __chkstk
; Line 6
*** 000008      b8 00 00        mov  ax,OFFSET DGROUP:$S G12
*** 00000b      50             push ax
*** 00000c      e8 00 00        call _printf
*** 00000f      83 c4 02        add  sp,2
```

Assembly Listing

The **-Fa** option produces an assembly listing. It contains the assembly code corresponding to your C source file, as shown in the following 286 example:

```
; Line 4
PUBLIC      _main
_main PROC NEAR
push bp
mov  bp,sp
xor  ax,ax
call __chkstk
; Line 6
mov  ax,OFFSET DGROUP:$SG12
push ax
call _printf
add  sp,2
```

Note that the example shows the same code as in the object listing example, except that the instruction encoding is omitted.

The listing generated by the **-Fa** option in Versions 5.0 and later of the XENIX C Compiler can be used as input to the XENIX Macro Assembler (**masm**).

Combined Source and Object Listing

The **-Fc** option produces a combined source and object listing. This shows each line of your source program followed by the corresponding line (or lines) of machine instructions, as in the following 286 example:

```

TEXT      SEGMENT
;|*** char hexvalue[10];
;|***
;|*** main()
;|*** {
; Line 4
      PUBLIC _main
_main PROC NEAR
      *** 000000      55                      push bp
      *** 000001      8b ec          mov  bp,sp
      *** 000003      33 c0          xor  ax,ax
      *** 000005      e8 00 00          call __chkstk
;|*** long htoi();
;|*** printf("Please enter the hex value you want to convert:0);
; Line 6
      *** 000008      b8 00 00          mov  ax,OFFSET DGROUP:$SG12
      *** 00000b      50                      push ax
      *** 00000c      e8 00 00          call _printf
      *** 00000f      83 c4 02          add  sp,2
;|*** scanf("%s", hexvalue);

```



Note that this sample is like the object-listing sample, except that the source-program line is provided in addition to the line number.

When you examine a listing file, you will notice that the names of globally visible functions and variables begin with an underscore, as shown in the following example. (This part of the listing is the same for all three kinds of listings.):

```

EXTRN _printf:NEAR
EXTRN _scanf:NEAR
EXTRN __chkstk:NEAR
EXTRN _aInmul:NEAR
EXTRN _aNNalshl:NEAR
EXTRN _hexvalue:TBYTE

```

The XENIX C Compiler automatically prefixes an underscore to all global names. If you write assembly-language routines to interface with your C program, this naming convention is important; see the section on “Controlling the Preprocessor” for more information.

XENIX C User's Guide

The listing may also contain names that begin with more than one underscore (for example, `__chkstk` in the example). Identifiers with more than one leading underscore are reserved for internal use by the compiler, and should not be used in your programs, except for those documented in the *XENIX C Library Guide*. Moreover, you should avoid creating global names that begin with an underscore. Since the compiler automatically adds another leading underscore, these names will have two, and might conflict with the names reserved by the compiler.

Map File

The `-Fm` option produces a map file. The map file contains a list of segments in order of their appearance within the load module. As an example, consider the following 386 example:

Start	Length	Name	Class
003f:00000000	015CDH	_TEXT	CODE
003f:000015d0	00000H	_ETEXT	ENDCODE
.			
.			
.			

The information in the *Start* column shows the 20-bit address (in hexadecimal) of each segment, relative to the beginning of the load module. The load module begins at location zero. The *Length* column gives the length of the segment in bytes; the *Name* column gives the name of the segment, and the *Class* column gives information about the segment type.

The starting address and name of each group appear after the list of segments. An example of a group listing follows:

Origin	Group
01EA:0	DGROUP

In this example, **DGROUP** is the name of the data group. **DGROUP** is the only group used for data segments by programs compiled with the XENIX C Compiler, Version 5.0.

The following map file contains two lists of global symbols: the first list is sorted in ASCII-character order by symbol name and the second is by symbol address. A maximum of 2048 symbols can be sorted in each list. (To increase the number of sorted symbols, you must specify the `-MAP` linker option with the *number* argument to create the map file; see the

“Linking with the cc Command” chapter of this guide for details.) The notation *Abs* appears next to the names of absolute symbols (symbols containing 16-bit constant values that are not associated with program addresses).

Many of the global symbols that appear in the map file are symbols used internally by the XENIX C Compiler. These usually begin with one or more leading underscores or end with *QQ*. The following 286 example illustrates this:

Address	Publics by Name
003F:0096	STKHQQ
0047:1D86	__brkctl
003F:04B0	__edata
0047:0910	__end
.	
.	
0047:00EC	__abrkp
0047:009C	__abrktb
0047:00EC	__abrktbe
003F:9876	Abs __acrmsg
0000:9876	Abs __acrused
.	
.	
0047:0240	____argc
0047:0242	____argv
Address	Publics by Value
003F:0010	__main
003F:0047	__htoi
003F:00DA	__expl6
003F:0113	__chkstk
003F:0129	__astart
003F:01C5	__cintDIV
.	
.	
.	

The addresses of the external symbols are in the “*selector:offset*” format, showing the location of the symbol relative to zero (the beginning of the load module).

Following the lists of symbols, the map file gives the program entry point, as shown in the following example:

```
Program entry point at 003F:0129
```

2.3.8 Controlling the Preprocessor

The `cc` command provides several options that control the operation of the C preprocessor. You can define macros and manifest (symbolic) constants from the command line, change the search path for include files, and stop compilation of a source file after the preprocessing stage to produce a preprocessed source-file listing.

The C preprocessor recognizes only preprocessor directives. It treats the source file as a text file, processing substitutions and definitions as directed. The preprocessor can be run on a file at any stage of development, whether or not the file is a complete C source file. In fact, the preprocessor is not restricted to processing C files; it can be run on any kind of file. However, input files to the preprocessor must follow the preprocessor rules; therefore, not all arbitrary text files may be suitable for use with the preprocessor. See the *XENIX C Language Reference* for a complete discussion of C preprocessor directives and the format expected for preprocessor input.

Defining Constants and Macros (-D)

Option

```
-D identifier[=string]
```

The `-D` option lets you define a constant or macro used in your source file. The *identifier* is the name of the constant or macro and *string* is its value or meaning. Note that spaces are permitted (but not required) between `-D` and the identifier.

If you leave out both the equal sign and *string*, the given constant or macro is assumed to be defined, and its value is set to 1. For example, `-DSET` is sufficient to define `SET`.

If you give the equal sign with an empty string, the given constant or macro is considered defined; its definition is the empty string. This definition effectively removes all occurrences of the identifier from the source file. For example, to remove all occurrences of *register*, use the following option:

```
-Dregister=
```

Note that the identifier *register* is still considered to be defined.

The effect of using the **-D** option is the same as using a preprocessor **#define** directive at the beginning of your source file: the identifier is defined in the source file being compiled either until an **#undef** directive removes the definition or until the end of the file is reached.

You can supply a command-line definition for an identifier that is also defined within the source file. However, you must use **#undef** to remove the source-file definition, unless the source-file definition is identical to the command-line definition. The command-line definition remains in effect until the identifier is removed with an **#undef** directive.

Normally, up to 17 definitions are allowed on the command line. Using either the **-Za** option or the **-J** option on the command line reduces the number of definitions allowed to 16; using both of these options reduces the number to 15. If you need to define more than the maximum number of identifiers, you can remove certain predefined definitions from the command line. See the discussion of the **-U** and **-u** options in the section on “Removing Definitions of Predefined Identifiers,” for more information.

The **-D** option is especially useful with the **#if** and **#ifdef** directives because you can control conditional-compilation directives in the source file from the command line.

Examples

```
cc -D NEED=2 main.c
```

This example defines the manifest constant *NEED* in the source file *main.c*. This definition is equivalent to placing the directive at the top of the source file as shown in the following example:

```
#define NEED 2
```

For the next example, suppose a source file named *other.c* contains the following fragment:

```
#if defined(NEED)
.
.
.
#endif
```

Suppose further that *other.c* does not explicitly define *NEED* (that is, no **#define** directive for *NEED* is present). Then all statements between the

#if and the **#endif** directives are compiled only if you supply a definition of *NEED* by using *-D*. For instance, the following command is sufficient to compile all statements following the **#if** directive:

```
cc -DNEED main.c
```

Note that *NEED* does not have to be set to a specific value to be considered defined. The following command, in contrast, causes the statements in the **#if** block to be ignored (not compiled):

```
cc main.c
```

Predefined Identifiers (Manifest Defines)

The compiler defines several identifiers that are useful in writing portable programs. These are known “manifest defines.” You can use these identifiers to compile code sections conditionally, depending on the processor and operating system being used. They begin with “M_” for “manifest.” The predefined identifiers and their functions are as follows:

Identifier	Function
M_I86	This is an Intel processor.
M_SYS3	This is Unix System III compatible.
M_SYS5	This is Unix System V compatible.
M_BITFIELDS	This compiler supports bitfields.

M_WORDSWAP	The word-within-a-longword order is swapped with respect to the DEC PDP11.
M_XENIX	Always defined, this identifies target operating system as XENIX.
M_In86	Depending on -M0 , -M1 , -M2 or -M3 , M_I386 is defined with 386 compiler unless -dos is used.
M_I86mM	Always defined, this identifies memory model, where <i>m</i> is either S (small model), C (compact model), M (medium model), L (large model), or H (huge model). If huge model is used, both M_I86LM and M_I86HM are defined. Small model is the default. Memory models are discussed in “Working with Memory Models.”
_CHAR_UNSIGNED	This is defined only when the -J option is given to make the char type unsigned by default. For more information, see the section on “Changing the Default char Type.”
M_SDATA , or M_LDATA	Depending on -M0 , -M1 , or -M2 .
M_STEXT or M_LTEXT	Depending on -M0 , -M1 , or -M2 .

Removing Definitions of Predefined Identifiers (-U, -u)

Options

-U *identifier*
-u

The **-U** (for “undefine”) option turns off the definition of one of the predefined identifiers discussed in the previous section; one or more

spaces may separate the **-U** and *identifier*. You can specify more than one **-U** option on the same command line. The **-u** option turns off all definitions.

These options are useful if you want to give more than the maximum number of definitions (16, if the **-Za** or **-J** option is used; 15, if both options are given; or 17, otherwise) on the command line, or if you have other uses for the predefined identifiers. For each definition of a predefined identifier you remove, you can substitute a definition of your own on the command line. When the definitions of all predefined identifiers are removed, you can specify up to 512 command-line definitions.

Example

```
cc -UM_XENIX -UM_I86 work.c
```

This example removes the definitions of two predefined identifiers. Note that the **-U** option must be given twice to do this.

Producing a Preprocessed Listing (-P, -E, -EP)

Options

- P** Writes preprocessed output to a file
- E** Writes preprocessed output to standard output; includes **#line** directives
- EP** Writes preprocessed output to standard output

The **-P**, **-E**, and **-EP** options produce listings of preprocessed files. These options allow you to examine the output of the C preprocessor.

The preprocessed listing file is identical to the original source file except that all preprocessor directives are carried out, macro expansions are performed, and comments are removed. All three options suppress compilation; no object file or listing is produced, even if you specify an **-Fo** option or a listing-file option on the **cc** command line.

The **-P** option writes the preprocessed listing to a file with the same base name as the source file, but with an **.i** extension.

The **-E** option copies the preprocessed listing to the standard output (usually your terminal). It places a **#line** directive in the output at the

beginning and end of each included file and around lines removed by preprocessor commands that specify conditional compilation.

The `-E` option is useful when you want to resubmit the preprocessed listing for compilation. The `#line` directives renumber the lines of the preprocessed file, so that errors generated in later stages of processing refer to the original source file rather than to the preprocessed file.

The `-EP` option combines features of the `-E` and `-P` options; the file is preprocessed and copied to the standard output, but no `#line` directives are added.

Examples

```
cc -P main.c
```

This example creates the preprocessed file `main.i` from the source file `main.c`.

```
cc -E add.c > preadd.c
```

This command creates a preprocessed file with inserted `#line` directives from the source file `add.c`. The output is redirected to the file `preadd.c`.

```
cc -EP add.c
```

The command shown here produces the same preprocessed output as the second example, but without the `#line` directives. The output appears on the screen.

Preserving Comments (`-C`)

Option

`-C`

Normally, comments are stripped from a source file in the preprocessing stage, since they do not serve any purpose in later stages of compiling. The `-C` (for “comment”) option preserves comments during preprocessing. The `-C` option is valid only when the `-E`, `-P`, or `-EP` option is also used.

Example

```
cc -P -C sample.c
```

XENIX C User's Guide

The example produces a listing named *sample.i*. The listing file contains the original source file, including comments, with all preprocessor directives expanded or replaced.

Searching for Include Files (-I, -X)

Options

-I *directory*
-X

The **-I** and **-X** options temporarily override the default search paths for include files. (Default path is */usr/include*.)

You can add to the list of directories searched by using the **-I** (for “include”) option. This option causes the compiler to search the directory or directories you specify before searching the default path */usr/include*. The space between **-I** and *directory* is optional. You can add more than one include directory by giving the **-I** option more than once in the **cc** command. The directories are searched in order of their appearance in the command line.

The directories are searched only until the specified include file is found. If the file is not found in the given directories or the standard places, the compiler prints an error message and stops processing. When this occurs, you must restart compilation with a corrected directory specification.

You can prevent the C compiler from searching the default paths for include files by using the **-X** (for “exclude”) option. When **cc** sees the **-X** option, it considers the list of standard places to be empty. This option is often used with the **-I** option to define the location of include files that have the same names as include files found in other directories, but that contain different definitions.

Examples

```
cc -I /include -I/alt/include main.c
```

In this example, **cc** looks for the include files requested by *main.c* in the following order: first in the directory */include*, then in the directory */alt/include*, and finally in the default directory */usr/include*.

```
cc -X -I /alt/include main.c
```

As shown in this example, the compiler looks for include files only in the directory `/alt/include`. First the `-X` option tells `cc` to consider the list of standard places empty; then the `-I` option specifies one directory to be searched.

2.3.9 Checking for Program Errors



You may encounter several different kinds of error messages when you compile, link, and run a XENIX C program.

Several `cc` options are available to control the types of warnings generated at compile time, help with syntax checking, and verify compatibility between the actual arguments and formal parameters of a function during the early stages of program development. This section describes these options.

Understanding Error Messages

Error messages can appear at different stages of program development:

- In the compiling stage, the compiler generates a broad range of error and warning messages to help you locate errors and potential problems in your source files.
- During the linking stage, the linker is responsible for generating error messages.
- During program execution, any error messages you see are runtime error messages. This category includes messages about floating-point exceptions, which are errors generated by an 8087 or 80287 coprocessor.

Other utilities included in this package, such as the XENIX Linker (`ld`) and the `make` program-maintenance utility, generate their own error messages.

When you are compiling and linking using the `cc` command, you may see both compiler and linker messages. Compiler messages have numbers preceded by the letter `C`, and linker messages have numbers preceded by the letter `L`.

You can also distinguish the type of a message by its format. See “C Error Messages and Exit Codes” in this guide for a description of compiler error-message formats, a list of actual compiler error messages, and explanations of the circumstances that cause them.

Compiler error messages are sent to the standard output, which is usually your terminal. If you are using the C shell, you can redirect the messages

XENIX C User's Guide

to a file by using the standard redirection symbols at the end of your command line:

```
>&.
```

If you are using the Bourne shell, you can redirect the messages to a file by using the standard redirection syntax:

```
cmd > outputfile 2>&1
```

Example

Assume the following source file named *rm.c*:

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char argv[];

    {
    register int i;
    char *name;

    for (i = 1; i < argc; ++i)
        if (unlink(name = argv[i])) {
            printf("couldn't delete %s : ", name);
            perror("");
        }
    }
```

The following C shell command line redirects error messages to a file named *rm.err*:

```
cc rm.c >& rm.err
```

In the previous command, only output that ordinarily goes to the console screen is redirected. The error-message file *rm.err* contains the following information:

```
rm.c
rm.c(11): error C2065: 'arg' : undefined
rm.c(12): warning C4047: '=' : different levels of indirection
```

Based on the errors generated, you can correct *rm.c* as shown below:

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char *argv[];           /* corrects warning C4047 */
{
    register int i;
    char *name;

    for (i = 1; i < argc; ++i) /* corrects error C2065 */
        if (unlink(name = argv[i])) {
            printf("couldn't delete %s : ", name);
            perror("");
        }
}
```

2

Setting the Warning Level (-W, -w)

Option

-W{0|1|2|3}
-w

You can suppress warning messages produced by the compiler by using the **-W** (for “warning”) option. Compiler warning messages are any messages beginning with *C4*; see “C Error Messages and Exit Codes,” for a full listing. Warnings indicate potential problems (rather than actual errors) with statements that may not be compiled as you intend. The **-W** options affect only source files given on the command line; they do not apply to object files.

The **-W0** option turns off warning messages. This option is useful when you compile programs that deliberately include questionable statements. The **-W0** option applies to the remainder of the command line or until the next occurrence of a **-W** option on the command line. The **-w** option has the same effect as the **-W0** option.

The **-W1** option (the default) causes the compiler to display most warning messages.

XENIX C User's Guide

The **-W2** option causes the compiler to display an intermediate level of warning messages. Level-2 warnings may or may not indicate serious problems; they include the following:

- Use of functions with no declared return type
- Failure to put **return** statements in functions with non-**void** return types
- Data conversions that would cause loss of data or precision

The **-W3** option displays the highest level of warning messages, including warnings about the uses of non-ANSI features and extended keywords and about function calls before the appearance of function prototypes in the program.

Note that the warning messages in “Error Messages and Exit Codes” indicate the warning level that must be set (that is, the number for the appropriate **-W** option) for the message to appear.

Example

```
cc -W3 crunch.c print.c
```

This example enables all possible warning messages when the *crunch.c* and *print.c* source files are compiled.

Checking Syntax (-Zs)

Option

-Zs

The **-Zs** option causes the compiler to perform only a syntax check on the source files that follow the option on the command line. This option provides a quick way to find and correct syntax errors before you try to compile and link a source file.

When you give the **-Zs** option, the compiler does not generate code or produce object files, object listings, or executable files. However, the compiler does display error messages if the source file has syntax errors. You can specify the **-Fs** option on the same command line to generate a source listing that shows these error messages. For more information about the **-Fs** option, see the section on “Creating Listing Files.”

Example

```
cc -Zs test*.c
```

This command causes the compiler to perform a syntax check on all source files in the current working directory that begin with *test* and end with the default extension (.c). The compiler displays messages for any errors found.



Generating Function Declarations (-Zg)

Option

-Zg

The **-Zg** option generates a function declaration for each function defined in the source file. The function declaration includes the function return type and an argument-type list created from the types of the formal parameters of the function. Any function declarations already present in the source file are ignored.

The generated list of declarations is written to the standard output. It can be saved in a file using shell redirection.

When the **-Zg** option is used, the source file is not compiled. As a result, no object file or listing is produced.

The list of declarations is helpful for verifying that actual arguments and formal parameters of a function are compatible. You can save the list and include it in your source file to cause the compiler to perform type-checking. The presence of a declared argument-type list for a function “turns on” the compiler’s type-checking between actual arguments to a function (given in the function call) and the formal parameters of a function.

This type-checking can be a helpful feature in writing and debugging C programs, especially when working with older C programs. Argument type checking is a recent addition to the C language, so many existing C programs will not have argument-type lists. See the *XENIX C Language Reference* for more information about function declarations and argument-type lists.

You can use the **-Zg** option even if your source program already contains some function declarations. The compiler accepts more than one occurrence of a function declaration, as long as the declarations do not

conflict. No conflict occurs when one declaration has an argument-type list and another declaration of the same function does not, as long as the return types are identical.

Note

If you use the **-Zg** option and your program contains formal parameters that have structure, enumeration, or union type (or pointers to such types), then the declaration for each structure, enumeration, or union type must have a tag. For example, use the following form:

```
struct tagA {  
    .  
    .  
    .  
} A;
```

Example

```
cc -Zg file.c > filedecls.h
```

This command causes the compiler to generate argument-type lists for functions defined in *file.c*. The list of declarations is redirected to *filedecls.h*.

2.3.10 Preparing for Debugging (-Zi, -Od)

Options

- Zi** Creates object file for use with the source-level debugger **sdb**
- Od** Disables code optimization to help with debugging

The **-Zi** option produces an object file containing full symbolic-debugging information for use with the source-level debugger. This object file includes full symbol-table information and line numbers. If the **-Zi** option is given with no explicit **-O** options, all optimizations involving code motion and rearrangement are suppressed, although simple optimizations are still performed. If any explicit **-O** options are given, *all* requested optimizations are performed.

The **-Od** option tells the compiler not to perform most optimizations. Some peephole optimizations and other simple optimizations are still

performed. (Without the **-Od** option, the default is to optimize.) You may want to use this option when you plan to use a symbolic debugger with your object file, since optimization can involve rearrangement of instructions that make it difficult for you to recognize and correct your code when debugging. However, turning off optimizations may increase the size of the code generated to the point where it might not be possible to link your program.

Other optimization options are discussed in the section on “Optimizing.”



Example

```
cc -zi -Od test.c
```

This command produces an object file named *test.o* that contains line numbers corresponding to the line numbers of *test.c*. Limited optimization is performed.

2.3.11 Optimizing

The optimizing capabilities available with the XENIX C Compiler can reduce the storage space or execution time required for a program. This is achieved by eliminating unnecessary instructions and rearranging code. The compiler performs some optimizations by default. You can use the **-O** options, the **loop_opt** pragma (described in the section on “Loop Optimization”), and the **intrinsic** pragma (described in the section under “Generating Intrinsic Functions”) to exercise greater control over the optimizations performed. In addition, you can use the **-Gs** option or **check_stack** pragma to reduce program size and speed up execution.

Controlling Optimization (-O Options)

Option

```
-Ostring  
#pragma loop_opt({on|off})  
#pragma intrinsic(function1[,function2]..)  
#pragma function(function1[,function2]..)
```

Note

This option is valid only for 286 code (generated using the **M2** compiler flag).

The **-O** options give you control over the optimization procedures that the compiler performs. One or more of the letters in *string* following the **-O** let you choose how the compiler performs optimization:

Letter	Optimizing Procedure
a	Relaxes alias checking
d	Disables optimization
i	Enables intrinsic functions
l	Enables loop optimization
p	Improves consistency of floating-point results
s	Favors code size during optimization
t	Favors execution speed during optimization (the default)
x	Maximizes optimization

The letters can appear in any order; for example, **-Oat** and **-Ota** have the same effect. More than one **-O** option can be given; the compiler uses the last **-O** option given if any conflict arises. Each option applies to all source files following that option on the command line.

The following sections discuss the various optimization options and their effects.

Relaxing Alias Checking (-Oa)

The **a** option letter can be used with the **l**, **s**, or **t** option letter to relax the assumptions the compiler makes about the use of "aliases" in the program. Aliases are multiple names (that is, symbolic references) for the same memory location in a program. Most commonly, aliases occur as a result of code similar to that shown in the following example:

```
func()
{
  int x, *p;

  p = &x; /* now "x" and "*p" refer to the same */
          /* memory location */
  .
  .
  .
}
```

Use of the **-Oa** option can reduce the size of executable files and speed program execution. Its use is especially recommended when you also specify the **-Ol** option, since the compiler can detect a number of loop optimizations when the **-Oa** option is in effect that it cannot detect when **-Oa** is not in effect. However, before you specify **-Oa**, you must make sure that your program does not use aliases either directly or indirectly.

The use of aliases is important only if both names are actually used to reference the memory location. The following example illustrates the use of aliases:

```
func()
{
    int x, *p;

    p = &x;

    .
    .
    .
  /* ...expressions involving only *p */
  .
  .
  .
}
```

Since all access to the memory location labeled *x* is through the pointer *p*, *x* has no significance in the function. To illustrate, *func* could be rewritten as the following pair of functions:

XENIX C User's Guide

```
func1()
{
    int x;

    func2(&x);
}

func2(p)
int *p;
{
    .
    .
    .
    /* ...expressions involving *p */
    .
    .
    .
}
```

In this equivalent form, the alias created in *func1* is insignificant, since the memory location is not referenced at all and *func2* does not use aliases since *x* is not even in the scope of the function. The **-Oa** option can be safely specified in compiling either of these equivalent forms.

In addition to the obvious cases discussed above, aliases can be created through the use of pointers in other, more subtle ways. Two such cases involving the use of pointers as function arguments are illustrated in the following example:

```
int x;

func(p)
int *p;
{
    .
    .
    .
    /* ...expressions involving *p and x */
    .
    .
    .
}
```

In this example, *x* is a communal variable, so the function can be called with *func(&x)*. The **-Oa** option can be used safely only if it is known that

func is never invoked with the address of *x* as an argument.

```

func(p1, p2)
int *p1, *p2;
{
    .
    .
    .
    /* ...expressions involving *p1 and *p2 */
    .
    .
    .
}

```



In this example, the function may be invoked with the same value for both arguments (that is, *func(p,p)* or *func(&x,&x)*). Thus, the **-Oa** option can be safely specified only if it is known that the function is always called with distinct values for the two arguments.

One use of aliases occurs so frequently that a special provision has been made for it. When the compiler encounters a call to a function with address-type arguments, it always assumes that all variables whose addresses are passed to the function are modified. If such function calls appear in a program, the **-Oa** option can be specified safely even though the function call results in an alias for each variable whose address is passed. The following example illustrates how the compiler handles this case:

```

func1 ()
{
    int x, y, a, b;
    .
    .
    .
    x = a + b;
func2 (&a);
    y = a + b;
}

```

As shown, when the compiler encounters the function call *func2(&a)*, it assumes that the function modifies *a*, even if the **-Oa** option has been specified. The compiler generates code to evaluate each instance of the expression *a + b*, rather than eliminating a common subexpression incorrectly.

XENIX C User's Guide

Although you should convert programs that use aliases if you plan to compile them with the **-Oa** option, it is helpful to know the units of a program where the optimizations affected by the use of **-Oa** are applied. This information indicates where the uses of aliases are most likely to cause incorrect optimizations if **-Oa** is specified. The following list describes the program units where such optimizations are performed:

- All of the C optimizations, except for loop optimizations, that may be affected by the incorrect use of **-Oa** are applied at the level of basic blocks. In the XENIX C Compiler, the **-Oa** option can generally be used even if aliases are employed, provided no memory location is referenced by more than one name within any basic block. (A "basic block" is a contiguous sequence of statements, with a unique entry point and exit point and no branching in between. In C programs, basic blocks most often appear as the clauses of **if** statements, **switch** statements, loop bodies, or function bodies, although they may also occur as sequences of statements delimited by user labels.)
- Loop optimizations are applied at the level of whole loop bodies. Thus, if loop optimization is enabled, **-Oa** can generally be used even if aliases are employed, provided that no memory location is referenced by more than one name within any basic block or loop body.

Disabling Optimization (-Od)

The **-Od** option turns off most optimizations. This is useful in the early stages of program development to avoid optimizing code that will later be changed. Because optimization may involve rearrangement of instructions, you may also want to specify the **-Od** option when you use a debugger with your program or when you want to examine an object-file listing. If you optimize before debugging, it can be difficult to recognize and correct your code. However, note that turning off or restricting optimization of a program usually increases the size of the generated code. If your program contains a module that is close to the 64K limit on compiled code, turning off optimization may cause the module to exceed the limit.

Generating Intrinsic Functions (-Oi)

The **-Oi** option tells the compiler to generate intrinsic functions instead of function calls for certain functions. Intrinsic functions may be in-line functions, may use special argument-passing conventions, or (in some cases) may do nothing. Programs that use intrinsic functions are faster

because they do not include the overhead associated with function calls. However, they may be larger because of the additional code that is generated.

Note

This option is only supported for the 286 compiler.

2

The following functions have intrinsic forms:

- **memset**, **memcpy**, and **memcmp**
- **strset**, **strcpy**, **strcmp**, and **strcat**
- **inp** and **outp**
- **_rotl**, **_rotr**, **_lrotl**, and **_lrotr**
- **min**, **max**, and **abs**

Note

Intrinsic versions of the **memset**, **memcpy**, and **memcmp** functions in compact- and large-model programs cannot handle huge arrays or huge pointers. To use huge arrays or huge pointers with these functions, you must compile your program with the huge memory model (that is, using the **-Mh** option on the command line).

You can use the **intrinsic** pragma to generate intrinsic functions only for selected functions. This pragma has the following format:

```
#pragma intrinsic (function1 [function2],...)
```

The **intrinsic** pragma affects the specified functions from the point where the pragma appears until either the end of the source file or the next **function** pragma specifying any of the same functions. The **function** pragma has the following format:

```
#pragma function (function1 [function2]...)
```

Note that you can also use the **function** pragma selectively to generate function calls instead of intrinsic functions when you compile a program with the **-Oi** option.

Note

The only pragma applicable to 386 code is the pack pragma; all others are not valid.

Loop Optimization (-Ol)

The **-Ol** option tells the compiler to perform loop optimizations. For best performance, the **-Ol** option should be specified along with the **a** option letter (**-Oal**), since the compiler can detect more loop optimizations when it relaxes its assumptions about the use of aliases.

You can use the **loop_opt** pragma to turn loop optimization on or off for selected functions. When you want to turn off loop optimization, put the following line before the code on which you don't want to perform loop optimization:

```
#pragma loop_opt (off)
```

Note that the preceding line disables loop optimization for all code that follows it in the source file, not just the routines on the same line. To reinstate loop optimization, insert the following line:

```
#pragma loop_opt (on)
```

If no argument is given to the **loop_opt** pragma, loop optimization reverts to the behavior specified on the command line: enabled if the **-Ox** or **-Ol** option is in effect, and disabled otherwise. The interaction of the **loop_opt**

`pragma` with the `-O1` and `-Ox` options is explained in greater detail in Table 2.4.

Table 2.4
Using the `loop_opt` Pragma

Syntax	Compiled with 2Ox or 2O1?	Action
<code>#pragma loop_opt()</code>	no	Turns off optimization for loops that follow
<code>#pragma loop_opt()</code>	yes	Turns on optimization for loops that follow
<code>#pragma loop_opt (on)</code>	yes or no	Turns on optimization for loops that follow
<code>#pragma loop_opt (off)</code>	yes or no	Turns off optimization for loops that follow



Achieving Consistent Floating-Point Results (`-Op`)

The `-Op` option is useful when floating-point results must be consistent within a program. This option changes the way in which the program handles floating-point values by default.

Ordinarily the compiler stores each floating-point value in an 80-bit register. In subsequent references to that value, the compiler reads the value from the register. When the final value is written to memory, it is truncated, since floating-point types are allocated fewer than 80 bits of storage (32 bits for the `float` type and 64 bits for the `double` type). Thus, the value stored in the register may actually be more precise than the same value stored in a floating-point variable. Since the value is truncated each time it is written to memory, over the course of the program the value stored in the machine register may become quite different from the value that is written to memory.

If you use the `-Op` option, when floating-point values are referenced, the compiler reloads them from floating-point variables rather than from registers. Using `-Op` gives less precise results than using registers, and it may increase the size of the generated code. However, it gives you more

control over the truncation (and hence the consistency) of floating-point values.

Optimizing for Speed and Code Size (-Ot, -Os)

When you do not give a **-O** option to the **cc** command, it automatically uses **-Ot**, meaning that program-execution speed is favored in the optimization. Wherever the compiler has a choice between producing smaller (but perhaps slower) and larger (but perhaps faster) code, the compiler generates faster code. For example, when the **-Ot** option is in effect, the compiler generates intrinsic functions to perform shift operations on long operands.

To cause the compiler to favor smaller code size instead, use the **-Os** option. For example, when the **-Os** option is in effect, the compiler uses function calls to perform shift operations on long operands.

Producing Maximum Optimization (-Ox)

The **-Ox** option is a shorthand way to combine optimizing options to produce the fastest possible program. Its effect is the same as using the following options on the same command line:

```
-Oaillt -Gs
```

That is, the **-Ox** option relaxes alias checking, generates all intrinsics for the functions listed in the section “Generating Intrinsic Functions,” performs loop optimizations, favors execution time over code size; and removes stack probes. Note that the interactions between the **-Ox** option and the **loop_opt** pragma are the same as those described in Table 2.4. For more information about stack probes and ways of controlling their use, see the following section, “Removing Stack Probes.”

Examples

```
cc -Oal file.c
```

This command tells the compiler to perform loop optimizations and relax alias-checking when it compiles *file.c*. The compiler favors program speed over program size, since the **-Ot** option is also specified by default.

```
cc -c -Os file.c
```

This command favors code size over execution speed when *file.c* is compiled:

```
cc -Od *.c
```

This command compiles and links all C source files with the default extension (`.c`) in the current directory and disables optimization. This command is most useful during the early stages of program development, since it improves compilation speed.



Removing Stack Probes (`-Gs`)

Options

```
-Gs  
#pragma check_stack([{on|off}])
```

You can reduce the size of a program and speed up execution slightly by removing stack probes. You can do this either with the `-Gs` option or with the `check_stack` pragma.

A “stack probe” is a short routine called on entry to a function to verify that there is enough room in the program stack to allocate local variables required by the function. The stack probe routine is called at every function entry point. Ordinarily, the stack probe routine generates a stack overflow message when it determines that the required stack space is not available. When stack-checking is turned off, the stack probe routine is not called, and stack overflow can occur without being diagnosed (that is, no error message is printed).

Use the `-Gs` option when you want to turn off stack-checking for an entire module if you know that the program does not exceed the available stack space. For example, stack probes may not be needed for programs that make very few function calls, or that have only modest local variable requirements. In the absence of the `-Gs` option, stack-checking is on.

Use the `check_stack` pragma when you want to turn stack-checking on or off only for selected routines, leaving the default (as determined by the presence or absence of the `-Gs` option) for the rest. When you want to turn off stack-checking, put the following line before the definition of the function you don't want to check:

```
#pragma check_stack (off)
```


Note that the preceding line disables stack-checking for all routines that follow it in the source file, not just the routines on the same line. To reinstate stack-checking, insert the following line:

```
#pragma check_stack (on)
```

Note

For earlier versions of XENIX C, the **check_stack** pragma had a different format: **check_stack+** to enable stack-checking and **check_stack-** to disable stack-checking. Although the XENIX C Compiler still accepts this format, its use is discouraged, since it may not be supported in future versions.

If no argument is given for the **check_stack** pragma, stack-checking reverts to the behavior specified on the command line: disabled if the **-Gs** option is given, or enabled if otherwise. The interaction of the **check_stack** pragma with the **-Gs** option is explained in greater detail in Table 2.5.

Table 2.5
Using the **check_stack** Pragma

Syntax	Compiled with -Gs Option?	Action
#pragma check_stack()	yes	Turns off stack-checking for routines that follow
#pragma check_stack()	no	Turns on stack-checking for routines that follow
#pragma check_stack(on)	yes or no	Turns on stack-checking for routines that follow
#pragma check_stack(off)	yes or no	Turns off stack-checking for routines that follow

Note

The `-Gs` option should be used with great care. Although it can make programs smaller and faster, it may mean that the program will not be able to detect certain execution errors.

Example

```
cc -Oa1s -Gs file.c
```

This example optimizes the file *file.c* by removing stack probes with the `-Gs` option. The letters specified with the `-O` option tell the compiler to relax alias-checking (**a**), perform loop optimization (**l**), and favor code size over program speed (**s**). If you want stack-checking for only a few functions in *file.c*, you can use the `check_stack` pragma around the definitions of functions you want to check. Similarly, if you want to perform loop optimization on only a few functions in *file.c*, you can use the `loop_opt` pragma around the definitions of functions on which you want to perform loop optimization.

2.3.12 Enabling/Disabling Language Extensions (-Ze, -Za)

Option

- `-Ze` Enables language extensions (default)
- `-Za` Disables language extensions

The XENIX C Compiler is moving to support the ANSI C standard. In addition, it offers a number of features beyond those specified in the ANSI C standard. These features are enabled when the `-Ze` (default) option is in effect and disabled when the `-Za` option is in effect. They include the following:

- The `cdecl`, `far`, `fortran`, `huge`, `near`, and `pascal` keywords
- Use of casts to produce values, as in this example:

```
int *p;
((long *)p)++;
```

XENIX C User's Guide

The preceding example could be rewritten to conform with ANSI C as shown here:

```
p = (int *) ((char *)p + sizeof(long));
```

- Redefinitions of **extern** items as **static**, as follows:

```
extern int foo();  
static int foo()  
{}
```

- Use of trailing commas (,) rather than an ellipsis (...) in function declarations to indicate variable-length argument lists, such as:

```
int printf(char *,);
```

- Benign **typedef** redefinitions within the same scope, like this:

```
typedef int INT;  
typedef int INT;
```

- Use of mixed character and string constants in an initializer, for instance:

```
char arr[5] = {'a', 'b', "cde"};
```

- Use of bit fields with base types other than **unsigned int** or **signed int**

Use the **-Za** option if you will be porting your program to other environments. The **-Za** option tells the compiler to treat extended keywords as simple identifiers and disable the other extensions listed previously.

2.3.13 Packing Structure Members (-Zp)

Option

```
-Zp{1|2|4}  
#pragma pack({1|2|4})
```

When storage is allocated for structures, structure members are ordinarily stored as follows:

- Items of type **char** or **unsigned char**, or arrays containing items of these types, are byte-aligned.
- Structures are word-aligned; structures of odd size are padded to an even number of bytes.
- All other types of structure members are word aligned.

To conserve space, or to conform to existing data structures, you may want to store structures more or less compactly. The **-Zp** option and the **pack** pragma control how structure data are “packed” into memory.

Use the **-Zp** option when you want to specify the same packing for all structures in a module. When you give the **-Zp[n]** option, where *n* is 1, 2, or 4, each structure member after the first is stored on *n*-byte boundaries, depending on the option you choose. If you use the **-Zp** option without an argument, structure members are packed on 1-byte boundaries.

On some processors, the **-Zp** option may result in slower program execution because of the time required to unpack structure members when they are accessed. For example, on an 8086 processor, this option can reduce efficiency if members with **int** or **long** type are packed in such a way that they begin on odd-byte boundaries.

Use the **pack** pragma when you want to specify packing other than that specified on the command line for particular structures. Give the **pack(n)** pragma, where *n* is 1, 2, or 4, before structures that you want to pack differently. To reinstate the packing given on the command line, give the **pack()** pragma with no arguments.

Table 2.6 shows the interaction of the **-Zp** option with the **pack** pragma.

Table 2.6
Using the pack Pragma

Syntax	Compiled with -Zp Option?	Action
#pragma pack()	yes	Reverts to packing specified on the command line for structures that fol-

XENIX C User's Guide

		low
#pragma pack()	no	Reverts to default packing for structures that follow
#pragma pack(<i>n</i>)	yes or no	Packs the following structures to the given byte boundary until changed or disabled

Example

```
cc -Zp prog.c
```

This command causes all structures in the program *prog.c* to be stored without extra space for alignment of members on **int** boundaries.

2.3.14 Setting the Stack Size (-F)

Option

-F *hexnum*

The **-F** option sets the size of the program stack. A space must separate the **-F** and *hexnum*. (This option applies only to the 286 compiler; 386 code uses a dynamic stack.)

The *hexnum* is a hexadecimal value representing the stack size in bytes. The value must be less than 0xFFFF hexadecimal (65,535 decimal).

If you do not specify this option, the start-up routine in the standard C library sets the default stack size to 2K.

If you get a stack-overflow message, you may need to increase the size of the stack. In contrast, if your program uses the stack very little, you may save some space by decreasing the stack size.

The **-F** option is a linking option that affects executable files only; it does not have any effect on source or object files.

Example

```
cc -F C00 *.o
```

This example sets the stack size to C00 hexadecimal (3K decimal) for the program created by linking all of the object files in the current working directory.

2.3.15 Restricting the Length of External Names (-nl)

Option

`-nl number`

The `cc` command allows you to restrict the length of external (public) names by using the `-nl` option. The *number* is an integer specifying the maximum number of significant characters in external names. The space between `-nl` and *number* is optional.

When you use the `-nl` option, the compiler considers only the first *number* characters of external names used in the program. The program may contain external names longer than *number* characters; the extra characters are simply ignored.

The `-nl` option is typically used to conserve space or to aid in creating portable programs. The XENIX C Compiler imposes no restrictions on the length of external names (although it uses only the first 31 characters), but other compilers or linkers may produce errors when they encounter names longer than a predetermined limit.

2.3.16 Labeling the Object File (-V)

Option

`-V string`

Use the `-V` (for “version”) option to embed a text string in an object file. The *string* must be enclosed in double quotation marks (" ") if it contains white-space characters or embedded double quotation marks. A backslash (\) must precede any embedded double quotation marks.

A typical use of the `-V` option is to label an object file with a version number or copyright notice.

Example

```
cc -V"XENIX C Optimizing Compiler Version 5.0" main.c
```



XENIX C User's Guide

This command places the following string in the object file *main.o*:

```
XENIX C Optimizing Compiler Version 5.0
```

2.3.17 Suppressing Default-Library Selection (-Zl)

Option

-Zl

Ordinarily, the compiler places the names of the default libraries for the memory-model options you have chosen in the object file for the linker to read. This allows the appropriate library to be linked with a program automatically.

The **-Zl** option tells the compiler not to place the default library name in the object file. As a result, the object file is slightly smaller.

The **-Zl** option is useful when you are building a library of routines. Every routine in the library need not contain the default-library information. Although the **-Zl** option saves only a small amount of space for a single object file, the total space saved is significant in a library containing many object modules. When you link a library of object modules created *with* the **-Zl** option and a C program file compiled *without* the **-Zl** option, the default-library information is supplied by the program file.

Example

```
cc one.c -Zl two.c
```

This example creates the following two object files:

1. An object file named *one.o* that contains the default library information
2. An object file named *two.o* that contains no default-library information

When *one.o* and *two.o* are linked, the default-library information in *one.o* causes the given library to be searched for any unresolved references in either *one.o* or *two.o*.

2.3.18 Changing the Default char Type (-J)

Option

-J

In XENIX C, the **char** type is signed by default, so if a **char** value is widened to **int** type, the result is sign-extended. You can change this default to **unsigned** with the **-J** option, causing the **char** type to be zero-extended when widened to **int** type. However, if a **char** value is explicitly declared **signed**, the **-J** option does not affect it, and the value is sign-extended when widened to **int** type.

When you specify **-J**, the compiler automatically defines the identifier **_CHAR_UNSIGNED**.

2.3.19 Controlling the Calling Convention (-Gc)

Options

-Gc
fortran
pascal
cdecl

The **-Gc** option and the **fortran**, **pascal**, and **cdecl** keywords allow you to control the function-calling and naming conventions so that your C programs can call and be called by functions that are written in FORTRAN and Pascal.

Because C, unlike other languages such as XENIX Pascal and XENIX FORTRAN, allows you to write functions that take variable numbers of arguments, it must handle function calls differently. Languages such as Pascal and FORTRAN normally push actual parameters to a function in left-to-right order, with the last argument in the list being the last one pushed on the stack. In contrast, C functions do not always know the number of actual parameters, so they must push their arguments from right to left, with the first argument in the list being the last one pushed.

Additionally, the calling function must remove the arguments from the stack in C (rather than having the called function do it, as in Pascal and FORTRAN). If the code for removing arguments is in the called function (as in Pascal and FORTRAN), it appears only once; if it is in the calling

XENIX C User's Guide

function (as in C), it appears every time there is a function call. Since function calls are more numerous than individual functions, the Pascal/FORTRAN method is slightly smaller and more efficient.

The XENIX C Compiler has the ability to generate the Pascal/FORTRAN calling convention in one of several ways. The first is through the use of the **pascal** and **fortran** keywords. When these keywords are applied to functions, or to pointers to functions, they indicate a corresponding Pascal or FORTRAN function. Therefore, the correct calling convention must be used. In the following example, *sort* is declared as a function using the alternative calling convention:

```
short pascal sort(char *, char *);
```

The **pascal** and **fortran** keywords can be used interchangeably. Use them when you want to use the left-to-right calling sequence for selected functions only.

The second method for generating the Pascal/FORTRAN calling convention is to use the **-Gc** option. If you do this, the entire module is compiled using the alternative calling convention. You might use this method to make it possible to call all the functions in a C module from another language, or to gain the performance and size improvement provided by this calling convention. When you use **-Gc** to compile a module, the compiler assumes that all functions called from that module use the Pascal/FORTRAN calling convention, even if the functions are defined outside that module. Thus, using **-Gc** would normally mean that you cannot call or define functions that take variable numbers of parameters, and that you cannot call functions such as the C library functions that use the C calling sequence. In addition, if you compile with the **-Gc** option, either you must declare the **main** function in the source program with the **cdecl** keyword, or you must change the start-up routine so that it uses the correct naming and calling conventions when calling **main**.

To overcome these restrictions, the **cdecl** keyword has been added to XENIX C. This keyword is the "inverse" of the **fortran** and **pascal** keywords. When applied to a function or function pointer, it indicates that the associated function is to be called using the normal C calling convention. This allows you to write C programs which take advantage of the more efficient calling convention while still having access to the entire C library, other C objects, and even user-defined functions that can take variable-length argument lists.

Run-time library functions all use the C calling convention. Therefore, care must be taken to declare them **cdecl** functions.

Use of the **pascal** and **fortran** keywords, or the **-Gc** option, also affects the naming convention for the associated item (or, in the case of **-Gc**, all items): the name is converted to uppercase (capital letters), and the leading underscore that C normally prefixes is not added. The **pascal** and **fortran** keywords can be applied to data items and pointers, as well as functions; when applied to data items or pointers, these keywords force the naming convention described above for that item or pointer.

The **pascal**, **fortran**, and **cdecl** keywords, like the **near**, **far**, and **huge** keywords, are disabled by use of the **-Za** option. If this option is given, these names are treated as ordinary identifiers, rather than keywords.

Examples

```
int cdecl var_print(char*, ...);
```

In this example, *var_print* is allowed to have a variable number of arguments by declaring it as a function using the normal right-to-left C function calling convention and naming conventions. The *cdecl* keyword overrides the left-to-right calling sequence set by use of the **-Gc** option when compiling the source file in which this declaration appears; if this file is compiled without the **-Gc** option, *cdecl* has no effect since it is the same as the default C convention.

```
float *pascal nroot(number, root)
```

This instruction declares *nroot* to be a function returning a pointer to a value of type *float*. The function *nroot* uses the default calling sequence (left-to-right) and naming conventions for XENIX FORTRAN and Pascal programs.

```
long pascal index
```

This example simply changes the naming convention for the data item *index*: it is included in the object file in all capital letters and without a leading underscore.

2.3.20 Compiling Programs for DOS Environment (**-dos**, **-FP**)

The XENIX C compiler is capable of compiling programs that will execute in the DOS environment.

The **-dos** option instructs the compiler to use a different set of libraries (from */usr/lib/dos*) and a different linker (**dosld**(CP)). Note that programs

XENIX C User's Guide

compiled with **-dos** will not run in the XENIX environment. Many XENIX system calls are not supported in DOS.

There are a variety of **-FP** options that can be used along with **-dos** to control floating point operations. For more information on **-FP** and on DOS cross-development in general, see "XENIX to DOS: A Cross-Development System," in the *XENIX C Library Guide*, and "Writing Portable Programs" in the *XENIX C User's Guide*.

2.3.21 Displaying Compiler Passes (-d, -z)

The **cc** command is actually a driver program which executes a series of compiler passes, perhaps an assembler pass, and a linker. It collects the various options and files on its command line and distributes them to the proper pass or to the linker. The XENIX C compiler is conceptually a four-pass compiler. The function of the various compiler passes is outlined below.

Pass 0

Pass zero of the compiler is commonly termed the pre-processor. It handles file inclusion, macro expansion and text substitution, and allows you to define constructs for conditional compilation.

Pass 1

Pass one of the compiler is called the parser. It performs two functions: (1) building a context-free grammar tree to pass to P2; and (2) constructing a symbol table.

Pass 2

Pass two generates code. It walks the grammar tree constructed by pass 1, applies semantic rules to each syntactic construct, and produces the binary code indicated by the semantic rules.

Pass 3

The third pass provides post-generation optimization. It analyzes the code generated by pass 2 and applies optimization rules to alter the code for better performance (such as elimination of redundant code, rearrangement, etc.). It creates the object code and outputs listing files (if requested).

The **-d** option displays the various passes and their arguments before they are executed. The **-z** option shows the passes but does not execute them.

Chapter 3

Linking with the cc Command

- 3.1 Introduction 3-1
- 3.2 The Default Linking Process 3-1
- 3.3 Passing Linker Information: The -link Option 3-1
 - 3.3.1 Specifying Libraries 3-2
 - 3.3.2 Specifying Linker Options 3-3

3.1 Introduction

Since the **cc** command controls linking as well as compiling, you can specify linker options and libraries other than the default combined library to be linked with your object files on the **cc** command line.

3.2 The Default Linking Process

When the **cc** command compiles a source file, it encodes the name of the appropriate library in the object file. The library name embedded in the library file is determined by the memory-model (**-M**) option you give on the **cc** command line. (For a list of the libraries, see the “Compiling with the cc Command” chapter of this guide.)

If you use the default memory-model option (**-Ms**), **cc** encodes the name of the standard library that corresponds to the defaults.

When an object file is linked, the linker looks for libraries matching the names encoded in the object file. The linker looks for these libraries first in */usr/lib/286*.

The result is that you do not ordinarily need to give library names on the **cc** command line. For descriptions of the situations that require you to specify libraries to the **cc** command, see the section on “Specifying Libraries.”

3.3 Passing Linker Information: The -link Option

To pass linker options or nondefault library names to the linker, give the following options on the **cc** command line after any source- and object-file names and **cc** options:

-link [*link-libinfo*]

Use the *link-libinfo* field to specify linker options, libraries, and library search paths. Note that library names can also be specified with source- and object-file names before the **-link** option on the command line, as long as the library names have the **.a** extension. These library names are searched before library names specified after the **-link** option. Refer to the following sections for more information:

- “Specifying Libraries,” to learn about specifying libraries and library search paths

- “Specifying Linker Options,” for descriptions of the linker options that apply to XENIX C.

If you use the **-link** option with the **cc** command, it must be the last option on the command line.

3.3.1 Specifying Libraries

To link object files with libraries other than the default library, give the names of the nondefault libraries on the **cc** command line. Library names appearing before **-link** must have the **.a** extension; library names appearing after **-link** may have blank extensions or no extensions. A space or plus sign (+) must follow each library name except the last.

Since the object file already contains the names of the correct combined library, you do not need to specify libraries unless you want to do any of the following:

- Link with additional libraries
- Look for libraries in different locations
- Override the use of the default library

Linking with Additional Libraries

If you specify additional libraries to **cc**, the linker searches the libraries you specify *before* it searches the default library to resolve external references in the object files. It searches the libraries you specify in their order of appearance on the command line.

If a library name includes a path specification, the linker searches only that path for the library.

If you specify only a library name (without a path specification), the linker searches in the following locations to find the given library file:

- The current working directory
- Any path specifications that you give in the *link-libinfo* field, in their order of appearance on the command line
- The default location */usr/lib*

If a library name without an extension appears after the **-link** option, the linker automatically supplies the **.a** extension. If you want to link a library file with an extension other than **.a**, you must specify the complete library name.

Looking in Different Locations for Libraries

You can tell the linker to look in different locations for libraries by giving a path specification in the *link-libinfo* field on the **cc** command line.

The linker looks for the default libraries in the same order as it looks for libraries given on the command line.



Overriding Libraries Named in Object Files

If you do not want to link with the library whose name is included in the object file, you can give the names of one or more different libraries instead. You will want to specify a different library name if you have renamed a standard library.

If you specify a new library name, the linker searches the new library to resolve external references before it searches the library specified in the object file.

If you want the linker to ignore the libraries named in the object file, you must use the **-NOD** linker option. This option tells the linker to ignore the default-library names encoded in the object files.

Example

```
cc fun.o text.o table.o care.o -link /testlib/newlibv3.a
```

This example links four object modules to create an executable file named *a.out*. The linker searches */testlib/newlibv3.a* before searching the default libraries to resolve references.

3.3.2 Specifying Linker Options

When you use the **cc** command to invoke the linker, any linker options you specify (other than those supported by **cc** options such as **-F** and **-Fm**) must appear after the **-link** option on the command line. All options begin with the linker's option character, the dash (-).

XENIX C User's Guide

The following sections outline the rules for specifying linker options on the `cc` command line.

Abbreviations

Since linker options are named according to their functions, some of these options are quite long. You can abbreviate the options to save space and effort. Be sure that your abbreviation is unique, so that the linker can determine which option you want. (The minimum legal abbreviation for each option is indicated in the syntax of the option.)

For example, several options begin with the letters “NO”; therefore, abbreviations for those options must be longer than “NO” to be unique. You cannot use “NO” as an abbreviation for the `-NOIGNORECASE` option, since the linker cannot tell which of the options beginning with “NO” you want. The shortest legal abbreviation for this option is `-NOI`.

Abbreviations must begin with the first letter of the option and must be continuous through the last letter typed. No gaps or transpositions are allowed.

Numerical Arguments

Some linker options take numerical arguments. A numerical argument can be any of the following:

- A decimal number from 0 to 65,535
- An octal number from 0 to 0177777. A number is interpreted as octal if it starts with 0. For example, the number `10` is a decimal number, but the number `010` is an octal number, equivalent to 8 in decimal
- A hexadecimal number from 0 to 0xFFFF. A number is interpreted as hexadecimal if it starts with 0x or 0X. For example, `0x10` is a hexadecimal number, equivalent to 16 in decimal

Differences from `cc` Options

If you are accustomed to using `cc` options, you should be aware that the linker options work in a slightly different manner. Keep the following differences in mind when you use linker options:

- Linker options can be abbreviated; **cc** options cannot. For example, the linker option **-NOIGNORECASE** can be abbreviated to **-NOI**.
- Case is not significant in linker options, as it is in **cc** options. For example, **-NOI** and **-noi** are equivalent.
- Linker options on the command line affect all files in the linking process, regardless of where the options appear in the *link-libinfo* field.

This section summarizes the linker options that can be used with XENIX C programs. Note that this section does not describe all available linker options; for a complete list, refer to the **ld(CP)** command in the *XENIX Programmer's Reference Manual*.



The following linker option is most commonly used with XENIX C programs:

-SE[GMENTS]:number

Controls the number of segments that the linker allows a program to have. The default is 128, but you can set *number* to any value (decimal, octal, or hexadecimal) in the range 1-1024 (decimal).

For each segment, the linker must allocate some space to keep track of segment information. When you set the segment limit higher than 128, the linker allocates more space for segment information. For programs with fewer than 128 segments, you can keep the storage requirements of the linker at the lowest level possible by setting *number* to reflect the actual number of segments in the program. The linker displays an error message if the number of segments allocated is too high for the amount of memory the linker has available.

The following linker options can be used with XENIX C programs, but they perform the same actions as **cc** options. Therefore, you do not need to use them unless you are compiling and linking in separate steps.

-M[AP][:number]

Creates a map file. This option is equivalent to using the **-Fm** option with the **cc** command, except that you can give a *number* argument with the **-M** option. The *number* argument is any positive integer (decimal, octal, or hexadecimal) up to 65,535 (decimal) specifying how many symbols are sorted in the map listing. If no *number* argument is given, a maximum of 2048 symbols is sorted. (In practice, the number of sorted symbols is limited by the amount of free heap space.) If a *number* argument is given, the alphabetical list of symbols does not appear in the map file.

-LI[NENUMBERS]

Creates a map file and includes the line numbers and associated addresses of the source program. This option is equivalent to using the **-Zd** option with the **cc** command. For more information about the **-Zd** option, see the "Compiling with the **cc** Command" chapter of this guide.

-ST[ACK]:*number*

Specifies the size of the stack for your program, where *number* is any positive value (decimal, octal, or hexadecimal) up to 65,535 (decimal) representing the size, in bytes, of the stack. This option is equivalent to using the **-F** option of the **cc** command. For more information about the **-F** option, see the "Compiling with the **cc** Command" chapter of this guide.

Chapter 4

Running C Programs

on XENIX

4.1 Introduction 4-1

4.2 Passing Command-Line Data to a Program 4-1

4.1 Introduction

After compiling and linking a program with the XENIX C Compiler and linking with the linker, you will have an executable file that can be run from the shell prompt.

XENIX uses the **PATH** environment variable to find executable files. You can execute your program from any directory, as long as the executable program file is in one of the directories on the path set in the **PATH** environment variable.

Your program can also be executed by other programs, and you can write it so that it will be capable of executing other programs. The **exec** and **system** routines provided in the run-time library allow your program to execute other programs. See the *XENIX C Library Guide* for a description of these routines.

XENIX has several other unique capabilities that your program can use if you write the program to take advantage of them. Among these capabilities are the following:

- Receiving arguments from the command line
- Reading information from the environment
- Sending a message to the shell by returning an exit code

This chapter explains how to write programs to take advantage of the first of these features, and how to use it once your program is completed.

4.2 Passing Command-Line Data to a Program

Your C program can access data from a command line or from the environment. You can use the Bourne shell commands to place data in the environment table. Command-line data are arguments that appear on the same line as the program name when you execute the program.

To pass data to your program on the command line, give one or more arguments after the program name when you execute the program. Each argument must be separated from the arguments around it by one or more spaces or tab characters, and may be enclosed in quotation marks (" "). If you want to give a single argument that includes spaces or tab characters,



XENIX C User's Guide

enclose the argument in quotation marks. For example, if your C program is called *try*, you might give it the following command line:

```
try 42 "de f" 16
```

In this case, the program will be executed and three arguments will be passed: *42*, *de f*, and *16*.

For a C program to read the data from the command line, the program should declare two variables as arguments to the **main** function. These variables and their contents are as follows:

Variable	Contents
<i>argc</i>	Number of arguments passed
<i>argv</i>	Array of strings containing arguments

By declaring these variables as arguments to **main**, you make them available as local variables in the **main** function. The following example illustrates how to declare these arguments:

```
main (argc, argv)
int argc;
char *argv[ ];
```

The number of arguments appearing on the command line is passed as the integer variable *argc*, and the command line is passed to the program as the array of strings indicated by *argv*.

The first argument of any command line is the name of the program to be executed. Therefore, the program name is the first string stored in *argv*, at *argv* [0]. Since a program name must be given to run the program, the integer value of *argc* is always at least 1. Therefore, if you pass two arguments to your program, *argc* will have a value of 3 (two arguments and the program name).

The first argument following the program name is stored at *argv* [1], the second is stored at *argv* [2], and so on, to the last argument. There is a third argument passed to the **main** function: *envp*, a pointer to the environment table. This argument is an extension provided by the XENIX C Compiler to support code ported from XENIX and other UNIX-like systems. When specified, it follows *argv* and is declared as follows:

```
char *envp[ ];
```

Although you can use this pointer to access the value of environment settings, this usage is nonstandard and not recommended. The **putenv** and **getenv** routines from the C run-time library accomplish the same task, and are easier and safer to use. Using the **putenv** routine may change the location of the environment table in memory, depending on memory requirements. Therefore, the value given to *envp* at the beginning of the program's execution may not be valid throughout. In contrast, the **putenv** and **getenv** routines access the environment table properly, even when its location changes. These routines use the global variable **environ** (described in the *XENIX C Library Guide*), which always points to the correct table location.

Example

```
myprog ABC "abc e" 3 8
```

This command line executes the program named *myprog* and passes the four command-line arguments to the **main** function. The arguments are stored as null-terminated strings, and the number of arguments is stored in *argc*. To access the last argument, for example, you would use an expression like the following:

```
argv[argc - 1]
```

Since the value of *argc* is 5 (counting the program name as an argument), this expression is equivalent to *argv[4]*, or the fifth string of the array.



Chapter 5

Working with Memory Models

- 5.1 Introduction 5-1
 - 5.1.1 Memory Model Considerations 5-2
- 5.2 Near, Far, and Huge Addressing 5-3
- 5.3 Using the Standard Memory Models 5-4
 - 5.3.1 Porting Considerations 5-5
 - 5.3.2 Creating Small-Model Programs 5-6
 - 5.3.3 Creating Medium-Model Programs 5-7
 - 5.3.4 Creating Compact-Model Programs 5-7
 - 5.3.5 Creating Large-Model Programs 5-9
 - 5.3.6 Creating Huge-Model Programs 5-9
 - 5.3.7 Segmentation Errors 5-10
- 5.4 Using the **near**, **far**, and **huge** Keywords 5-12
 - 5.4.1 Library Support for **near**, **far**, and **huge** 5-14
 - 5.4.2 Declaring Data with **near**, **far**, and **huge** 5-14
 - 5.4.3 Declaring Functions with the **near** and **far** Keywords 5-18
 - 5.4.4 Pointer Conversions 5-20
- 5.5 Creating Customized Memory Models 5-22
 - 5.5.1 Code Pointers 5-24
 - 5.5.2 Data Pointers 5-24
 - 5.5.3 Setting Up Segments 5-25
 - 5.5.4 Library Support for Customized Memory Models 5-26
- 5.6 Setting the Data Threshold 5-27
- 5.7 Naming Modules and Segments 5-28
- 5.8 Specifying Text and Data Segments 5-30

5.1 Introduction

Expanding the computing power of microcomputers often means giving the computer more “space” to work in. The Intel family of microprocessors (8080, 8086, 80286, and 80386) is a good example of such growth. Each new processor was capable of addressing more memory space than its predecessor.

The 8080 processor could address 64 kilobytes (64K) of memory, using 16-bit-wide address registers. For the 8086 processor, the address space was expanded to one-megabyte (1M). However, rather than expand the size of the address registers, a second set of “segment” registers was added. These registers select 64K blocks of memory, known as segments, within the one-megabyte address space. The 16-bit address registers then select an offset from the beginning of a segment through a hardware operation equivalent to shifting the segment register 4 bits (multiplying by 16) and adding that to the offset value. This allows the 8086 to have a larger address space, yet retain the 16-bit registers of the 8080 for backward compatibility.

The same architecture is used for the 80286 processor, except that in the processor’s “protected mode” the 16-bit segment base values are shifted over 8 bits instead of 4 as in the 8086 or in the 80286’s “real mode.” The 80286 thus uses a 24-bit address, capable of addressing up to 16 megabytes of memory.

This segmented architecture can complicate the development of large programs under the XENIX 86 and XENIX 286 Operating Systems. The 80386 processor with its 32-bit registers is not restricted by 64K segments; its segment size is 4096 Mbytes. It is therefore much more like non-segmented architectures such as the Motorola 68000.

However, a substantial amount of software development is done in the XENIX 86 and XENIX 286 environments. Understanding the potential stumbling blocks in the 80286 world is necessary to develop large programs effectively. Error messages such as “DGROUP allocation exceeds 64K,” “Not enough core,” and “Too big” can be incomprehensible without an understanding of segment usage under XENIX System V.

There are two types of segments under XENIX. Text segments (also called code segments) contain the actual machine instructions for the program. Data segments contain all the programs data, such as global variables and the stack. Under XENIX, the program’s stack is included in the first data segment. A program’s “memory model” determines how many text and data segments the program is allowed to have.

5.1.1 Memory Model Considerations

If you do not specify a memory model, `cc` uses the small memory model by default. This is sufficient for most programs.

You cannot use the small memory model if your program meets one or more of the following three conditions:

1. Your program has more than 64K of code.
2. Your program has more than 64K of data.
3. Your program contains individual arrays that need to be larger than 64K.

If you decide that the small memory model will not be adequate for your program, you have four options for larger memory models:

1. You can specify one of the other standard memory models (medium, compact, large, or huge) using one of the `-M` options.
2. You can create a mixed-model program using the `near`, `far`, and `huge` keywords.
3. You can create your own customized memory model using the `-Astring` option.
4. Method 2 can be combined with either method 1 or method 3.

Note

The only memory model supported for 80386 code is the pure small model. It is important to note that all other memory models apply to only 8086 and 80286 processors. Large and huge model programs will not run on an 8086, and any program for the 8086 or 80286, of any model, will run on an 80386, although the segment size is still limited to 64K.

When generating code specifically for the 80386 processor under SCO XENIX 386, the C compiler supports only “small” model programs, but without the 64K limit, since 80386 registers are all 32 bits wide, and its segments are over four billion bytes long. All models are supported for 86/286 code.

Choosing a memory model for a program is a trade-off between size and speed. Programs of all memory models have one “near” data segment that is addressed through the processor’s DS segment register. References to data in this segment require only a 16-bit address calculation. Large and huge model programs may have one or more additional segments. However, addressing data in these “far” segments requires loading a segment register in addition to calculating the offset within the segment.

5

5.2 Near, Far, and Huge Addressing

Understanding the terms “near,” “far,” and “huge” is crucial to understanding the concept of memory models. These terms indicate how data can be accessed in the segmented architecture of the 80x86 family of microprocessors (8086, 80186, 80286).

XENIX loads the code and data allocated by your program into “segments” of physical memory. Each segment is up to 64K long. With the exception of impure small model programs, separate segments are always allocated for the program code and data. Impure small model programs fit all data and code into one segment. Except for this case, the minimum number of segments allocated for a program is two; these two segments, required for every program, are called “the default segments.” The small memory model uses only the two default segments. The other memory models discussed in this chapter allow more than one code segment per program, more than one data segment per program, or both.

In the 80x86 family of microprocessors, all memory addresses consist of two parts:

XENIX C User's Guide

1. A 16-bit number that represents the base address of a memory segment
2. Another 16-bit number that gives an offset within that segment

The architecture of the 80x86 microprocessor is such that code can be accessed within the default code or data segment using just the 16-bit offset value. This is possible because the segment addresses for the default segments are always known. This 16-bit offset value is called a "near" address, and can be accessed with a "near" pointer. Since only 16-bit arithmetic is required to access any near item, near references to code or data are smaller and more efficient.

When data or code lies outside the default segments, the address must use both the segment and offset values. Such addresses are called "far" addresses, and can be accessed by using "far" pointers in a C program. Accessing far data or code items is more expensive in terms of program speed and size, but using them allows your programs to address all memory, rather than just a 64K piece.

There is a third type of address in XENIX C: the "huge" address. A huge address is similar to a far address in that each consists of a segment value and an offset value; but the two differ in the way address arithmetic is performed on pointers. Because items (both code and data) referenced by far pointers are still assumed to lie completely within the segment in which they start, pointer-arithmetic is done only on the offset portion of the address. This gain in pointer arithmetic efficiency is achieved, however, by limiting the size of any single item to 64K. With data items, huge pointers overcome this size limitation; pointer arithmetic is performed on all 32 bits of the data item's address, thus allowing data items referenced by huge pointers to span more than one segment, provided they conform to the rules outlined in the section on "Creating Huge-Model Programs."

The rest of this chapter deals with the various methods you can use to control whether your program makes far, near, or huge calls to access code or data.

5.3 Using the Standard Memory Models

The standard libraries provided with the XENIX Development System support five standard memory models. Using the standard memory models is the simplest way to control how your program accesses code and data in memory.

When you use the standard memory models, the compiler handles library support for you. The library corresponding to the memory model you

specify is used automatically. Each memory model has its own library, except for the huge memory model, which uses the large-model library.

The advantage of using standard models for your programs is simplicity. In the standard models, memory management is specified by compiler options; since the standard models do not require the use of extended keywords, they are the best way to write code that can be ported to other systems (particularly systems that do not use segmented architectures).

The disadvantage of using standard memory models exclusively is that they may not produce the most efficient code. For example, if you have an otherwise small-model program containing a large array that pushes the total data size for your program over the 64K limit for small-model, it may be to your advantage to declare the one array with the **far** keyword, while keeping the rest of the program small model, as opposed to using the standard compact-memory model for the entire program. For maximum flexibility and control over how your program uses memory, you can combine the standard-memory-model method with the **near**, **far**, and **huge** keywords described in “Using the **near**, **far**, and **huge** Keywords.”

The **-M** option for **cc** is used to specify one of the five standard memory models (small, medium, compact, large, or huge) at compile time. These options are discussed in the next five sections.

5

Note

In the following sections, which describe in detail the different memory-model addressing conventions, it is important to keep in mind two common features of all five models:

1. No *single* source module can generate 64K or more of code.
2. No *single* data item can exceed 64K, unless it appears in a huge-model program or it has been declared with the **huge** keyword.

5.3.1 Porting Considerations

When porting software to XENIX System V on Intel processors from other operating systems or other processors, it is important to recognize the differences that arise from the Intel-segmented architecture. One common assumption is that an integer occupies the same number of bytes as a

pointer. While this is true for small models, it is not true for middle and large models, and can cause many problems. Another common practice is to use the integer 0 to denote a null pointer. For large and huge model programs, 0 must be typecast to an appropriate pointer (typically a pointer to a **char**, such as **(char *)0**) to assure that operations with pointers work correctly.

5.3.2 Creating Small-Model Programs

Option

-Ms

The small-model option tells the compiler to create a program that occupies one segment for both code and data. (Impure Small Model)

Impure small-model programs are typically C programs that are short or have a limited purpose. Since code and data for these programs is limited to 64K, the total size of a small-model program can never exceed 64K. Most programs fit easily into this model. Using the **-i** flag, you can create a pure small model program. A pure small model program has one segment of code and one segment of data for a total of 128K.

The default in small-model programs is that both code and data items are accessed with near addresses. You can override the default for data by using the **far** or **huge** keyword, and the default for code by using the **far** keyword. (The **huge** keyword is relevant only to data items—specifically, arrays and pointers to arrays).

The compiler creates small-model programs by default when you do not specify a memory model. The **-Ms** option is provided for completeness; you need never give it explicitly unless you have added one of the other **-M** options to *etc/default/cc*.

Impure Small Model

An “impure” program is one in which both text and data occupy the same physical segment. Impure programs can be created for the 8086, 80186 or 80286 processor. There are no impure 80386 programs. The maximum program size is 64K. The **cc** program creates impure small-model programs by default on 8086/80286 systems. They can also be created using the **-Ms** option.

Pure Small Model

A “pure” program is one where text and data are in separate segments. The text is read-only and may be shared by several processes at once. On 8086/80186/80286 processors, the maximum program size is 128K (64K code + 64K data). On the 80386 processor, the maximum program size is 8 gigabytes (4G code plus 4G data). Pure small-model programs are created using the `-i` option. In this context, `-i` stands for “instruction” rather than “impure”. This is the default on 80386 systems.

5.3.3 Creating Medium-Model Programs

Option

`-Mm`

The medium-model option provides a single segment for program data, and multiple segments for program code. Each source module is given its own code segment.

Medium-model programs are typically C programs that have a large number of program statements (more than 64K of code), but a relatively small amount of data (less than 64K). Program code can occupy any amount of space and is given as many segments as needed; total program data cannot be greater than 64K. The medium model provides a useful trade-off between speed and space, since most programs refer more frequently to data items than to code.

5

5.3.4 Creating Compact-Model Programs

Option

`-Mc`

The compact-model option directs the compiler to allow multiple segments for program data but only one segment for the program code.

Compact-model programs are typically C programs that have large amounts of data, but relatively small numbers of program statements. Program data can occupy any amount of space and are given as many segments as needed.

XENIX C User's Guide

The default in compact-model programs is that code items are accessed with near addresses and data items are accessed with far addresses. You can override the default by using the **near** and **huge** keywords for data, and the **far** keyword for code.

Note

Note that in medium and compact models, **NULL** must be used carefully in certain situations. **NULL** actually represents a null data pointer. In memory models where code and data pointers are the same size, it can be used with either. However, in memory models where code and data pointers are different sizes, this is not the case. Consider the following example:

```
void func1(char *dp)
{
.
.
.
}

void func2(char (*fp)(void))
{
.
.
.
}

main()
{
func1(NULL);
func2(NULL);
}
```

This example passes a 16-bit pointer to both *func1* and *func2* if compiled in medium model, and a 32-bit pointer to both *func1* and *func2* if compiled in compact model, unless prototypes are added to the beginning of the program to indicate the types, or an explicit cast is used on the argument to *func1* (compact model) or *func2* (medium model).

5.3.5 Creating Large-Model Programs

Option

-MI

The large-model option allows the compiler to create multiple segments as needed for both code and data.

Large-model programs are typically very large C programs that use a large amount of data storage during normal processing.

The default in large-model programs is that both code and data items are accessed with far addresses. You can override the default by using the **near** and **huge** keywords for data, and the **near** keyword for code.

5.3.6 Creating Huge-Model Programs

Option

-Mh

The huge-model option is similar to the large-model option, except that the restriction on the size of individual data items is removed for arrays.

However, some size restrictions apply to elements of huge arrays where they are larger than 64K. To provide efficient addressing, array elements are not permitted to cross segment boundaries. This has the following implications:

1. No array element can be larger than 64K.
2. For any array larger than 128K, all elements must have a size in bytes equal to a power of 2 (that is, 2 bytes, 4 bytes, 8 bytes, 16 bytes, and so on). However, if the array is 128K or smaller, its elements may be any size, up to and including 64K.

In huge-model programs, care must be taken when using the **sizeof** operator or when subtracting pointers. The C language defines the value returned by the **sizeof** operator to be an **unsigned int** value, but the size in bytes of a huge array is an **unsigned long** value. To solve this discrepancy, the XENIX C Compiler produces the correct size of a huge array when a type cast like the following is used:

```
(unsigned long) sizeof(huge_item)
```



Similarly, the C language defines the result of subtracting two pointers as an **int** value. When subtracting two huge pointers, however, the result may be a **long int** value. The XENIX C Compiler gives the correct result when a type cast like the following is used:

```
(long) (huge_ptr1 - huge_ptr2)
```

5.3.7 Segmentation Errors

When compiling a small- or medium-model program, the compiler places all data in the data segment. However, the compiler cannot know how much total data is allocated in the segment. This is not determined until link time, when data from all the object modules are combined by the linker. If the linker finds that more than 64K have been allocated by the compiler, the linker will return the error message:

```
DGROUP allocation exceeds 64K
```

Errors with Small- and Medium-Model Programs

If this error occurs with a small- or medium-model program, there are three alternatives:

- Simply reduce the amount of data used by the program.
- Switch to the large-memory model.
- Create a hybrid-model program.

Hybrid models are created by declaring data using the “far” keyword and compiling with the **-Me** flag. The compiler then allocates additional segments for the far data. Care must be taken when referencing data declared in this manner. Since all the library functions will be expecting near data, far data must be transferred into a near data buffer before being passed to any library function, such as `printf()`. The hybrid model is best suited for programs with one or more large, seldom-used arrays or data structures where the rest of the program uses less than 64K of data.

Errors with Large-Model Programs

For large-model programs, the compiler divides different kinds of data into different segments. All initialized data is placed in DATA segments. Uninitialized data is placed in BSS (Blank Storage Space) segments. A large-model program may have as many DATA and BSS segments as needed, but only one near DATA segment (the segment addressed by the CPU DS register). For maximum efficiency, the compiler allocates as much data as possible to the first DATA segment. However, since the total amount of data is not known until all the object modules are linked together, more than 64K of data might be allocated for the first DATA segment. Thus, it is still possible to get the error DGROUP allocation error from the linker even with a large-model program.

One possible solution to this problem is to reduce the amount of initialized data in the program by declaring it uninitialized, then initializing at runtime. Another possibility is to use the **-Mt** flag to force the compiler to move some data out of the DATA segment. Normally, the compiler places any initialized data item (single variable, array or structure) in the first data segment if its size is less than 32767 bytes. The **-Mt** flag will lower this limit. For example, **-Mt1024** tells the compiler to place any data item larger than 1024 bytes in its own segment. The drawback to this solution is that, at runtime, a segment register must be loaded for each access to that data. This may affect performance of the program. This method is most appropriate if the program contains a few large arrays or structures.

Another method of reducing the size of the first DATA segment is the use of the **-ND** compiler flag. (See “Setting Up Segments” in this chapter.) When a module is compiled with this flag, all the data in the module will be placed in its own data segment. Modules compiled using this flag should contain data only, or data and functions that do not use any data items declared in other modules.

80286 programs allocate their maximum stack size at runtime; the default size is 4K. Since the stack must also fit in the first data segment, a problem will arise if there is not enough space in the first data segment to fit both the data and the stack. If the size of the data plus the size of the stack exceeds 64K, then, even if the linker will successfully link a program, the program’s first data segment will be too large for the program to run. This problem will be reported by the C shell with the message “Not enough core.” The Bourne shell will report the error with the message “too big.” The two possible solutions to this problem are to reduce the stack size, or to reduce the amount of data in the first data segment. The latter method is recommended, since reducing the stack size may cause the program to run out of stack space.

Determining Segment Size

There are three utilities that are useful for finding and correcting problems related to program segmentation. The `size` utility, `size(CP)`, takes one or more executable or object file names as arguments, and prints the size of the text, DATA, and BSS segments in bytes. This information is helpful in determining exactly how much data is used by a program, and how it is divided between the DATA and BSS segments. The `hdr` utility, `hdr(C)`, prints other information about an executable file, such as its memory model and stack size. The `fixhdr` utility, `fixhdr(CP)`, can be used (among other things) to alter the stack size of any executable. This is useful for experimenting with different stack sizes without the need to relink, or for cases where the source code is not available.

5.4 Using the `near`, `far`, and `huge` Keywords

One limitation of the predefined memory-model structure is that, when you change memory models, all data and code address sizes are subject to change. However, the XENIX C Compiler lets you override the default addressing convention for a given memory model and access items with a `near`, `far`, or `huge` pointer. This is done with the `near`, `far`, and `huge` keywords. These special type modifiers can be used with a standard memory model to overcome addressing limitations for particular data or code items, or to optimize access to these items, without changing the addressing conventions for the program as a whole. Table 5.1 explains how the use of these keywords affects the addressing of code or data, or pointers to code or data.

Table 5.1
Addressing of Code and Data
Declared with near, far, and huge

Key-word	Data	Pointer Function	Arithmetic
near	Reside in default data segment; referenced with 16-bit addresses (Pointers to data are 16 bits)	Assumed to be in current code segment; referenced with 16-bit addresses (Pointers to functions are 16 bits)	Uses 16 bits
far	May be anywhere in memory, not assumed to reside in current data segment; referenced with 32-bit addresses (Pointers to data are 32 bits)	Not assumed to be in current code segment; referenced with 32-bit address (Pointers to functions are 32 bits)	Uses 16 bits
huge	May be anywhere in memory, not assumed to reside in current data segment; individual data items (arrays) can exceed 64K in size; referenced with 32-bit addresses (Pointers to data are 32 bits)	Not applicable to code	Uses 32 bits for data

5

Note

The **near**, **far**, and **huge** keywords are not standard parts of the C language; they are meaningful only for systems that use a segmented architecture similar to that of the 80x86 microprocessors. Keep this in mind if you want your code to be ported to other systems.

XENIX C User's Guide

In the XENIX C Compiler, the **near**, **far**, and **huge** keywords are enabled by default. To treat these keywords as ordinary identifiers, you must give the **-Za** option at compile time. This option is useful if you are concerned with porting C programs from environments in which these are not keywords for instance, if you are porting a program in which one of these words is used as a label. For further information about the use and effects of the **-Za** option, see the "Compiling with the cc Command" chapter of this guide.

5.4.1 Library Support for near, far, and huge

When using the **near**, **far**, and **huge** keywords to modify addressing conventions for particular items, you can usually use one of the standard libraries (small, compact, medium, or large) with your program. The large-model libraries are also appropriate for use with huge-model programs. However, you must use care when calling library routines. In general, you cannot pass far pointers, or the addresses of far data items, to a small-model library routine. Of course, you can always pass the *value* of a far item to a small-model library routine. For example:

```
long far time_val;

time(&time_val);      /* Illegal */
printf("%ld\n", time_val); /* Legal */
```

If you use the **near**, **far**, or **huge** keyword, it is strongly recommended that you use function prototypes with argument-type lists to ensure that all pointer arguments are passed to functions correctly. See the section on "Pointer Conversions," for more information.

To learn more about library routines and memory models, see the *XENIX C Library Guide*.

5.4.2 Declaring Data with near, far, and huge

The **near**, **far**, and **huge** keywords modify either objects or pointers to objects. When using them to declare data or code (or pointers to data or code), keep the following rules in mind:

- The keyword always modifies the object or pointer immediately to its right. In complex declarations, think of the **far** keyword and the item to its right as being a single unit. For example, in the case of the declaration:

```
char far* *p;
```

p is a pointer (whose size depends on the specified memory model) to a far pointer to **char**. See the *XENIX C Language Reference* for complete rules governing the use of special keywords in complex declarations.

- If the item immediately to the right of the keyword is an identifier, the keyword determines whether the item will be allocated in the default data segment (**near**) or a separate data segment (**far** or **huge**). For example:

```
char far a;
```

allocates a as an item of type **char** with a far address.

- If the item immediately to the right of the keyword is a pointer, the keyword determines whether the pointer will hold a near address (16 bits), a far address (32 bits), or a huge address (also 32 bits). For example,

```
char far *p;
```

allocates p as a far pointer (32 bits) to an item of type **char**.

Examples

The examples in this section show data declarations using the **near**, **far**, and **huge** keywords.

```
char a[3000];           /* small-model program */
char far b[30000];
```

The first declaration in the example allocates the array a in the default data segment. By contrast, the array b in the second declaration may be allocated in any far data segment. Since these declarations appear in a small-model program, array a probably represents frequently used data that was deliberately placed in the default segment for fast access. Array b probably represents seldom used data that might make the default data segment exceed 64K and force the programmer to use a larger memory model if the array were not declared with the **far** keyword. The second declaration uses a large array, because it is more likely that a programmer would want to specify the address allocation size for items of substantial size.

```
char a[3000];           /* large-model program */
char near b[3000];
```



XENIX C User's Guide

In this example, access speed would probably not be critical for array *a*. Even though it may or may not be allocated within the default data segment, it is always referenced with a 32-bit address. Array *b* is explicitly allocated **near** to improve speed of access in this memory model (large).

```
char huge a[70000];    /* small-model program */
char huge *pa;
```

In this small-model program, *a* must be declared as **huge** because it is larger than 64K. Using the **huge** keyword instead of the standard huge memory model means that the price for using huge data is only paid for this one large item. Other data can be accessed quickly within the default segment. The pointer *pa* could be used to point to *a*. Any pointer arithmetic for *pa* (such as *pa++*) would be performed using 32-bit arithmetic.

```
char *pa;              /* small-model program */
char far *pb;
```

The pointer *pa* is declared as a near pointer to **char** in the example. The pointer is near by default since the example appears in a small-model program. By contrast, *pb* is allocated as a far pointer to **char**; *pb* could be used to point to, and step through, an array of characters stored in a segment other than the default data segment. For example, *pa* might be used to point to array *a* in the first example, while *pb* might be used to point to array *b*.

```
char far * *pa;       /* small-model program */
char far * *pb;       /* large-model program */
```

The pointer declarations in the example illustrate the interaction between the memory model chosen and the **near** and **far** keywords. Although the declarations for *pa* are identical, in a small-model program, *pa* is declared as a near pointer to an array of far pointers to type **char**, while in a large-model program, *pa* is declared as a far pointer to an array of far pointers to type **char**.

```
char far * near *pb;  /* any model */
char far * far *pb;
```

In the first declaration in the example, *pb* is declared as a near pointer to an array of far pointers to type **char**; in the second declaration, *pb* is declared as a far pointer to an array of far pointers to type **char**. Note that, in this example, the **far** and **near** keywords override the model-specific addressing conventions shown in the example preceding the one above; the declarations for *pb* would have the same effect, regardless of

the memory model. The examples in the following table illustrate the **far** and **near** keywords as used in declarations in a small-model program. It also gives the size in bits of the address and the value and the type of the value.

Table 5.2
Uses of 8086/80186/80286 **near** and **far** Keywords

	Size of	Size of	Type of Value
Declaration	Address	Value	Type of Value
char c;	16	8	data
char far d;	32	8	data
char *p;	16	16	near pointer
char far *q;	16	32	far pointer
char * far r;	32	16	near pointer ¹
char far * far s;	32	32	far pointer ²
int foo();	16	16	integer function
int far foo();	32	16	integer function ³

Notes

- 1 This example of a near 16-bit pointer which may lie in a far data segment is unlikely to be useful; it is shown for syntactic completeness only.
- 2 This is similar to accessing data in a large-model program.
- 3 This example leads to trouble in most environments. The far call changes the CS register, and makes run time support unavailable.

The following example is from a middle-model compilation:

```
int near foo();
```

This allows a near call (to the routine *foo*) in a program where calls are normally **far**.

If you are using one of the keywords, it would be advisable to check the type of item in separate source files as the compiler does not do this.

If the **-M3e** option is used, the **near** keyword can address items in the program segment itself and the **far** keyword can address items in segments other than the one in which the program resides. The **near** keyword

defines an item with a 32-bit address (relative to **DS**). The **far** keyword defines an item with a 48-bit address. Any data item, construct, or function can be addressed.

These keywords override the normal address length generated by the compiler for variables and functions. In pure-text small-model programs, **far** lets you access data and functions in segments outside the **PATH** and **DATA** segments.

The examples in the table that follows show **near** and **far** keywords used in declarations of pure-text small- and mixed-model programs configured with the **-M3e** option:

Table 5.3
Uses of 80386 near and far Keywords

Declaration	Address Size	Allocation Size
char c;	near (32 bits)	8 bits (data)
char far d;	far (48 bits)	8 bits (data)
char *p;	near (32 bits)	32 bits (near pointer)
char far *q;	near (32 bits)	64 bits (far pointer)
char * far r;	far (48 bits)	32 bits (near pointer) ¹
char far * far s;	far (48 bits)	64 bits (far pointer) ²
int foo();	near (32 bits)	function returning 32 bits
int far foo();	far (64/48 bits)	function returning 32 bits ³

Notes

- 1 This example is shown for syntactic completeness only.
- 2 This resembles accessing data in a large-model program.
- 3 This example creates problems in most environments. The far call changes the CS register, and makes run-time support unavailable.

5.4.3 Declaring Functions with the near and far Keywords

The rules for using the **near** and **far** keywords for functions are similar to those for using them with data, as specified in the following list:

- The keyword always modifies the function or pointer immediately to its right. For more information about rules for evaluating complex declarations, see the *XENIX C Language Reference*.

- If the item immediately to the right of the keyword is a function, then the keyword determines whether the function will be allocated as near or far. For example:

```
char far fun( );
```

defines *fun* as a function called with a 32-bit address and returning type **char**.

- If the item immediately to the right of the keyword is a pointer to a function, then the keyword determines whether the function will be called using a near (16-bit) or far (32-bit) address. For example:

```
char (far * pfun)( );
```

defines *pfun* as a far pointer (32 bits) to a function returning type **char**.

- Function declarations must match function definitions.
- The **huge** keyword cannot be applied to functions.

Examples

5

```
void char far fun(void);           /* small model */
void char far fun(void)
{
  .
  .
  .
}
```

In this example, *fun* is declared as a function returning type **char**. The **far** keyword in the declaration means that *fun* must be called with a 32-bit call.

```
static char far * near fun( );          /* large model */
static char far * near fun( )
{
    .
    .
    .
}
```

In the large-model example, *fun* is declared as a near function that returns a far pointer to type **char**. Such a function might be seen in a large-model program as a helper routine that is used frequently, but only by the routines in its own module. Since all routines in a given module share the same code segment, the function could always be accessed with a near call. However, you could not pass a pointer to *fun* as an argument to another function outside the module in which *fun* was declared.

```
void far *fun(void);                    /* small model */
void (far * pfun) ( ) = fun;
```

The small-model example declares *pfun* as a far pointer to a function that has a **void** return type, and then assigns the address of *fun* to *pfun*. In fact, *pfun* could be used to point to any function accessed with a far call. Note that if the function indicated by *pfun* has not been declared with the **far** keyword, or if it is not far by default, then calling that function through *pfun* would cause the program to fail.

```
double far * (far fun) ( );             /* compact model */
double far * (far *pfun) ( ) = fun;
```

In this final example, *pfun* is declared as a far pointer to a function that returns a far pointer to type **double**, and then assigns the address of *fun* to *pfun*. This might be used in a compact-model program for a function that is not used frequently and thus does not need to be in the default code segment. Both the function and the pointer to the function must be declared with the **far** keyword.

5.4.4 Pointer Conversions

Passing pointers as arguments to functions may cause automatic conversions in the size of the pointer argument, since passing a pointer to a function forces the pointer size to the larger of the following two sizes:

- The default pointer size for that type, as defined by the memory model used during compilation

For example, in medium-model programs, data-pointer arguments are near by default, and code-pointer arguments are far by default.

- The type of the argument

If a function prototype with argument types is given, the compiler performs type-checking and enforces the conversion of actual arguments to the declared type of the corresponding formal argument. However, if no declaration is present or the argument-type list is empty, the compiler will convert pointer arguments automatically to the default type or the type of the argument whichever is larger. To avoid mismatched arguments, you should always use a prototype with the argument types.

Examples

```

/* This program produces unexpected results in compact-,
** large-, or huge-model programs.
*/

main( )
{
    int near *x;
    char far *y;
    int z = 1;

    test_fun(x, y, z); /* x will be coerced to far
                       ** pointer in compact, large,
                       ** or huge model
                       */
}

int test_fun(ptr1, ptr2, a)
    int near *ptr1;
    char far *ptr2;
    int a;

{
    printf("Value of a = %d\n", a);}

```

If the preceding example is compiled as a small-model program (with no memory-model options or the **-Ms** option on **cc** command line) or medium-model program (**-Mm** option), then the size of pointer argument *x* is 16 bits, the size of pointer argument *y* is 32 bits, and the value printed for *a* is 1. However, if the preceding example is compiled with the **-Mc**, **-Ml**, or **-Mh** option, both *x* and *y* are automatically converted to far pointers when they are passed to *test_fun*. Since *ptr1*, the first parameter of *test_fun*, is defined as a near-pointer argument, it takes only 16 bits of the 32 bits passed to it. The next parameter, *ptr2*, takes the remaining 16 bits passed to *ptr1*, plus 16 bits of the 32 bits passed to it. Finally, the third parameter, *a*, takes the left-over 16 bits from *ptr2*, instead of the value of *z* in the *main* function. This shifting process does not generate an error message, since both the function call and the function definition are

legal, but in this case the program does not work as intended, since the value assigned to *a* is not the value intended.

To pass *ptr1* as a near pointer, you should include a forward declaration that specifically declares this argument for *test_fun* as a near pointer, as shown in the following example:

```
/* First, declare test_fun so the compiler knows in advance
** about the near pointer argument:
*/
int test_fun(int near*, char far *, int);

main( )
{
    int near *x;
    char far *y;
    int z = 1;

    test_fun(x, y, z); /* now, x will not be coerced
                       ** to a far pointer; it will be
                       ** passed as a near pointer,
                       ** no matter what memory
                       ** model is used
                       */
}

int test_fun(ptr1, ptr2, a)
    int near *ptr1;
    char far *ptr2;
    int a;

{
    printf("Value of a = %d\n", a);
}
```

Note that it would not be sufficient to reverse the definition order for *test_fun* and *main* in the first example to avoid pointer coercions; the pointer arguments must be declared in a forward declaration, as in the second example.

5.5 Creating Customized Memory Models

A third method of managing memory models is to combine features of the standard memory models to create your own customized memory model. You should have a thorough understanding of C memory models and the architecture of 8086 and 80286 processors before creating your own non-standard memory models, since there is no library support—other than the C start-up routines—for nonstandard memory models.

The *-Astring* option lets you change the attributes of the standard memory models to create your own memory models. The three letters in *string* correspond to the code-pointer size, the data-pointer size, and the stack-and data-segment setups, respectively. Because the letter allowed in each field is unique to that field, you can give the letters in any order after *-A*. All three letters must be present.

The standard-memory-model options (*-Ms*, *-Mm*, *-Mc*, *-Ml*, and *-Mh*) can be specified in the *-Mstring* form. As an example of how to construct memory models, the standard-memory-model options are listed with their standard equivalents:

Standard	Custom Equivalent
<i>-Ms</i>	<i>-Asnd</i>
<i>-Mm</i>	<i>-Alnd</i>
<i>-Mc</i>	<i>-Asfd</i>
<i>-Ml</i>	<i>-Alfd</i>
<i>-Mh</i>	<i>-Alhd</i>

As an example of the use of customized models, you might want to create a huge-compact model. This model would allow huge data items, but only one code segment. The option for specifying this model would be *-Ashd*.

An even more common use of customized models is to set up segments. (See the section on “Setting Up Segments,” for more information).

If you use a customized memory model for a program that includes both far and near functions, be aware of the following issues:

- The **chkstk** library function should be called only in functions that are compiled in the same model as the library being used. (For compatibility with XENIX, the **chkstk** function name cannot be model-encoded.)
- The interfaces to floating-point function calls are not model encoded, so the same restriction is placed on functions containing floating-point calls: they must be compiled with the same model as the library being used.

Note

For the purposes of the descriptions that follow, the letters **l** (for “long”) and **s** (for “short”) are used as code pointers to distinguish them from the letters for data pointers in the memory-model string.

5.5.1 Code Pointers

Options

-Asxx	Near code pointers
-Alxx	Far code pointers

The letter **s** tells the compiler to generate near (16-bit) pointers and addresses for all code items. This is the default for small- and compact-model programs.

The letter **l** means that far (32-bit) pointers and addresses are used to address all code items. Far pointers are the default for medium-, large-, and huge-model programs.

5.5.2 Data Pointers

Options

-Anxx	Near data pointers
-Afxx	Far data pointers
-Ahxx	Huge data pointers

Three sizes are available for data pointers: near, far, and huge. The letter **n** tells the compiler to use near (16-bit) pointers and addresses for all data. This is the default for small- and medium-model programs.

The letter **f** specifies that all data pointers and addresses are far (32-bit). This is the default for compact- and large-model programs.

The letter **h** also specifies that all data pointers and addresses are far (32-bit). This is the default for huge-model programs.

When far data pointers are used, no single data item may be larger than a segment (64K) because address arithmetic is performed only on 16 bits

(the offset portion) of the address. When huge data pointers are used, individual data items can be larger than a segment (64K) because address arithmetic is performed on the entire 32 bits of the address.

5.5.3 Setting Up Segments

Options

-Adxx	Sets SS = DS
-Au[xxx]	Sets SS != DS ; DS reloaded on function entry
-Aw[xxx]	Sets SS != DS ; DS not reloaded on function entry

The letter **d** tells the compiler that the segment addresses stored in the **SS** and **DS** registers are equal; that is, the stack segment and the default data segment are combined into a single segment. This is the default for all programs. In small- and medium-model programs, the stack plus all data must occupy less than 64K; thus, any data item is accessed with only a 16-bit offset from the segment address in the **SS** and **DS** registers.

In compact-, large-, and huge-model programs, initialized global and static data are placed in the default data segment. The address of this segment is stored in the **DS** and **SS** registers. All pointers to data, including pointers to local data (the stack), are full 32-bit addresses. This is important to remember when passing pointers as arguments in large-model programs. Although you may have more than 64K of total data in these models, there can be no more than 64K of data in the default segment. The **-Gt** and **-ND** options can be used to control allocation of items in the default data segment if a program exceeds this limit. (For more information about these options, see the section on “Setting the Data Threshold,” and “Naming Modules and Segments.”)

The letter **u** allocates different segments for the stack and the data segments. Each object file (module) is allocated its own segment for global and static data items. Note that the **-ND** option, described in “Naming Modules and Segments,” must be specified along with the letter **u** to allocate data segments other than the default. When the letter **u** is specified with **-ND**, the address in the **DS** register is saved upon entry to each function, and the new **DS** value for the module in which the function was defined is loaded into the register. The previous **DS** value is restored on exit from the function. Therefore, only one data segment is accessible at any given time. The **-ND** option can be used to combine these segments into a single segment.



If a standard memory-model option precedes it on the command line, the **-Au** option can be specified without any letters indicating data- or code-pointer sizes. In this case, the program uses the specified memory model, but different segments are set up for the stack and data segments.

A single segment must be allocated for the stack, and its address stored in the **SS** register. The stack segment does not change throughout the entire program.

The letter **w**, like the letter **u**, sets up a separate stack segment, but does not automatically load the **DS** register at each module entry point. This option is typically used when writing application programs that interface with an operating system or with a program running at the operating-system level. The operating system or the program running under the operating system actually receives the data intended for the application program and places that data in a segment; then the operating system or program must load the **DS** register with the segment address for the application program.

As with the **-Au** option, the **-Aw** option can be specified without data- and code-pointer letters if a standard memory-model option precedes it on the command line. In this case, the program uses the specified memory model, but different segments are set up for the stack and data segments, and the **DS** register is not reloaded at each module entry point.

Even though **u** and **w** set up a separate segment for the stack, the stack's size is still fixed at the default unless this is overridden with the **-F** compiler option.

5.5.4 Library Support for Customized Memory Models

Most C programs make function calls to the routines in the C run-time library. Library support is provided for the five standard memory models (small, medium, compact, large, and huge) through four separate run-time libraries. (Huge and large models both use the large-model library.) When you write mixed-model programs, you are responsible for determining which library (if any) is suitable for your program and for ensuring that the appropriate library is used.

Library support is provided for customized memory models where the stack and default data segments are combined into a single segment (**-Adxx**), but not for customized memory models where these segments are different (**-Auxx**, **-Awxx**, **-Au**, and **-Aw**). In the latter cases, you probably need to create a customized library to be used with your customized memory model. Specify the library files and object files you want to use when linking. Be sure to use the start-up routine from the appropriate

library for your memory model. Table 5.2 shows the libraries from which to extract the start-up routine for each customized memory model.

Table 5.4
Start-Up Routines for
Customized Memory Models

<u>Memory-Model Option</u>	<u>Use Start-Up from Library</u>
-Asnx; -MS plus -Ax ¹	/usr/lib/286/Sseg.o
-Asfx; -Ashx; -MC ¹ plus -Ax	/usr/lib/286/Cseg.o
-Alnx; -MM plus -Ax ¹	/usr/lib/286/Mseg.o
-Alfx; -Alhx; -ML plus -Ax; -MH plus -Ax ¹	/usr/lib/286/Lseg.o

Notes

- ¹ *x* must be either **u** or **w**.

In general, library functions do not support customized memory models, since a particular run-time routine may in turn call another library routine that conflicts with your customized model.

5.6 Setting the Data Threshold

Option

-Gt[*number*]

By default, the compiler allocates all static and global data items within the default data segment in the small and medium memory models. In compact-, large-, and huge-model programs, only *initialized* static and global data items are assigned to the default data segment. The **-Gt** option causes all data items whose sizes are greater than or equal to *number* bytes to be allocated to a new data segment. When *number* is specified, it must follow the **-Gt** option immediately, with no intervening spaces. When *number* is omitted, the default threshold value is 256. When the **-Gt** option is omitted, the default threshold value is 32,767.

You can use the **-Gt** option only with compact-, large-, and huge-model programs, since small- and medium-model programs have only one data



segment. The option is particularly useful with programs that have more than 64K of initialized static and global data in small data items.

5.7 Naming Modules and Segments

Options

- NM *module*name
- NT *text*segment
- ND *data*segment

“Module” is another name for an object file created by the C compiler. Every module has a name. The compiler uses this name in error messages if problems are encountered during processing. The module name is usually the same as the source-file name. You can change this name using the -NM (for “name module”) option. The new *module*name can be any combination of letters and digits. The space between -NM and *module*name is optional.

A “segment” is a contiguous block of binary information (code or data) produced by the C compiler. Every module except impure small has at least two segments: a text segment containing the program instructions, and a data segment containing the program data. Each segment in every module has a name. The linker uses this name to define the order in which the segments of the program appear in memory when loaded for execution. (Note that the segments in the group named **DGROUP** are an exception.)

Text and data segment names are normally created by the C compiler. These default names depend on the memory model chosen for the program. For example, in small-model programs, the text segment is named **_TEXT** and the data segment is named **_DATA**. These names are the same for all small-model modules, so all text segments from all modules are loaded as one contiguous block, and all data segments from all modules form another contiguous block.

In medium-model programs, the text from each module is placed in a separate segment with a distinct name, formed by using the module base name along with the suffix **_TEXT**. The data segment is named **_DATA**, as in the small model.

In compact-model programs, the data from each module are placed in a separate segment with a distinct name, formed by using the module base name along with the suffix **_DATA**. The exception to this is initialized global and static data, which are put in the default data segment **_DATA**. The code segment is named **_TEXT**, as in the small model.

In large- and huge-model programs, the text and data from each module are loaded into separate segments with distinct names. Each text segment is given the name of the module plus the suffix `_TEXT`. The data from each segment is placed in a private segment with a unique name (except for initialized global and static data placed in the default data segment). The naming conventions for text and data segments are summarized in Table 5.3.

Table 5.5
Segment-Naming Conventions

Model	Text	Data	Module
Small	<code>_TEXT</code>	<code>_DATA</code>	<i>filename</i>
Medium	<i>module</i> <code>_TEXT</code>	<code>_DATA</code>	<i>filename</i>
Compact	<code>_TEXT</code>	<code>_DATA</code> ¹	<i>filename</i>
Large	<i>module</i> <code>_TEXT</code>	<code>_DATA</code> ¹	<i>filename</i>
Huge	<i>module</i> <code>_TEXT</code>	<code>_DATA</code> ¹	<i>filename</i>

Notes

- ¹ `_DATA` is the name of default data segment; other data segments have unique private names.

You can override the default names used by the C compiler (thus overriding the default loading order) by using the `-NT` (for “name text”) and `-ND` (for “name data”) options. These options set to a given name the names of the text and data segments in each module being compiled. The *textsegment* argument used with the `-NT` option and the *datasegment* argument used with the `-ND` option can be any combination of letters and digits. The space between `-NT` and *textsegment*, like the space between `-ND` and *datasegment*, is optional.

If you use the `-ND` option to change the name of the default data segment, your program can no longer assume that the address contained in the stack segment register (`SS`) is the same as the address in the data segment register (`DS`). You must therefore compile your program either with the `-Mstring` form of the memory-model option and the `u` option for the segment-setup letter, or with the `-M` option for a standard memory model followed by the `-Mu` option, as in the following example:

XENIX C User's Guide

```
cc -Ms -Mu -ND DATA1 prog1.c
```

Use of the **-Mu** option forces the compiler to generate code to load **DS** with the correct data-segment value on entry to the code. See the section on "Creating Customized Memory Models," for more information on the options. All modules whose data segments have the same name have these segments combined into a single segment named *DATA1* at link time.

5.8 Specifying Text and Data Segments

Pragmas

```
#pragma alloc_text (textsegment, function1[, function2]...)  
#pragma same_seg (variable1[, variable2]...)
```

The **alloc_text** pragma gives you source-level control over the segment to which particular functions are allocated. The **same_seg** pragma provides information the compiler can use to generate better code.

If you use overlays or swapping techniques to handle large programs, **alloc_text** allows you to tune the contents of their text segments for maximum efficiency. The **alloc_text** pragma must appear before the definitions of any of the specified functions, but it may appear either before or after the functions are declared or called. Any functions specified in an **alloc_text** pragma must be either explicitly declared with the **far** keyword or assumed to be far because of the memory model used (medium, large, or huge).

The **same_seg** pragma tells the compiler to assume that the specified external variables are allocated in the same data segment. You are responsible for making sure that these variables are put in the same data segment; one way to do this is to specify the **-ND** option when you compile the program. The **same_seg** pragma must appear before any of the specified variables is used in executable code and after the variables are declared. Variables specified in a **same_seg** pragma must be explicitly declared with **extern** storage class, and they must either be explicitly declared with the **far** keyword or assumed to be far because of the memory model used (compact, large, or huge).

Chapter 6

Improving Program Speed

- 6.1 Introduction 6-1
- 6.2 Using Register Variables 6-1
- 6.3 Optimization Options and Pragmas 6-2
 - 6.3.1 Default Optimization 6-3
 - 6.3.2 Generating Intrinsic Functions 6-3
 - 6.3.3 Relaxing Alias-Checking 6-4
 - 6.3.4 Performing Loop Optimizations 6-4
 - 6.3.5 Removing Stack Probes 6-5
 - 6.3.6 Maximum Optimization 6-5
- 6.4 Choosing the Function-Calling Convention 6-5
- 6.5 Efficiency in Large Data Models 6-6
 - 6.5.1 Changing Addressing with **near**, **far**, and **huge** Keywords 6-6
 - 6.5.2 Setting the Data Threshold 6-7
 - 6.5.3 Controlling Segments Used for Allocation 6-7
- 6.6 Efficiency in Large Code Models 6-7

6.1 Introduction

This chapter describes a number of ways that you can improve the execution speed of programs compiled with the XENIX C Compiler. These techniques include:

- Using register variables
- Using optimization options and pragmas
- Choosing function-calling conventions
- Choosing and adjusting memory models

Where applicable, this chapter discusses the interactions between these techniques and the trade-offs involved in using them.

6.2 Using Register Variables

One common way to write a program for maximum speed is to declare selected local (**auto**) variables with **register** storage class. The declaration of a register variable requests the compiler to use machine registers when allocating space for the variable, if possible. The **register** storage class can be specified for any variable, but **register** specifications are ignored except for variables of type **int** or **short** or for pointer types that are the same size as type **int**.

Up to two register variables may be allocated per function. In lexical order, the compiler takes the first two variables with **register** storage class that meet the size criteria. Any later requests for **register** storage class are ignored, so be sure to declare the most important register variables first. You may also want to declare register variables in parallel scope to achieve the effect of having more than two register variables per function.

The XENIX C Compiler automatically uses registers for variables within loops. Using register declarations for such variables may interfere with optimal loop code; you can experiment with various combinations of register and nonregister declarations to determine which combinations give the best results.

Register declarations can be used effectively for values, especially pointers, that appear outside of loops. Since a certain amount of code is required to save and restore registers, register declarations must be applied to values that are accessed at least three times within a function to cause any improvement in program speed.



XENIX C User's Guide

Example

```
find_string(arr_of_chars, string)
char *string;
char *arr_of_chars[];
{
    int ix = 0;
    register char *q;
    while (*(q = string)) {      /* string is not null */
        {
            register int i = ix;

            /* search for entry whose first character
             * matches first character of string, if any
             */

            while (i < MAX_ARR_SIZE && *arr_of_chars[i] != *q)
                i++;
            if (i == MAX_ARR_SIZE)
                return(1); /* no matching entry */
            ix = i;
        }

        /* we've found an entry in arr_of_chars which
         * might match string */

        {
            register char *p = arr_of_chars[ix];
            while (*p && *q && *p++ == *q++)
                ;
            if ((*p - *q) == 0)
                return(0) /* they match, return 0 */
            /* otherwise continue checking for possible
             * matches
             */
        }
    }
}
```

In this example, the function named *find_string* actually has three register variables: *q*, *i*, and *p*. The function can use all three variables because *i* is through being used by the time *p* is needed. Introducing the *ix* variable to save the pointer from block-to-block speeds execution considerably because most work is being done in register variables.

6.3 Optimization Options and Pragas

The **cc** compiler/linker driver provides a number of optimization options (**-O**, followed by one or more letters) that can improve program speed. In addition, the XENIX C Compiler includes several pragmas that allow you to control some of these optimizations on a local basis within a source

program. The following sections outline these **cc** options and pragmas and their effects.

6.3.1 Default Optimization

If no **-O** option is given, the compiler uses the **-Ot** option, which optimizes programs for execution speed. However, this option does not enable loop optimizations or intrinsics. Some optimizations, such as long shifts, may be performed in line rather than using helper functions.

6.3.2 Generating Intrinsic Functions

The **-Oi** option generates intrinsic forms of the following functions:

- **memset, memcpy, memcmp**
- **strset, strcpy, strcmp, strcat**
- **inp, outp**
- **_rotl, _rotr, _lrotl, _lrotr,**
- **min, max, abs**

Intrinsics may be generated as in-line code or with different calling sequences. In general, using intrinsics increases program size but improves program speed. Note that the intrinsic forms of some functions may have slightly different semantics: for example, the intrinsic form of the **memcpy** function in compact- and large-model programs cannot handle huge arrays, but the function form can.

As with **-Ot**, this option may increase program size due to the additional code generated in line for each function. However, program execution is faster because no instructions for calling and returning from functions need to be performed.

The **intrinsic** pragma can be used to specify intrinsic functions on a local basis for any of the functions listed above. For information about the use of the **intrinsic** pragma, see the “Compiling with the **cc** Command” chapter of this guide.



6.3.3 Relaxing Alias-Checking

The **a** option letter can be used with the **l**, **s**, or **t** option letter to relax the assumptions the compiler makes about the use of “aliases” in the program. Use of the **-Oa** option can reduce the size of executable files and speed program execution. This is especially recommended when you also specify the **-Ol** option, since the compiler can detect a number of loop optimizations when the **-Oa** option is in effect that it cannot detect when **-Oa** is not in effect. However, before you specify **-Oa**, you must make sure that your program does not use multiple aliases to refer to the same memory location either directly or indirectly. For example, a program might do this indirectly in functions that operate on a communal variable and a pointer argument, or on multiple pointer arguments.

The **-Oa** option can be specified safely for programs that include calls to functions with address-type arguments. In this case, the compiler assumes that all variables whose addresses are passed to the function are modified, even if **-Oa** is specified.

In the cases noted above, the use of **-Oa** is most likely to cause incorrect optimizations within basic blocks (where most optimizations are applied) and within whole loop bodies (where loop optimizations are applied). In these cases, **-Oa** can still be specified safely even if aliases are used in the program, provided that no memory location is referenced by more than one name within any basic block or (if loop optimization is enabled) any loop body.

For more information and specific examples, see the “Compiling with the **cc** Command” chapter of this guide.

6.3.4 Performing Loop Optimizations

The **-Ol** option tells the compiler to perform loop optimizations. For best performance, use **-Ol** in conjunction with the **a** option letter (**-Oal**), which relaxes the assumptions the compiler makes about the use of aliases in the program. Using **-Oal** instead of just **-Ol** allows the compiler to detect many loop optimizations that it could not otherwise detect. For information about possible restrictions on the uses of the **-Oa** option, see the “Compiling with the **cc** Command” chapter of this guide.

You can control loop optimization on a local basis by specifying the **loop_opt** pragma. Loop optimization is turned off for any functions following **#pragma loop_opt(off)** and turned on for any functions following **#pragma loop_opt(on)** in a source program. This pragma overrides any loop optimization specified on the **cc** command line.

6.3.5 Removing Stack Probes

The **-Gs** option, described in the “Compiling with the **cc** Command” chapter of this guide; speeds program execution slightly by removing calls to stack-checking routines known as “stack probes.” Stack probes verify that a program has enough stack space to allocate required local variables. The potential disadvantage in removing stack probes is that stack-overflow errors may occur without generating a diagnostic message. However, this technique can be useful for programs that are known not to exceed the available stack space.

You can also control stack checking on a local basis by specifying the **check_stack** pragma. Stack checking is turned off for any functions following a **#pragma check_stack(off)** pragma and turned on for any functions following a **#pragma check_stack(on)** pragma in the source program. This pragma overrides the stack checking (or removal of stack checking) specified on the **cc** command line.

6.3.6 Maximum Optimization

The **-Ox** option combines the **-Ot**, **-Of**, **-Oa** and **-Ol** optimization options described in this section. Provided that the restrictions outlined for each optimization option do not apply, you can use the **-Ox** option to create the fastest possible program.

6.4 Choosing the Function-Calling Convention



Because C functions can accept variable numbers of arguments, arguments passed to these functions must be pushed on the stack from right to left, with the first argument in the list being the last one pushed. In addition, the calling function, rather than the called function, is responsible for removing arguments from the stack.

This convention results in somewhat slower programs than the alternative convention used by XENIX FORTRAN and XENIX Pascal. In the FORTRAN/Pascal convention, arguments are pushed on the stack from left to right, in the order in which they are passed to the function, and the called function removes arguments from the stack. Since the code for removing arguments appears only once (in the called function) for the FORTRAN/Pascal convention, rather than multiple times (every time a function is called) as in the C convention, and since most programs have fewer functions than function calls in a program, the FORTRAN/Pascal calling convention usually results in smaller, faster programs.

XENIX C User's Guide

You can specify the FORTRAN/Pascal calling convention for all functions in a module by compiling with the **-Gc** option. The trade-off for improved program speed is that you cannot call functions that use the C calling convention or take variable numbers of arguments unless you declare these functions, or pointers to these functions, with the **cdecl** keyword, which specifies the normal C calling conventions for these functions.

If you do not want to specify the FORTRAN/Pascal convention for a whole module, you can declare individual functions or pointers to functions with the **pascal** or **fortran** keyword. Either of these keywords tells the compiler that the function uses the FORTRAN/Pascal calling conventions.

6.5 Efficiency in Large Data Models

Programs are most efficient when their data reside in the default data segment, that is, when the data can be accessed with 16-bit (near) addresses. The XENIX C Compiler provides two standard memory models in which all data reside in the default data segment: the small (default) model and the medium model. The customized memory models that use near data pointers (**-Mnxx**) also restrict program data to the default data segment. Programs compiled with these models are restricted to 64K of total data.

For programs compiled with the compact, large, and huge memory models, the compiler creates a default data segment containing all initialized global and static data and creates an additional data segment for each program module. Since accessing data outside the default data segment is slower than accessing data within the default data segment, programs will run faster if as many of their variables as possible are declared in such a way that they are allocated in the default data segment. One way to accomplish this is to initialize variables at the time you declare them. This section discusses other ways of controlling the allocation of data for large data models.

6.5.1 Changing Addressing with **near**, **far**, and **huge** Keywords

The **near**, **far**, and **huge** keywords allow you to specify explicitly the addressing used for particular data items and functions. These keywords override the default addressing conventions specified by the program's memory model. Thus, you can use them to improve the speed of access to program data. For example, you can tell the compiler to allocate data items in the default data segment for a compact-, large-, or huge-model program by declaring the items (or pointers to the items) with the **near** keyword. Alternatively, if a program has a small amount of code and data

except for one particularly large array, you could compile the program with the small or medium memory model and declare the array with the **far** or **huge** keyword.

The disadvantage of using these keywords is that they are specific to the MS-DOS/XENIX implementation of XENIX C and, thus, are not portable to other operating environments.

For more information about **near**, **far**, and **huge** and for examples of their use, see the “Working with Memory Models” chapter in this guide.

6.5.2 Setting the Data Threshold

Another way to control allocation in large data models is to set a data threshold by compiling with the **-Gt** option. This option is especially useful if your program uses more than 64K of initialized static and global data and does not fit in the default data segment. Any data items larger than the value you specify are allocated to their own data segments.

6.5.3 Controlling Segments Used for Allocation

If programs compiled with large data models use external far data items, you can tell the compiler which items reside in the same far data segment by using the **same_seg** pragma. The variables you specify in this pragma help the optimizer recognize common subexpressions involving data loads. Note that you must also compile your program with the **-ND** option to ensure that the variables you specify are allocated in the same segment.

For a description of the **-ND** option and the **same_seg** pragma, see the “Working with Memory Models” chapter of this guide.

6.6 Efficiency in Large Code Models

One linker option, **-T**, can result in smaller and faster executable files and improved program-load times for programs that explicitly or implicitly use far-function calls.

The **-T** option tells the linker to optimize far calls to procedures that lie in the same segment as the caller. When you specify the **-T** option, the linker optimizes 32-bit calls to procedures in the same segment as the calling procedure. Since the segment addresses of the calling and called procedures are the same, only a 16-bit call is required. If the **-T** option is given, the linker removes the far call and replaces it with code that first places **CS** on the stack, then makes a near call. The called procedure still

returns with a far (32-bit) return instruction. However, because both the code segment (stored in **CS**) and the near address are on the stack, the far return is done correctly. The linker also adds a **NOP** instruction so that the five-byte far call is replaced by exactly five bytes of instructions.

Note

You may not want to use the **-T** option if your program includes system-level assembly-language routines or if you are linking object files that were compiled with a different C compiler.

Chapter 7

Object and Executable File Formats

- 7.1 Introduction 7-1
- 7.2 iAPX 286, 386 System Architecture 7-1
 - 7.2.1 Memory Management 7-1
 - 7.2.2 Logical Address Space 7-1
 - 7.2.3 Logical-to-Physical Address Translation 7-2
- 7.3 The Intel Object Module Format 7-2
- 7.4 Definition of Terms 7-4
- 7.5 Module Identification and Attributes 7-6
- 7.6 Segment Definition 7-7
- 7.7 Segment Addressing 7-7
- 7.8 Symbol Definition 7-8
- 7.9 Indices 7-8
- 7.10 Conceptual Framework for Fixups 7-8
- 7.11 Self-Relative Fixups 7-13
- 7.12 Segment-Relative Fixups 7-14
- 7.13 Record Order 7-15
- 7.14 Introduction to the Record Formats 7-16
 - 7.14.1 Title and Official Abbreviation 7-16
 - 7.14.2 The Boxes 7-17
 - 7.14.3 Rectyp 7-17

7.14.4	Record Length	7-17
7.14.5	Name	7-17
7.14.6	Number	7-17
7.14.7	Repeated or Conditional Fields	7-17
7.14.8	Chksum	7-18
7.14.9	Bit Fields	7-18
7.14.10	T-Module Name	7-19
7.14.11	Name	7-19
7.14.12	Seg Attr	7-20
7.14.13	Segment Length	7-22
7.14.14	Segment Name Index	7-22
7.14.15	Class Name Index	7-22
7.14.16	Overlay Name Index	7-23
7.14.17	Group Name Index	7-23
7.14.18	Group Component Descriptor	7-24
7.14.19	Name	7-25
7.14.20	Eight-Leaf Descriptor	7-25
7.14.21	Public Base	7-26
7.14.22	Public Name	7-28
7.14.23	Public Offset	7-28
7.14.24	Type Index	7-28
7.14.25	External Name	7-28
7.14.26	Type Index	7-29
7.14.27	Line-Number Base	7-30
7.14.28	Line-Number	7-30
7.14.29	Line Number Offset	7-30
7.14.30	Segment Index	7-31
7.14.31	Enumerated Data Offset	7-31
7.14.32	Dat	7-32
7.14.33	Segment Index	7-32
7.14.34	Iterated Data Offset	7-32
7.14.35	Iterated Data Block	7-33
7.14.36	Repeat Count	7-33
7.14.37	Block Count	7-33
7.14.38	Content	7-33
7.14.39	Thread	7-35
7.14.40	Fixup	7-36
7.14.41	Mod Type	7-39
7.14.42	Comment Type	7-41
7.14.43	Comment	7-42
7.15	Numeric List of Record Types	7-42
7.16	Type Representations for Communal Variables	7-43
7.17	The Segmented x.out Format	7-45

7.17.1	General Description of x.out	7-46
7.17.2	Example of File Layout	7-48
7.17.3	Iterated Segments	7-48
7.17.4	Non-Iterated Segments and Implicit bss	7-49
7.17.5	Large Model	7-49
7.17.6	Special Header Fields	7-49
7.17.7	Symbol Table	7-50
7.17.8	XENIX Executable Format	7-50
7.17.9	Selected Portions of Include Files	7-52

7.1 Introduction

This chapter is divided into three sections. The first provides you with a brief introduction to the architecture of the iAPX-286 and -386 processors.

The second section provides a discussion of the Intel (O)bject (M)odule (F)ormat, which we follow. The implementation of this format makes it possible to compile programs that run in both the XENIX and MS-DOS environments.

The third section provides a brief description of our implementation of the **x.out** format in a segmented environment. For detailed information, see the **x.out** header file.

7.2 iAPX 286, 386 System Architecture

XENIX runs on both the 80286 and the 80386 processors in protected-mode. This section provides a general introduction to the architecture of protected mode operation. It does not discuss the various 80386 paging mechanisms. For an in-depth discussion of the iAPX286 and iAPX386, refer to the appropriate Programmer's Technical Reference Manual published by Intel.

7.2.1 Memory Management

Memory management provides a mapping from the logical addresses used within a program to physical machine addresses. This serves two purposes:

- Programs are not tied to any particular physical address.
- Access permissions to particular areas of memory can be controlled.



7.2.2 Logical Address Space

The mapping of virtual addresses to physical addresses is achieved by means of descriptor tables which are themselves resident in memory. At any given moment, there are two alternate descriptor tables available: the Global Descriptor Table (GDT) and the Local Descriptor Table (LDT).

The XENIX kernel uses the GDT to map the kernel's virtual address space. Each user process has its own LDT as part of its per-process data which maps the logical address space of the process.

C User's Guide

Each entry in a descriptor table specifies the base address, length and access permissions of a particular segment of physical memory.

7.2.3 Logical-to-Physical Address Translation

Logical addresses consist of two parts: a segment selector used to select a particular descriptor table entry, and an offset added to the base address found in the descriptor table to give a physical memory address.

The segment selector is a 16-bit number containing three pieces of information:

1. The Request Privilege Level (RPL) is encoded as the low order two bits of the selector. The RPL is a feature of the system architecture protection scheme. Segment selectors in user processes always have both of these bits set to indicate RPL 3, the lowest privilege level.
2. The Table Indicator (TI) is encoded as the next most significant bit (bit 2). The TI indicates whether address translation will use the GDT (TI = 0) or the LDT (TI = 1). User processes can only access the LDT; therefore the TI for a segment selector in a user process is always 1.
3. The Index field is encoded as the high-order 13 bits of the selector. This is used to index into the appropriate descriptor table and select a particular entry.

A descriptor table entry having been selected, the offset is added to the base address in physical memory to form a physical address.

Depending on the characteristics of the segment (as defined in the descriptor table) the offset may be a 16- or 32-bit number. The offset will be 16 bits on an 80286 processor or in a 16-bit segment on an 80386 processor. The 32-bit offsets apply only to the 80386.

7.3 The Intel Object Module Format

This section presents the object record formats that define the relocatable object language for the iAPX-86 family of microprocessors. The 8086 object language is the output of all language translators that have the 8086 as their target processor and are linked by the link editor. The 8086

object language is input and output for object language processors such as linkers and librarians.

Note

Except where otherwise noted, references to the 8086 in this document refer to the 8086/80286/80386 processors. In general, the 8086/80286 references are made to 16-bit offsets and 64K segment offsets, which do not apply to the 80386.

The 8086 object module formats permit you to specify relocatable memory images that may be linked together. The formats allow efficient use of the memory-mapping facilities of the 8086 microprocessor.

The following record formats, as described in this chapter, are supported. Those formats preceded by an asterisk (*) deviate from the Intel® specification.

Object Module Record Formats

- T-Module Header Record
- List of Names Record
- *Segment Definition Record
- *Group Definition Record
- *Type Definition Record

Symbol Definition Records

- *Public Names Definition Record
- *External Names Definition Record
- *Line Numbers Record

Data Records

- Logical Enumerated Data Record
- Logical Iterated Data Record

- Fixup Record
- *Module End Record
- Comment Record

C User's Guide

7.4 Definition of Terms

The following terms are used to describe 8086 relocation and linkage.

OMF

Object Module Formats

MAS

Memory Address Space. Note that the MAS is distinguished from actual memory, which may occupy only a portion of the MAS.

MODULE

An "inseparable" collection of object code and other information produced by a translator.

T-MODULE

A module created by a translator, such as C, Pascal or FORTRAN.

The following restrictions apply to object modules:

- Every module needs a name. Translators provide names for T-Modules, giving a default name (possibly the filename or a null name) if neither source code nor user specifies otherwise.
- Every T-Module in a collection of linked modules must have a different name so that symbolic debugging systems can distinguish the various line numbers and local symbols. This restriction is not required by **ld**.

FRAME

A contiguous region of MAS that can be addressed using a single segment register. This concept is useful because the content of the four 8086 segment registers defines four (possibly overlapping) FRAMES; no 16-bit address in the 8086 code can access a memory location outside of the current four FRAMES. On an 8086, a FRAME must begin on a paragraph boundary (that is, a multiple of 16 bytes). On 80286 and 80386 processors, this restriction does not apply. On an 80386, a FRAME is a region of up to (2**32) bytes addressed by a single segment register.

LSEG

Logical Segment. A contiguous region of memory whose contents are determined at translation time (except for address-binding). Neither size nor location in MAS is necessarily determined at translation time; size, although partially fixed, may not be final because the LSEG may be combined at LINK time with other LSEGs, forming a single LSEG. On 8086/80286 processors, an

LSEG must not be larger than 64K, so that it can fit in a FRAME. This means that any byte in an LSEG may be addressed by a 16-bit offset from the base of a FRAME covering the LSEG. An 80386 LSEG may be as much as (2^{32}) bytes in size and any byte in it addressed by a 32-bit offset from the base of the FRAME containing the LSEG.

PSEG

Physical Segment. This term is equivalent to FRAME. Some people prefer PSEG to FRAME because the terms PSEG and LSEG reflect the physical and logical nature of the underlying segments.

FRAME NUMBER

This term is only used in reference to 8086 processors, or 80286/80386 processors operating in real address mode. Every FRAME begins on a paragraph boundary. The paragraphs in MAS can be numbered from 0 through 65535. These numbers, each of which defines a FRAME, are called FRAME NUMBERS.

PARAGRAPH NUMBER

This term is equivalent to FRAME NUMBER.

PSEG NUMBER

This term is equivalent to FRAME NUMBER.

GROUP

A collection of LSEGs defined at translation time, whose final locations in MAS are constrained such that there is at least one FRAME that covers (contains) every LSEG in the collection.

The notation Gr A(X,Y,Z) means that LSEGs X, Y and Z form a group whose name is A. The fact that X, Y and Z are all LSEGs in the same group does not imply any ordering of X, Y and Z in MAS, nor does it imply any contiguity between X, Y and Z.

The link editor does not currently allow an LSEG to be a member of more than one group. The link editor ignores all attempts to place an LSEG in more than one group.

CANONIC

Any location in the 8086 MAS is contained in exactly 4096 distinct FRAMES; but one of these FRAMES can be distinguished because it has a higher FRAME NUMBER. This distinguished FRAME is called "the canonic FRAME" of the location. The canonic FRAME of a given byte is the FRAME so chosen that the byte's offset from that FRAME lies in the range 0 to 15 (decimal). Thus, if FOO is a symbol defining a memory location, one may speak of the "canonic FRAME of FOO," or of "FOO's canonic

7

C User's Guide

FRAME.” By extension, if S is any set of memory locations, then there exists a unique FRAME that has the lowest FRAME NUMBER in the set of canonic FRAMEs of the locations in S. This unique FRAME is called the canonic FRAME of the set S. Thus, we may speak of the canonic FRAME of an LSEG, or of a group of LSEGs.

SEGMENT NAME

LSEGs are assigned segment names at translation time. These names serve two purposes:

- They play a role at LINK time in determining which LSEGs are combined with other LSEGs.
- They are used in assembly source code to specify groups.

CLASS NAME

LSEGs may optionally be assigned class names at translation time. Classes define a partition on LSEGs: two LSEGs are in the same class if they have the same class name.

The link editor applies the following semantics to class names. The class name “CODE” or any class name whose suffix is “CODE” implies that all segments of that class contain only code and may be considered read-only. Such segments may be overlaid if the user specifies the module containing the segment as part of an overlay.

OVERLAY NAME

LSEGs may optionally be assigned an overlay names. The overlay name of an LSEG is ignored by **ld** (version 2.40 and later versions), but it is used by Intel relocation and linkage products.

COMPLETE NAME

The complete name of an LSEG consists of the segment name, class name, and overlay name. LSEGs from different modules are combined if their complete names are identical.

7.5 Module Identification and Attributes

A module header record is always the first record in a module and provides the module name.

In addition to a name, a module may have the attribute of being a main program as well as having a specified starting address. When you are linking multiple modules together, only one module with the main attribute should be given.

In summary, modules may or may not be main and may or may not have a starting address.

7.6 Segment Definition

A module is a collection of object code defined by a sequence of records produced by a translator. The object code represents contiguous regions of memory whose contents are determined at translation time. These regions are called LOGICAL SEGMENTS (LSEGS). A module defines the attributes of each LSEG. The SEGMENT DEFINITION RECORD (SEGDEF) is the vehicle by which all LSEG information (name, length, memory alignment, and so on) is maintained. The LSEG information is required when multiple LSEGS are combined and when segment addressability is established. (See “Segment Addressing”). The SEGDEF records are required to follow the first header record.

7.7 Segment Addressing

The 8086/80286 addressing mechanism provides segment base registers from which a 64-Kbyte region of memory, called a FRAME, may be addressed. There are one code-segment base register (CS), two data-segment base registers (DS, ES), and one stack-segment base register (SS). The 80386 has two additional segment registers: FS and GS, and can address up to (2**32) bytes of memory from each segment register.

The greatest possible number of LSEGS that may make up a memory image far exceeds the number of available base registers. Thus, base registers may require frequent loading. This would occur in a modular program with many small data and/or code LSEGS.

Since such frequent loading of base registers is undesirable, it is a good strategy to collect many small LSEGS together into a single unit that fits in one memory frame so that all the LSEGS may be addressed using the same base register value. This addressable unit is a GROUP. See “Definition of Terms.”

To have addressability of objects within a GROUP, each GROUP must be explicitly defined in the module. The GROUP DEFINITION RECORD (GRPDEF) provides a list of constituent segments either by segment name or by segment attribute such as “the segment defining symbol FOO” or “the segments with class name ROM.”

C User's Guide

The GRPDEF records within a module must follow all SEGDEF records because GRPDEF records can reference SEGDEF records when defining a GROUP. The GRPDEF records must also precede all other records except header records, as **ld** must process them first.

7.8 Symbol Definition

The **ld** command supports three different types of records that fall into the class of symbol definition records. The two most important types are PUBLIC NAMES DEFINITION RECORDs (PUBDEFs) and EXTERNAL NAMES DEFINITION RECORDs (EXTDEFs). These types are used to define globally visible procedures and data items and to resolve external references. In addition, TYPDEF records are used by **ld** for the allocation of communal variables. (See "Type Representations for Communal Variables").

7.9 Indices

"Index" fields appear throughout this document. An index is an integer that selects some particular item from a collection of such items. (This is a list of examples: NAME INDEX, SEGMENT INDEX, GROUP INDEX, EXTERNAL INDEX, TYPE INDEX.)

In general, indices must assume values quite large (that is, much larger than 255). Nevertheless, a great number of object files will contain no indices with values greater than 50 or 100. Therefore, indices will be encoded in one or two bytes, as required.

The high-order (left-most) bit of the first (and possibly the only) byte determines whether the index occupies one byte or two. If the bit is 0, then the index is a number between 0 and 127, occupying one byte. If the bit is 1, then the index is a number between 0 and 32K-1, occupying two bytes, and is determined as follows: the low-order 8 bits are in the second byte, and the high-order 7 bits are in the first byte.

7.10 Conceptual Framework for Fixups

A "fixup" is some modification to object code, requested by a translator, performed by **ld**, achieving address-binding.

Note

This definition of “fixup” accurately represents the viewpoint maintained by **ld**. Nevertheless, the link editor can be used to achieve modifications of object code (that is, “fixups”) that do not conform to this definition. For example, the binding of code to either hardware floating-point or software floating-point subroutines is a modification to an operation code, where the operation code is treated as if it were an address. The previous definition of “fixup” is not intended to disallow or disparage object code modifications.

8086 translators specify a fixup with four data items:

- The place and type of a LOCATION to be fixed up.
- One of two possible fixup MODES.
- A TARGET, which is a memory address to which LOCATION must refer.
- A FRAME defining a context within which the reference takes place.

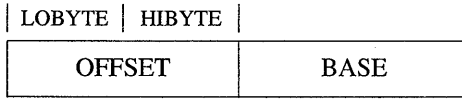
There are 5 types of LOCATION: a POINTER, a BASE, an OFFSET, a HIBYTE, and a LOBYTE.

The vertical alignment of the following figure illustrates four points. (Remember that the high-order byte of a word in 8086 memory is the byte with the higher address.) The **ld** command does not require the presence of the high- or low-order complement of these items. (For instance, in the case of HIBYTE, a high-order word, it doesn’t matter if the low-order word is present.)

- A BASE is the high-order word of a pointer.
- An OFFSET is the low-order word of a pointer.
- A HIBYTE is the high-order half of an OFFSET.

C User's Guide

- A LOBYTE is the low-order half of an OFFSET.



P O I N T E R

LOCATION Types

A LOCATION is specified by two data: (1) the LOCATION type, and (2) where the LOCATION is. The first is specified by the LOC subfield of the LOCAT field of the FIXUP record; the second is specified by the DATA RECORD OFFSET subfield of the LOCAT field of the FIXUP record.

The link editor supports two fixup MODES: "self-relative" and "segment-relative."

Self-Relative fixups support the 8- and 16-bit offsets that are used in the CALL, JUMP and SHORT-JUMP instructions. Segment-Relative fixups support all other addressing modes of the 8086.

The TARGET is the location in MAS being referenced. (More explicitly, the TARGET may be considered the lowest byte in the object being referenced.) A TARGET is specified in one of eight ways. There are four "primary" ways, and four "secondary" ways. Each primary way of specifying a TARGET uses two kinds of data: an INDEX-or-FRAME-NUMBER 'X', and a displacement 'D'.

- (T0) X is a SEGMENT INDEX. The TARGET is the Dth byte in the LSEG identified by the INDEX.
- (T1) X is a GROUP INDEX. The TARGET is the Dth byte in the LSEG identified by the INDEX.
- (T2) X is an EXTERNAL INDEX. The TARGET is the Dth byte following the byte whose address is (eventually) given by the External Name identified by the INDEX.
- (T3) X is a FRAME NUMBER. The TARGET is the Dth byte in the FRAME identified by the FRAME NUMBER (that is, the address of TARGET is $(X*16)+D$).

Each secondary way of specifying a TARGET uses only one data item: the INDEX-or-FRAME-NUMBER X. An implicit displacement equal to zero is assumed.

- (T4) X is a SEGMENT INDEX. The TARGET is the 0th (first) byte in the LSEG identified by the INDEX.

Object and Executable File Formats

- (T5) X is a GROUP INDEX. The TARGET is the 0th (first) byte in the LSEG in the specified group that is eventually LOCATED lowest in MAS.
- (T6) X is an EXTERNAL INDEX. The TARGET is the byte whose address is the External Name identified by the INDEX.
- (T7) X is a FRAME NUMBER. The TARGET is the byte whose 20-bit address is (X*16).

Note

The link editor does not support methods T3 and T7.

The following nomenclature is used to describe a TARGET:

TARGET:	SI (<segment name>), <displacement>	[T0]
TARGET:	GI (<group name>), <displacement>	[T1]
TARGET:	EI (<symbol name>), <displacement>	[T2]
TARGET:	SI (<segment name>)	[T4]
TARGET:	GI (<group name>)	[T5]
TARGET:	EI (<symbol name>)	[T6]

The following examples illustrate how this notation is used:

TARGET: SI(CODE), 1024	The 1025th byte in the segment "CODE".
TARGET: GI(DATAAREA)	The location in MAS of a group called "DATAAREA".
TARGET: EI(SIN)	The address of the external subroutine "SIN".
TARGET: EI(PAYSCHEDULE), 24	The 24th byte following the location of an EXTERNAL data structure called "PAYSCHEDULE".

Every 8086 memory reference is to a location contained within some FRAME, where the FRAME is designated by the content of some segment register. For **ld** to form a correct, usable memory reference, it must

C User's Guide

know what the TARGET is, and to which FRAME the reference is being made. Thus, every fixup specifies such a FRAME in one of six ways. Some use data X, which is in INDEX-or-FRAME-NUMBER, as above. Others require no data.

The six methods of specifying frames are:

1. (F0) X is a SEGMENT INDEX. The FRAME is the canonic FRAME of the LSEG defined by the INDEX.
2. (F1) X is a GROUP INDEX. The FRAME is the canonic FRAME defined by the group (that is, the canonic FRAME defined by the LSEG in the group that is eventually LOCATED lowest in MAS).
3. (F2) X is an EXTERNAL INDEX. The FRAME is determined when the External Name's public definition is found. There are three cases:
 - (F2a) The symbol is defined relative to some LSEG, and there is no associated GROUP. The LSEGs canonic FRAME is specified.
 - (F2b) The symbol is defined absolutely, without reference to an LSEG, and there is no associated GROUP. The FRAME is specified by the FRAME NUMBER subfield of the PUBDEF record that gives the symbol's definition.
 - (F2c) Regardless of how the symbol is defined, there is an associated GROUP. The canonic FRAME of the GROUP is specified. (The group is specified by the GROUP INDEX subfield of the PUBDEF Record.)
4. (F3) X is a FRAME NUMBER (specifying the obvious FRAME).
5. (F4) No X. The FRAME is the canonic FRAME of the LSEG containing LOCATION.
6. (F5) No X. The FRAME is determined by the TARGET. There are four cases:
 - (F5a) The TARGET specifies a SEGMENT INDEX: in this case, the FRAME is determined as in (F0).
 - (F5b) The TARGET specifies a GROUP INDEX: in this case, the FRAME is determined as in (F1).
 - (F5c) The TARGET specifies an EXTERNAL INDEX: in this case, the FRAME is determined as in (F2).

- (F5d) The TARGET is specified with an explicit FRAME NUMBER: in this case the FRAME is determined as in (F3).

Note

The link editor does not support frame methods F2b, F3, or F5d.

Nomenclature describing FRAMEs is similar to the above nomenclature for TARGETs.

FRAME:	SI (<segment name>)	[F0]
FRAME:	GI (<group name>)	[F1]
FRAME:	EI (<symbol name>)	[F2]
FRAME:	LOCATION	[F4]
FRAME:	TARGET	[F5]
FRAME:	NONE	[F6]

For an 8086 memory reference, the FRAME specified by a self-relative reference is usually the canonic FRAME of the LSEG containing the LOCATION, and the FRAME specified by a segment relative reference is the canonic FRAME of the LSEG containing the TARGET.

7.11 Self-Relative Fixups

Self-relative fixups can be applied to LOCATIONS which are a 16- or 32-bit OFFSET or LOBYTES. (The result of applying a self-relative fixup to any other type of LOCATION is undefined.)

Both the LOCATION and the TARGET must lie within the FRAME specified for the fixup.

The value to be used in the fixup is defined as the displacement from the byte in memory following the LOCATION to the TARGET.

If the LOCATION to be fixed-up is a LOBYTE, the fixup value must lie in the range -128 to 127.

C User's Guide

If the LOCATION to be fixed up is a 16-bit OFFSET, the fixup value must lie in the range -32768 to 32767.

The fixup value is added to the existing contents of the LOCATION, ignoring any overflow.

Self-relative fixups are typically applied to the relative displacement values used in instructions such as conditional jumps.

7.12 Segment-Relative Fixups

Segment-relative fixups can be applied to any type of LOCATION.

The way in which a LOCATION containing a BASE component (that is, a BASE or a POINTER) is fixed up depends on whether the code is to run in real or virtual address mode. The contents of the BASE portion of a LOCATION must ultimately be capable of being loaded into a segment register; therefore, in real address mode this will be a paragraph number and in virtual address mode this will be a selector value.

Fixup values for the BASE and OFFSET components of a LOCATION are calculated as follows:

1. In real address mode:

The base fixup value (FBVAL) is defined as the FRAME NUMBER of the FRAME specified in the fixup.

The offset fixup value (FOVAL) is defined as the offset of the TARGET from the start of the FRAME specified in the fixup. This offset must be ≥ 0 and \leq FFFF.

2. In protected mode:

The base fixup value (FBVAL) is defined as the segment selector of the FRAME specified in the fixup.

The offset fixup value (FOVAL) is defined as the offset of the TARGET from the start of the FRAME specified in the fixup. This offset must be ≥ 0 and \leq the maximum segment size implied by the segment selector for the FRAME. (that is, $(2^{**}16)-1$ for 80286 segments and 16-bit 80386 segments, or $(2^{**}32)-1$ for 32-bit 80386 segments.)

The fixup values for BASE and OFFSET are applied to the LOCATION as follows:

1. If the LOCATION is a BASE or a POINTER, then FBVAL is stored in the BASE component of the LOCATION.
2. If the LOCATION is a POINTER, or a 16- or 32-bit OFFSET, or a LOBYTE, then the offset fixup value (FOVAL) is added to the existing contents of the OFFSET component of the LOCATION ignoring any overflow.
3. If the LOCATION is a HIBYTE, then (FOVAL/256) is added to the LOCATION, ignoring overflow.

7.13 Record Order

An object code file must contain a sequence of (one or more) modules, or a library containing zero or more modules. A module is defined as a collection of object code defined by a sequence of object records. The following syntax shows the valid orderings of records to form a module. In addition, the given semantic rules provide information about how to interpret the record sequence.

Note

The syntactic description language used below is defined in WIRTH: CACM, November 1977, vol.#20, no.#11, pp.#822-823. The character strings represented by capital letters above are not literals, but are identifiers that are further defined in the section describing the record formats.

C User's Guide

```
object file    = tmodule
tmodule       = THEADR seg-grp {component} modtail
seg_grp      = {LNAMES} {SEGDEF} {TYPDEF | EXTDEF | GRPDEF}
component    = data | debug_record
data         = content_def | thread_def | TYPDEF | PUBDEF | EXTDEF
debug_record = LINNUM
content_def  = data_record {FIXUPP}
thread_def   = FIXUPP (containing only thread fields)
data_record  = LIDATA | LEDATA
modtail      = MODEND
```

The following rules apply:

- A FIXUPP record always refers to the previous DATA record.
- All LNAMES, SEGDEF, GRPDEF, TYPDEF, and EXTDEF records must precede all records that refer to them.
- COMMENT records may appear anywhere in a file, except as the first or last record in a file or module, or within a content_def.

7.14 Introduction to the Record Formats

The following pages present diagrams of record formats in schematic form. Here is a sample record format, to illustrate the various conventions.

SAMPLE RECORD FORMAT (SAMREC)

REC TYP xxH	RECORD LENGTH	NAME	NUMBER	CHK SUM
← repeated →				

7.14.1 Title and Official Abbreviation

At the top is the name of the record format described, with an official abbreviation. To promote uniformity among various programs, including translators and debuggers, the abbreviation should be used in both code and documentation. The record format abbreviation is always six letters.

7.14.2 The Boxes

Each format is drawn with boxes of two sizes. The narrow boxes represent single bytes. The wide boxes represent two bytes each. The wide dashed boxes represent a variable number of bytes, one or more, depending upon content. The wide solid boxes represent 4-byte fields.

7.14.3 Rectyp

The first byte in each record contains a value between 0 and 255, indicating the record type. For records that have both 16- and 32-bit versions, the low-order bit of the record type indicates the type: 0=16-bit, 1=32 bit.

7.14.4 Record Length

The second field in each record contains the number of bytes in the record, exclusive of the first two fields.

7.14.5 Name

Any field that indicates a “NAME” has the following internal structure: the first byte contains a number between 0 and 127, inclusive, that indicates the number of remaining bytes in the field. The remaining bytes are interpreted as a byte string.

Most translators constrain the character set to be a subset of the ASCII character set.

7.14.6 Number

A 4-byte NUMBER field represents a 32-bit unsigned integer, where the first 8 bits (least-significant) are stored in the first byte (lowest address), the next 8 bits are stored in the second byte, and so on.

7.14.7 Repeated or Conditional Fields

Some portions of a record format contain a field or a series of fields that may be repeated one or more times. Such portions are indicated by the “repeated” or “rpt” brackets below the boxes.

Similarly, some portions of a record format are present only if some given condition is true; these fields are indicated by similar “conditional” or

C User's Guide

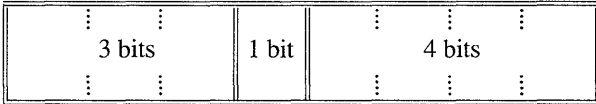
“cond” brackets below the boxes.

7.14.8 Chksum

The last field in each record is a check sum, which contains the 2's complement of the sum (modulo 256) of all other bytes in the record. Therefore, the sum (modulo 256) of all bytes in the record equals 0.

7.14.9 Bit Fields

Descriptions of contents of fields will sometimes be at the bit level. Boxes with complete vertical lines drawn through them represent bytes or words; the partial vertical lines indicate bit boundaries; thus the byte represented below, has three bit-fields of 3-, 1-, and 4-bits.



T-MODULE HEADER RECORD (THEADR)

REC TYP 80H	RECORD LENGTH	T-MODULE NAME	CHK SUM
-------------------	------------------	------------------	------------

Every module output from a translator must have a T-MODULE HEADER RECORD.

7.14.10 T-Module Name

The T-MODULE NAME provides a name for the T-MODULE.

LIST OF NAMES RECORD
(LNAMES)

REC TYP 96H	RECORD LENGTH	NAME	CHK SUM
-------------------	------------------	------	------------

| ← repeated ⇒ |

This Record provides a list of names that may be used in following SEGDEF and GRPDEF records as the names of Segments, Classes and/or Groups.

The ordering of LNAMES records within a module, together with the ordering of names within each LNAMES Record, induces an ordering on the names. Thus, these names are considered to be numbered: 1, 2, 3, 4, ... These numbers are used as “Name Indices” in the Segment Name Index, Class Name Index and Group Name Index fields of the SEGDEF and GRPDEF Records.

7.14.11 Name

This repeatable field provides a name, which may have zero length.

SEGMENT DEFINITION RECORD
(SEGDEF)

REC TYP 98H 99H	RECORD LENGTH	SEGMENT ATTR	SEGMENT LENGTH	SEGMENT NAME INDEX	CLASS NAME INDEX	OVER- LAY NAME INDEX	CHK SUM
--------------------------	------------------	-----------------	-------------------	--------------------------	------------------------	-------------------------------	------------

SEGMENT INDEX values 1 through 32767, which are used in other record types to refer to specific LSEGS, are defined implicitly by the sequence in which SEGDEF Records appear in the object file.

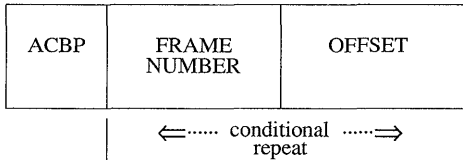


C User's Guide

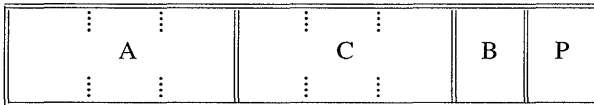
In the RECORD TYPE field, 98H and 99H describe 16- and 32-bit segments, respectively.

7.14.12 Seg Attr

The SEG ATTR field provides information on various attributes of a segment, and has the following format:



The ACBP byte contains four numbers which are the A, C, B, and P attribute specifications. This byte has the following format:



"A" (Alignment) is a 3-bit subfield that specifies the alignment attribute of the LSEG. The semantics are defined as follows:

- A=0 SEGDEF describes an absolute LSEG.
- A=1 SEGDEF describes a relocatable, byte-aligned LSEG.
- A=2 SEGDEF describes a relocatable, word-aligned LSEG.
- A=3 SEGDEF describes a relocatable, paragraph-aligned LSEG.
- A=4 SEGDEF describes a relocatable, page-aligned LSEG.
- A=5 SEGDEF describes a relocatable, double-word-aligned LSEG.
(386 OMF only)

If A=0, the FRAME NUMBER and OFFSET fields will be present. Using **ld**, absolute segments may be used for addressing purposes only; for example, defining the starting address of a ROM and defining symbolic names for addresses within the ROM. **ld** will ignore any data specified as belonging to an absolute LSEG.

“C” (Combination) is a 3-bit subfield that specifies the combination attribute of the LSEG. Absolute segments (A=0) must have combination zero (C=0). For relocatable segments, the C field encodes a number (0,1,2,4,5,6 or 7) that indicates how the segment can be combined. The interpretation of this attribute is best given by considering how two LSEGs are combined:

- Let X,Y be LSEGs, and let Z be the LSEG resulting from the combination of X,Y.
- Let LX and LY be the lengths of X and Y, and let MXY denote the maximum of LX, LY.
- Let G be the length of any gap required between the X- and Y-components of Z to accommodate the alignment attribute of Y.
- Let LZ denote the length of the (combined) LSEG Z; let dx ($0 \leq dx < LX$) be the offset in X the (combined) LSEG Z; let dy ($0 \leq dy < LY$) be the offset in Y of a byte, and let dx' ($0 \leq dx' < LZ$) be the offset in X of a byte, and let dy' ($0 \leq dy' < LZ$) be the offset in Y of a byte.

The following table gives the length LZ of the combined LSEG Z, and the offsets dx' and dy' in Z for the bytes corresponding to dx in X and dy in Y. Intel defines additionally alignment types 5 and 6 and also processes code and data placed in segment with align-type.

Combination Attribute Example

C	LZ	dx'	dy'	
2	LX+LY+G	dx	$dy+LX+G$	Public
5	LX+LY+G	dx	$dy+LX+G$	Stack
6	MXY	dx	dy	Common



The table has no lines for C=0, C=1, C=3, C=4 and C=7. C=0 indicates that the relocatable LSEG may not be combined; C=1 and C=3 are undefined. C=4 and C=7 are treated like C=2. C1, C4, and C7 all have different meanings according to the Intel standard.

“B” (Big) is a 1-bit subfield which, if 1, indicates that the Segment Length is exactly 2^{16} (2^{32} in the case of 32-bit segments). In this case the SEGMENT LENGTH field must contain zero.

The “P” field must always be zero. The “P” field is the “Page resident” field according to the Intel standard.

C User's Guide

The FRAME NUMBER and OFFSET fields (present only for absolute segments, A=0) specify the placement in MAS of the absolute segment. The range of OFFSET is constrained to be between 0 and 15 inclusive. If a value larger than 15 is desired for OFFSET, then an adjustment of the FRAME NUMBER should be done.

7.14.13 Segment Length

The SEGMENT LENGTH field gives the length of the segment in bytes. The length may be zero; if so, **ld** will *not* delete the segment from the module. The SEGMENT LENGTH field is two bytes for a 16-bit segment (Rectyp 98) and four bytes for a 32-bit segment (Rectyp 99). This is large enough for numbers up to $(2^{16})-1$ and $(2^{32})-1$, respectively. The B attribute bit in the ACBP field (see SEG ATTR section) must be used to indicate a length of (2^{16}) or (2^{32}) .

7.14.14 Segment Name Index

The Segment Name is a name the programmer or translator assigns to the segment. Examples: CODE, DATA, STACK, TAXDATA, MODULENAME_CODE. This field provides the Segment Name, by indexing into the list of names provided by the L NAMES Record(s).

7.14.15 Class Name Index

The Class Name is a name the programmer or translator can assign to a segment. If none is assigned, the name is null, and has length 0. The purpose of Class Names is to allow the programmer to define a "handle" used in the ordering of the LSEGS in MAS. Examples: RED, WHITE, BLUE; ROM FASTRAM, DISPLAYRAM. This field provides the Class Name, by indexing into the list of names provided by the L NAMES Record(s).

7.14.16 Overlay Name Index

Note

This is ignored in **ld** versions 2.40 and later, but supported in all earlier versions. However, semantics differ from Intel semantics.

The Overlay Name is a name the translator and/or **ld**, at the programmer's request, applies to a segment. The Overlay Name, like the Class Name, may be null. This field provides the Overlay Name, by indexing into the list of names provided by the L NAMES Record(s).

Note

The "Complete Name" of a segment is a 3-component entity comprising a Segment Name, a Class Name and an Overlay Name. (The latter two components may be null.)

GROUP DEFINITION RECORD

(GRPDEF)

REC TYP 9AH	RECORD LENGTH	GROUP NAME INDEX	GROUP COMPONENT DESCRIPTOR	CHK SUM
-------------------	------------------	------------------------	----------------------------------	------------

| ← repeated ⇒ |



7.14.17 Group Name Index

The Group Name is a name by which a collection of LSEGs may be referenced. The important property of such a group is that, when the LSEGs are eventually fixed in MAS, there must exist some FRAME which "covers" every LSEG of the group.

The GROUP NAME INDEX field provides the Group Name, by indexing into the list of names provided by the L NAMES Record(s).

C User's Guide

7.14.18 Group Component Descriptor

Each GROUP COMPONENT DESCRIPTOR has the following format:

SI (FFH)	SEGMENT INDEX
-------------	------------------

The first byte of the DESCRIPTOR contains 0FFH; the DESCRIPTOR contains one field, which is a SEGMENT INDEX that selects the LSEG described by a preceding SEGDEF record.

Intel defines 4 other group descriptor types, each with its own meaning. They are 0FEH, 0FDH, 0fBH, and 0fAH. The link editor will treat all of these values the same as 0FFH (i.e., it always expects 0FFH followed by a segment index, and it does not check to see if the value is actually 0FF).

TYPE DEFINITION RECORD

(TYPDEF)

REC TYP 8EH	RECORD LENGTH	NAME (usually NULL)	EIGHT LEAF DESCRIPTOR	CHK SUM
-------------------	------------------	------------------------	--------------------------	------------

| ← repeated ⇒ |

The link editor uses TYPDEF records only for communal variable allocation. This is *not* Intel's intended purpose. See "Type Representations for Communal Variables."

As many "EIGHT LEAF DESCRIPTOR" fields as necessary are used to describe a branch. (Every such field except the last in the record describes eight leaves; the last such field describes from one to eight leaves.)

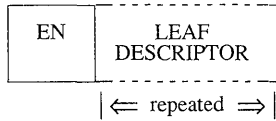
TYPE INDEX values 1 through 32767, which are contained in other record types to associate object types with object names, are defined implicitly by the sequence in which TYPDEF records appear in the object file.

7.14.19 Name

Use of this field is reserved. Translators should place a single byte containing 0 in it (the representation of a name of length zero).

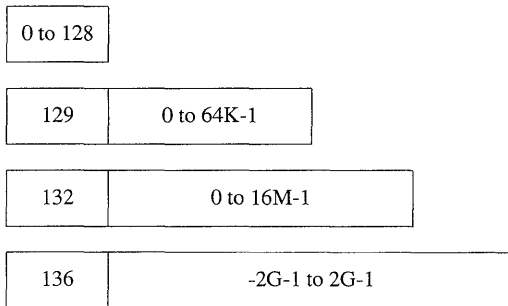
7.14.20 Eight-Leaf Descriptor

This field can describe up to eight Leaves.



The EN field is a byte: the 8 bits, left to right, indicate if the following 8 Leaves (left to right) are Easy (bit=0) or Nice (bit=1).

The LEAF DESCRIPTOR field, which occurs between 1 and 8 times, has one of the following formats:



The first format (single byte), containing a value between 0 and 127, represents a Numeric Leaf whose value is the number given.

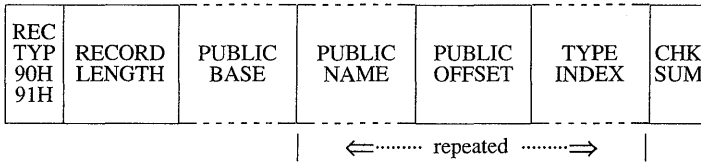
The second format, with a leading byte containing 129, represents a Numeric Leaf. The number is contained in the following two bytes.

The third format, with a leading byte containing 132, represents a Numeric Leaf. The number is contained in the following three bytes.

C User's Guide

The fourth format, with a leading byte containing 136, represents a Signed Numeric Leaf. The number is contained in the following four bytes, sign extended if necessary.

PUBLIC NAMES DEFINITION RECORD (PUBDEF)



This record provides a list of one or more PUBLIC NAMES; for each one, three data are provided: (1) a base value for the name, (2) the offset value of the name, and (3) the type of entity represented by the name.

In the RECORD TYPE field, 90H and 91H describe 16- and 32-bit public definition records, respectively.

7.14.21 Public Base

The PUBLIC BASE has the following format:



The GROUP INDEX field has a format given earlier, and provides a number between 0 and 32767 inclusive. A non-zero GROUP INDEX associates a group with the public symbol, and is used as described in “Conceptual Framework for Fixups,” case (F2c). A zero GROUP INDEX indicates that there is no associated group.

The SEGMENT INDEX field has a format given earlier, and provides a number between 0 and 32767, inclusive.

A non-zero SEGMENT INDEX selects an LSEG. In this case, the location of each public symbol defined in the record is taken as a non-negative displacement (given by a PUBLIC OFFSET field) from the first

Object and Executable File Formats

byte of the selected LSEG, and the FRAME NUMBER field must be absent.

A SEGMENT INDEX of 0 (legal only if GROUP INDEX is also 0) means that the location of each public symbol defined in the record is taken as a displacement from the base of the FRAME defined by the value in the FRAME NUMBER field.

The FRAME NUMBER is present if both the SEGMENT INDEX and GROUP INDEX are zero.

A non-zero GROUP INDEX selects some group; this group is taken as the “frame of reference” for references to all public symbols defined in this record; that is, **ld** will perform the following:

1. Any fixup of the form:

TARGET: EI(P)
FRAME: TARGET

(where “P” is a public symbol in this PUBDEF record) will be converted by **ld** to a fixup of the form:

TARGET: SI(L),d
FRAME: GI(G)

where “SI(L)” and “d” are provided by the SEGMENT INDEX and PUBLIC OFFSET fields. (The “normal” action would have the frame specifier in the new fixup be the same as in the old fixup: FRAME: TARGET.)

2. When the value of a public symbol, as defined by the SEGMENT INDEX, PUBLIC OFFSET, and (optional) FRAME NUMBER fields, is converted to a {base,offset} pair, the base part will be taken as the base of the indicated group. If a non-negative 16-bit offset cannot then complete the definition of the public symbol’s value, an error occurs.

A GROUP INDEX of zero selects no group. **ld** will not alter the FRAME specification of fixups referencing the symbol, and will take, as the base part of the absolute value of the public symbol, the canonic frame of the segment (either LSEG or PSEG) determined by the SEGMENT INDEX field.

7

C User's Guide

7.14.22 Public Name

The PUBLIC NAME field gives the name of the object whose location in MAS is made available to other modules. The name must contain one or more characters.

7.14.23 Public Offset

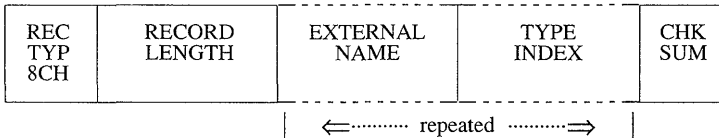
The PUBLIC OFFSET field is a 16-bit value (Rectyp=90H), or a 32-bit value (Rectyp=91H), which is either the offset of the Public Symbol with respect to an LSEG (if SEGMENT INDEX > 0), or the offset of the Public Symbol with respect to the specified FRAME (if SEGMENT INDEX = 0).

7.14.24 Type Index

The TYPE INDEX field identifies a single preceding TYPDEF (Type Definition) Record containing a descriptor for the type of entity represented by the Public Symbol. This field is ignored by **ld**.

EXTERNAL NAMES DEFINITION RECORD

(EXTDEF)



This record provides a list of external names, and for each name, the type of object it represents. **ld** will assign to each External Name the value provided by an identical Public Name (if such a name is found).

7.14.25 External Name

This field provides the name, which must have non-zero length, of an external object.

Inclusion of a Name in an External Names Record is an implicit request that the object file be linked to a module containing the same name declared as a Public Symbol. This request obtains whether or not the External Name is referenced within some FIXUPP Record in the module.

The ordering of EXTDEF Records within a module, together with the ordering of External Names within each EXTDEF Record, induces an ordering on the set of all External Names requested by the module. Thus, External Names are considered to be numbered 1, 2, 3, 4, These numbers are used as ‘‘External Indices’’ in the TARGET DATUM and/or FRAME DATUM fields of FIXUPP Records to refer to a particular External Name.

Note

8086 External Names are numbered positively: 1,2,3,... This is a change from 8080 External Names, which were numbered starting from zero: 0,1,2,... This conforms with other 8086 Indices (Segment Index, Type Index, etc.) which use 0 as a default value with special meaning.

External indices may not reference forward. For example, an external definition record defining the *k*th object must precede any record referring to that object with index *k*.

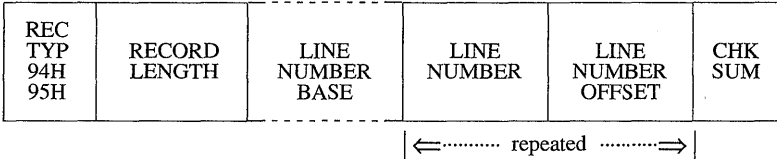
7.14.26 Type Index

This field identifies a single preceding TYPDEF (Type Definition) record containing a descriptor for the type of object named by the External Symbol.

The TYPE INDEX is used only in communal variable allocation by the link editor.



LINE NUMBERS RECORD
(LINNUM)

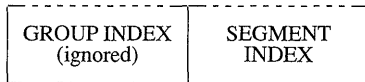


This record provides the means by which a translator may pass the correspondence between a line number in source code and the corresponding translated code.

In the RECORD TYPE field, 94H and 95H describe 16- and 32-bit line number records, respectively.

7.14.27 Line-Number Base

The LINE-NUMBER BASE has the following format:



The SEGMENT INDEX determines the location of the first byte of code corresponding to some source line number.

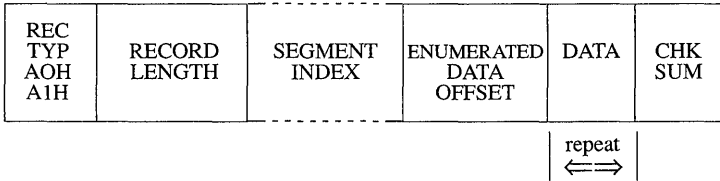
7.14.28 Line-Number

A line number between 0 and 32767, inclusive, is provided in binary by this field. The high-order bit is reserved for future use and must be zero.

7.14.29 Line Number Offset

The LINE-NUMBER OFFSET field is either a 16-bit value (Rectyp=94H) or a 32-bit value (Rectyp=95H), which is the offset of the line number with respect to an LSEG (if SEGMENT INDEX > 0).

LOGICAL ENUMERATED DATA RECORD
(LEDATA)



This record provides contiguous data from which a portion of an 8086 memory image may be constructed.

In the RECORD TYPE field, A0H and A1H describe 16- and 32-bit LEDATA records, respectively.

7.14.30 Segment Index

This field must be non-zero and specifies an index relative to the SEGMENT DEFINITION RECORDS found previous to the LEDATA RECORD.

7.14.31 Enumerated Data Offset

This field specifies either a 16-bit offset (Rectype=A0H) or a 32-bit offset (Rectyp=A1H), that is relative to the base of the LSEG specified by the SEGMENT INDEX and defines the relative location of the first byte of the DAT field. Successive data bytes in the DAT field occupy successively higher locations of memory.

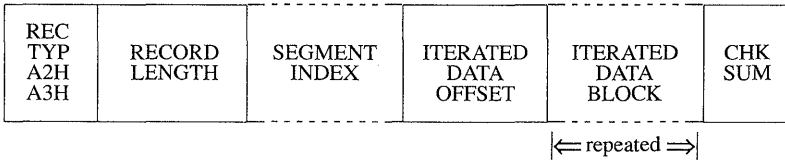


C User's Guide

7.14.32 Dat

This field provides up to 1024 consecutive bytes of relocatable or absolute data.

LOGICAL ITERATED DATA RECORD
(LIDATA)



This record provides contiguous data from which a portion of an 8086 memory image may be constructed.

In the RECORD TYPE field, A2H and A3H describe 16- and 32-bit LIDATA records, respectively.

7.14.33 Segment Index

This field must be non-zero and specifies an index relative to the SEG-DEF records found previous to the LIDATA RECORD.

7.14.34 Iterated Data Offset

This field specifies either a 16-bit offset (Rectype=A2H) or a 32-bit offset (Rectype=A3H), that is relative to the base of the LSEG specified by the SEGMENT INDEX and defines the relative location of the first byte in the ITERATED DATA BLOCK. Successive data bytes in the ITERATED DATA BLOCK occupy successively higher locations of memory.

7.14.35 Iterated Data Block

This repeated field is a structure specifying the repeated data bytes. The structure has the following format:

REPEAT COUNT	BLOCK COUNT	CONTENT
-----------------	----------------	---------

Note

The link editor cannot handle LIDATA records whose ITERATED DATA BLOCK is larger than 512 bytes.

7.14.36 Repeat Count

This field specifies the number of times that the CONTENT portion of this ITERATED DATA BLOCK is to be repeated. REPEAT COUNT must be non-zero.

7.14.37 Block Count

This field specifies the number of ITERATED DATA BLOCKS that are to be found in the CONTENT portion of this ITERATED DATA BLOCK. If this field has value zero, then the CONTENT portion of this ITERATED DATA BLOCK is interpreted as data bytes. If non-zero, then the CONTENT portion is interpreted as that number of ITERATED DATA BLOCKS.

7.14.38 Content

This field may be interpreted in one of two ways, depending on the value of the previous BLOCK COUNT field.

If BLOCK COUNT is zero, then this field is a 1-byte count followed by the indicated number of data bytes.

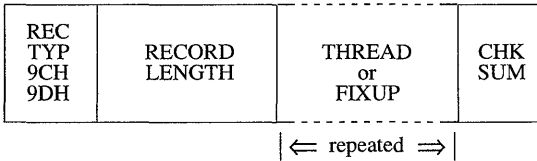
C User's Guide

If BLOCK COUNT is non-zero, then this field is interpreted as the first byte of another ITERATED DATA BLOCK.

Note

From the outermost level, the number of nested ITERATED DATA BLOCKS is limited to 17, i.e., the number of levels of recursion is limited to 17.

FIXUP RECORD (FIXUPP)



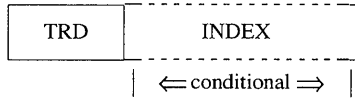
This record specifies 0 or more fixups. Each fixup requests a modification (fixup) to a LOCATION within the previous DATA record. A data record may be followed by more than one fixup record that refers. Each fixup is specified by a FIXUP field that specifies four data: a location, a mode, a target and a frame. The frame and the target may be specified totally within the FIXUP field, or may be specified by reference to a preceding THREAD field.

A THREAD field specifies a default target or frame that may subsequently be referred to in identifying a target or a frame. Eight threads are provided; four for frame specification and four for target specification. Once a target or frame has been specified by a THREAD, it may be referred to by following FIXUP fields (in the same or following FIXUPP records), until another THREAD field with the same type (TARGET or FRAME) and Thread Number (0 - 3) appears (in the same or another FIXUPP record).

In the RECORD TYPE field, 9CH and 9DH describe 16- and 32-bit FIXUPP records, respectively.

7.14.39 Thread

THREAD is a field with the following format.



The TRD DAT (ThReaD DATA) subfield is a byte with this internal structure:



The “Z” is a 1-bit subfield, currently without any defined function, that is required to contain 0.

The “D” subfield is one bit that identifies what type of thread is being specified. If D=0, then a target thread is being defined; if D=1, then a frame thread is being defined.

METHOD is a 3-bit subfield containing a number between 0 and 3 (D=0) or a number between 0 and 6 (D=1).

If D=0, then METHOD = (0, 1, 2, 3, 4, 5, 6, 7) mod 4, where the 0, ..., 7 indicate methods T0, ..., T7 of specifying a target. Thus, METHOD indicates what kind of Index or Frame Number is required to specify the target, without indicating if the target will be specified in a primary or secondary way. Note that methods 2b, 3, and 7 are not supported by **ld**.



If D=1, then METHOD = 0, 1, 2, 4, 5, corresponding to methods F0, ..., of specifying a frame. Here, METHOD indicates what kind (if any) of Index is required to specify the frame. Note that methods 3 and 5d are not supported by **ld**.

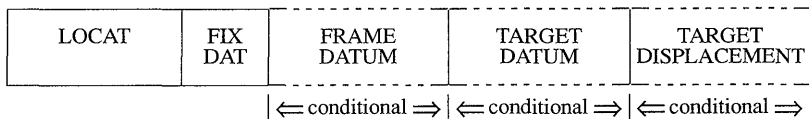
THRED is a number between 0 and 3, and associates a Thread Number to the frame or target defined by the THREAD field.

INDEX contains a Segment Index, Group Index, or External Index depending on the specification in the METHOD subfield. This subfield will not be present if F4 or F5 are specified by METHOD.

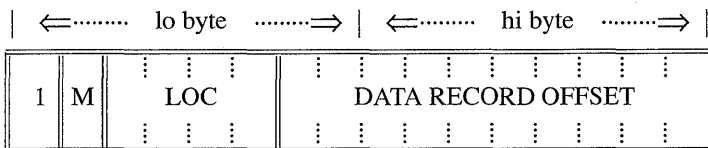
C User's Guide

7.14.40 Fixup

FIXUP is a field with the following format:



LOCAT is a byte pair with the following format:



M is a 1-bit subfield that specifies the mode of the fixups: self-relative (M=0) or segment-relative (M=1).

Note

Self-Relative fixups may not be applied to LIDATA records.

LOC is a four-bit sub-field indicating the type of location that is to be fixed up:

- 0 - 8 bit lobyte
- 1 - 16 bit offset
- 2 - 16 bit base
- 3 - 32 bit pointer
- 4 - 8 bit hi byte
- 5 - 16 bit offset (linker resolved)
- 9 - 32 bit offset
- 11 - 48 bit pointer
- 13 - 32 bit offset (linker resolved)

Object and Executable File Formats

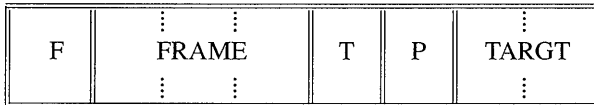
LOC values 9, 11 and 13 are only valid in 32-bit FIXUPP records (record type 9D). All other values of LOC are invalid.

The DATA RECORD OFFSET is a number between 0 and 1023, inclusive, that gives the relative position of the lowest order byte of LOCATION (the actual bytes being fixed up) within the preceding DATA record. The DATA RECORD OFFSET is relative to the first byte in the data fields in the DATA RECORDS.

Note

It is possible for the value of DATA RECORD OFFSET to designate a ‘location’ within a REPEAT COUNT subfield or a BLOCK COUNT subfield of the ITERATED DATA field. Such a reference is an error. The action of **ld** on such a malformed record is undefined.

FIX DAT is a byte with the following format:



Note

Frame method 2b, F3, and F5d are not supported. Target method T3 and T7 are not supported.

F is a 1-bit subfield that specifies whether the frame for this FIXUP is specified by a thread (F=1) or explicitly (F=0).

FRAME is a number interpreted in one of two ways as indicated by the F bit. If F is zero, FRAME is a number between 0 and 5 and corresponds to methods F0, ..., F5 of specifying a FRAME. If F=1, then FRAME is a thread number (0-3). It specifies the frame most recently defined by a THREAD field that defined a frame thread with the same thread number. (Note that the THREAD field may appear in the same, or in an earlier FIXUPP record.)

C User's Guide

“T” is a 1-bit subfield that specifies whether the target specified for this fixup is defined by reference to a thread (T=1), or is given explicitly in the FIXUP field (T=0).

“P” is a 1-bit subfield that indicates whether the target is specified in a primary way (requires a TARGET DISPLACEMENT, P=0) or specified in a secondary way (requires no TARGET DISPLACEMENT, P=1). Since a target thread does not have a primary/secondary attribute, the P bit is the only field that specifies the primary/secondary attribute of the target specification.

TARGET is interpreted as a 2-bit subfield. When T=0, it provides a number between 0 and 3, corresponding to methods T0, ..., T3 or T4, ..., T7, depending on the value of P (P can be interpreted as the high-order bit of T0, ..., T7). When the target is specified by a thread (T=1), then TARGET specifies a thread number (0-3).

FRAME DATUM is the “referent” portion of a frame specification, and is a Segment Index, a Group Index, an External Index. The FRAME DATUM subfield is present only when the frame is specified neither by a thread (F=0) nor explicitly by methods F4 or F5 or F6.

TARGET DATUM is the “referent” portion of a target specification, and is a Segment Index, a Group Index, an External Index or a Frame Number. The TARGET DATUM subfield is present only when the target is not specified by a thread (T=0).

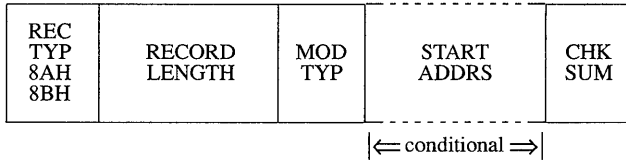
TARGET DISPLACEMENT is the displacement required by “primary” methods of specifying TARGETS. This field is 2 bytes long in 16-bit FIXUPP records (Rectyp=9CH) and 4 bytes long in 32-bit FIXUPP records (Rectyp=9DH). This subfield is present if P=0.

Note

All these methods are described in “Conceptual Framework for Fixups.”

MODULE END RECORD

(MODEND)

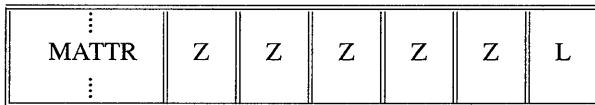


This record serves two purposes. It denotes the end of a module and indicates whether the module just terminated has a specified entry point for initiation of execution. If the latter is true, the execution address is specified.

In the RECORD TYPE field, 8AH and 8BH describe 16- and 32-bit MODEND records, respectively.

7.14.41 Mod Type

This field specifies the attributes of the module. The bit allocation and associated meanings are as follows:



MATTR is a 2-bit subfield that specifies the following module attributes:

<u>MATTR</u>	<u>MODULE ATTRIBUTE</u>
0	Non-main module with no START ADDRS
1	Non-main module with START ADDRS
2	Main module with no START ADDRS
3	Main module with START ADDRS

“L” indicates whether the START ADDRS field is interpreted as a logical address that requires fixing up by **ld**. (L=1). Note that with **ld**, L must always equal 1.

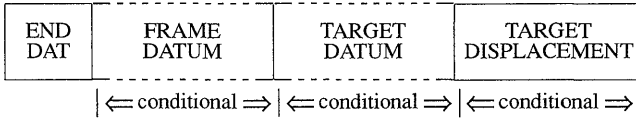
C User's Guide

“Z” indicates that this bit has not currently been assigned a function. These bits are required to be zero.

Physical start addresses (L=0) are not supported.

The START ADDRS field (present only if MATTR is 1 or 3) has the following format:

START ADDRS



The starting address of a module has all the attributes of any other logical reference found in a module. The mapping of a logical starting address to a physical starting address is done in exactly the same manner as mapping any other logical address to a physical address as specified in the discussion of fixups and the FIXUPP record. The above subfields of the START ADDRS field have the same semantics as the FIX DAT, FRAME DATUM, TARGET DATUM, and TARGET DISPLACEMENT fields in the FIXUPP record. Only “primary” fixups are allowed. Frame method F4 is not allowed.

The TARGET DISPLACEMENT field is 2 bytes in a 16-bit MODEND record (Rectyp=8AH) and 4 bytes in a 32-bit MODEND record (Rectyp=8BH).

COMMENT RECORD

(COMENT)



This record allows translators to include comments in object text.

7.14.42 Comment Type

This field indicates the type of comment carried by this record. This allows comments to be structured for those processes that wish to selectively act on comments. The format of this field is as follows:

NP	NL	Z	Z	Z	Z	Z	Z	COMMENT CLASS
----	----	---	---	---	---	---	---	------------------

The NP (NOPURGE) bit, if 1, indicates that it is not able to be purged by object file utility programs which implement the capability of deleting COMMENT record.

The NL (NOLIST) bit, if 1, indicates that the text in the COMMENT field is not to be listed in the listing file of object file utility programs which implement the capability of listing object COMMENT records.

The COMMENT CLASS field is defined as follows:

- 0 Language translator comment.
- 1 Intel copyright comment. The NP bit must be set.
- 2-155 Reserved for Intel use. (See note 1 below.)
- 156-255 Reserved for users. Intel products will apply no semantics to these values. (See Note 2 below.)

NOTES:

1. Class value 159 is used to specify a library to add to the link editor's library search list. The comment field will contain the name of the library. Note that unlike all other name specifications, the library name is not prefixed with its length. Its length is determined by the record length.
2. Class value 156 is used to specify a DOS level number. When the class value is 156, the comment field will contain a two-byte integer specifying a DOS level number.
3. Class value 161 is used to indicate that the module contains XENIX extensions to OMF, such as the various 32-bit record types.

C User's Guide

7.14.43 Comment

This field provides the commentary information.

7.15 Numeric List of Record Types

*6E	RHEADR	*92	LOCSYM
*70	REGINT	*93	MLOC386
*72	REDATA	94	LINNUM
*74	RIDATA	95	MLIN386
*76	OVLDEF	96	LNAMES
*78	ENDREC	98	SEGDEF
*7A	BLKDEF	99	MSEG386
*7C	BLKEND	9A	GRPDEF
*7E	DEBSYM	9C	FIXUPP
80	THEADR	9D	MFIX386
*82	LHEADR	*9E	(none)
*84	PEDATA	A0	LEDATA
*86	PIDATA	A1	MLED386
88	COMENT	A2	LIDATA
8A	MODEND	A3	MLID386
8B	H386END	*A4	LIBHED
8C	EXTDEF	*A6	LIBNAM
8E	TYPDEF	*A8	LIBLOC
90	PUBDEF	*AA	LIBDIC
91	MPUB386		

Note

The record types marked with an asterisk are not supported by the link editor. They will be ignored if they are found in an object module.

7.16 Type Representations for Communal Variables

This section defines the XENIX standard for communal variable allocation on the 8086 and 80286.

A communal variable is an uninitialized public variable whose final size and location are not fixed at compile time. Communal variables are similar to FORTRAN common blocks in that if a communal variable is declared in more than one object module being linked together, then its actual size will be the largest size specified in the several declarations. In the C language, all uninitialized public variables are communal. The following example shows three different declarations of the same C communal variable:

```
char foo[4];           /* In file a.ce */
char foo[1];          /* In file b.ce */
char foo[1024];       /* In file c.ce */
```

If the objects produced from a.ce, b.c, and c.c are linked together, then the linker will allocate 1024 bytes for the char array “foo.”

A communal variable is defined in the object text by an external definition record (EXTDEF) and the type definition record (TYPDEF) to which it refers.

The TYPDEF for a communal variable has the following format:

REC TYP 8EH	RECORD LENGTH	0	EIGHT LEAF DESCRIPTOR	CHK SUM
-------------------	------------------	---	--------------------------	------------

C User's Guide

The EIGHT LEAF DESCRIPTOR field has the following format:

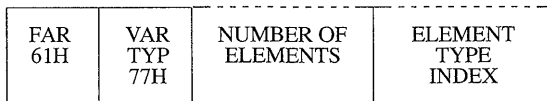


The EN field specifies whether the next 8 leaves in the LEAF DESCRIPTOR field are EASY (bit = 0) or NICE (bit = 1). This byte is always zero for TYPDEFS for communal variables.

The LEAF DESCRIPTOR field has one of the following two formats. The format for communal variables in the default data segment (near variables) is as follows:



The VARIABLE TYPE field may be either SCALAR (7BH), STRUCT (79H), or ARRAY (77H). The VAR SUBTYP field (if any) is ignored by **ld**. The format for communal variables not in the default data segment (far variables) is as follows:



The VARIABLE TYPE field must be ARRAY (77H). The length field specifies the NUMBER OF ELEMENTS, and the ELEMENT TYPE INDEX is an index to a previously defined TYPDEF whose format is that of a near communal variable.

The format for the LENGTH IN BITS or NUMBER OF ELEMENTS fields is the same as the format for the LEAF DESCRIPTOR field, described in the TYPDEF record format section of this guide.

Link Time Semantics

All EXTDEFs referencing a TYPDEF of the previously described formats are treated as communal variables. All others are treated as externally defined symbols for which a matching public symbol definition (PUBDEF) is expected. A PUBDEF matching a communal variable definition will override the communal variable definition. Two communal variable definitions are said to match if the names given in the definitions match. If two matching definitions disagree as to whether a communal variable is near or far, the linker will assume the variable is near.

If the variable is near, then its size is the largest specified for it. If the variable is far, then the link editor issues a warning if there are conflicting array element size specifications; if there are no such conflicts, then the variable's size is the element size times the largest number of elements specified. The sum of the sizes of all near variables must not exceed 64K bytes. The sum of the sizes of all far variables must not exceed the size of the machine's addressable memory space.

“Huge” Communal Variables

A far communal variable whose size is larger than 64K bytes will reside in segments that are contiguous (8086) or have consecutive selectors (80286). No other data items will reside in the segments occupied by a huge communal variable.

If the linker finds matching huge and near communal variable definitions, it issues a warning message, since it is impossible for a near variable to be larger than 64K bytes.

7.17 The Segmented x.out Format

This section describes the executable object file format used in XENIX. The format used is an extension to the existing “x.out” format, specifically enhanced for the segmented architecture of the 286 CPU.

The XENIX linker (*/bin/ld*, see the “ld: the Link Editor” chapter) will link the Intel 86 Relocatable Object Format into the executable format described in this section.



C User's Guide

The XENIX product supports a subset of segmented omf. Other parts are specified here for use by other vendors, and to reserve their meaning for possible future use. Those parts supported in this release of XENIX are:

- The x.out header
- The x.out extended header
- The file segment table
- Multiple non-iterated text segments
- Multiple non-iterated data segments
- Symbol table segments in the format described herein.

Note specifically that the machine-dependent table is *not* supported. The iterated text/data feature is supported by the kernel, but the XENIX linker will expand iterated records.

7.17.1 General Description of x.out

The following is a general description of the *x.out* object file format, extended to handle segmentation. It implements iterated text and data segments, huge, large, middle and small model, as well as block alignment to improve the efficiency of loading executable files.

The extensions to the existing format consist of adding a file segment table that describes and points to various (possibly block aligned) file segments. A file segment may contain a memory image, may indicate how to construct a memory image (iterated text or data), or may contain symbols or other non-executable information. In addition to the file segment table, there is an optional machine-dependent table.

The header must be first in the object file, and the extended header must immediately follow the header. The extended header indicates the segment and (optional) machine-dependent tables' sizes and positions. Although the segment table is not block aligned, individual entries will line up on a multiple of 32 bytes (the size of a segment table entry). The segment table indicates the sizes and positions of the remaining file segments. The file segments may be aligned on a boundary that is a multiple of 512 bytes, with that multiple stored in the extended header, or at location zero if the file segments are not block aligned.

Object and Executable File Formats

The segment table is an array of records describing the file segments, each containing:

- A segment type: text, data, symbols, etc.
- Segment attributes, specific to the type of segment.
- A file pointer to the (possibly iterated) text/data for this segment.
- A physical size, the size of the segment in the file.
- A virtual size, the size the segment will occupy in memory.
- A location counter, this segment's current base address, usually 0.

A sample of a segment table entry is shown below. The *xs* fields in this data structure are referred to throughout the remaining discussion in this section.

```


Segment table entry


struct xseg {
    unsigned short xs_type;           /* x.out segment table entry */
    unsigned short xs_attr;          /* segment type */
    unsigned short xs_seg;           /* segment attributes */
    unsigned short xs_sres;          /* segment number */
    long           xs_filpos;         /* unused */
    long           xs_psize;          /* file position */
    long           xs_vsize;          /* physical size (in file) */
    long           xs_rbase;          /* virtual size (in core) */
    long           xs_lres;           /* relocation base address */
    long           xs_lres2;          /* unused */
};
```

The segment table is a contiguous array of the above structures. Each file segment has a corresponding segment table entry that describes the segment's position *xs_filpos* and physical size *xs_psize* in the file. If there is no associated file segment, both fields must be set to zero.

The kernel's local descriptor table (LDT) can be built from the virtual size, the segment type, and segment attribute fields.

7.17.2 Example of File Layout

This section provides an example of the layout of an *x.out* file where:

- The segment table has two entries (segments).
- The file page size is 512 bytes (*xext.xe_pagesize=1*).
- Both file segments are smaller than 512 bytes.
- The second file segment contains iterated data.

The file layout is illustrated below:

Accessing the machine-dependent table and the file segment table must always be done through the absolute file pointers in the extended header. The ordering of the two tables and file segments shown above is not required to be consistent with the *x.out* XENIX specification.

7.17.3 Iterated Segments

The data structure for an iterated segment is shown below:

```
struct xiter {
    long    xi_size;      /* byte count */
    long    xi_rep;      /* replication count */
    long    xi_offset;   /* destination offset in segment */
};
```

If the segment contains iterated text/data (indicated by a bit in the *xs_attr* field), the *xs_filpos* field is the file position of some number of iteration records mixed with the text/data to be iterated. If any part of a segment is iterated, then all of that segment is represented as iterated; non-iterated portions may be represented by an iteration record with a replication count of one.

The format of the text/data to be iterated is:

<iteration record> <text/data> <iteration record> <text/data> ...

where each <iteration record> is of the above “struct xiter” data structure. Each iteration record is followed by *xi_size* bytes of text/data that are to be placed in the current segment at the specified offset *xi_offset* *xi_rep* times. When *xs_psize* bytes of iteration records and text/data have been expanded, the iteration is complete.

Object and Executable File Formats

Under XENIX, areas of memory that are initialized by more than one iteration record will have the contents of those memory areas undefined. Areas of memory that are not initialized by any iteration records will be zeroed out. An iteration byte count *xi_size* of zero will not result in any iteration. Portions of a segment that are to be **bss** should use an iteration record with a non-zero byte count and replicate one or more zeroed data bytes.

This representation of iterated text/data will handle iterations that contain very large replication counts and/or very large non-iterated sizes.

7.17.4 Non-Iterated Segments and Implicit bss

If the iteration bit in *xs_attr* is **not** set, no iterations are required to initialize the segment. If the implicit bss bit in the *xs_attr* field is set and the virtual size is greater than the physical size, then the rest of the segment (up to *xs_vsize* bytes) is filled with zeros by the kernel loader. This implicit bss definition means that small and middle model executables' single data segments may still contain unexpanded bss without the use of explicit iteration records.

Segments made up entirely of implicit "C" bss need only set the physical size to zero, and set the implicit bss bit. This eliminates the need for any file segment containing data or iteration records. If there are no iterations and no implicit bss, the virtual size of the segment *xs_vsize* must be the same as the physical size *xs_psize*, and a single copy of the text/data located at *xs_filpos* is all that is required to initialize the segment.

7.17.5 Large Model

With x.out format, large model is supported by allowing multiple logical text and/or data segments. Middle and small models are simpler cases, with perhaps single logical segments for data (or both text and data). Iterated segments are independent of memory model.

7.17.6 Special Header Fields

The model bits in the *x_renv* field of the main header, XE_LDATA and XE_LTEXT, usually indicate the default size of data and text pointers used in the executable code. The kernel depends on these two bits to indicate the size of data and text pointers passed in system calls. However, since multiple segments are allowed in small and middle model, there can be little other meaning attached to these bits. Passing near data and/or text pointers implies use of the first data and text segments, respectively.



C User's Guide

Also in the *x_renv* field, the absolute bit, XE_ABS, identifies a standalone executable file. When this bit is set, the extended header stack size field is used as the default physical load address. The XENIX kernel loader will not load a binary if the XE_ABS bit is set. The XENIX boot loader will not load a binary unless the XE_ABS bit is set. See the **ld(CP)** command in the *XENIX Reference* for information about how to set the XE_ABS bit and the physical load address.

7.17.7 Symbol Table

The data structure for the *x.out* symbol table is shown below:

```
struct sym {                                /* x.out symbol table entry */
    unsigned short  s_type;
    unsigned short  s_seg;
    long            s_value;
};
```

The symbol table differs from the previous *x.out* only in that the *s_seg* field now holds the selector of the segment that defines the symbol. If the symbol is absolute, the value field holds the symbol's value; otherwise, it holds the offset in the indicated segment to which the symbol refers.

The symbol name trails the above "struct sym" data structure in the form of a null terminated string. The type field values are defined in */usr/include/sys/relsym.h*.

The use of the *xs_seg* field in the segment table is undefined for symbol table segments. Its use may be defined by the particular symbol table format used.

7.17.8 XENIX Executable Format

XENIX does not execute binaries that make use of selectors below 0x3f or selectors that do not have the low 3 bits set (LDT, ring 3). XENIX also requires that the first data selector be after the last text selector. Binaries are allowed to have zero length segments or "holes" (unused selectors) in text or data, but holes in text may not contain data selectors, and holes in data may not contain text selectors.

The fields, *xext.xe_eseg:xexec.x_entry*, must contain the initial **cs:ip** of the user process.

Small model impure binaries (text and data combined into a single segment) must have a single file segment, of type data, with a selector of at

Object and Executable File Formats

least 0x47. It must contain all text, followed by all data, followed by bss. The sizes of each must be stored in the *x_text*, *x_data* and *x_bss* fields of the main header. XENIX will use the value stored in the *xext.xe_eseg* field as the text selector, which must be at least 0x3f and less than the data selector. All text/data/bss binaries are executable through the text selector, and all text/data/bss binaries are readable and writable through the data selector. XENIX maps the text selector to the same memory as the data selector.

In addition to the above, the XENIX linker generates binaries that conform to the following:

- Text selectors start at 0x3f.
- Data selectors start at the first free selector past text.
- All text selectors are contiguous.
- All data selectors are contiguous.
- Small model impure binaries conform to the above specification, with 0x47 as the data selector. In the symbol table, the selector 0x47 is associated with data symbols, and the selector 0x3f is associated with text symbols, to allow **adb** and **nm** to present consistent data to the user.

C User's Guide

7.17.9 Selected Portions of Include Files

The following are selected portions of the *usr/include/sys/a.out.h* and *usr/include/sys/relsym.h* include files.

```
struct xexec {
    /* x.out header */
    unsigned short x_magic;
    /* magic number */
    unsigned short x_ext;
    /* size of header extension */
    long x_text;
    /* size of text segment */
    long x_data;
    /* size of initialized data */
    long x_bss;
    /* size of uninitialized data */
    long x_syms;
    /* size of symbol table */
    long x_reloc;
    /* relocation table length */
    long x_entry;
    /* entry offset, see xe_eseg */
    char x_cpu;
    /* cpu type & byte/word order */
    char x_relsym;
    /* relocation & symbol format */
    unsigned short x_renv;
    /* run-time environment */
};
```

Object and Executable File Formats

```
struct xext {
    /* x.out header extension */
    long    xe_trsize;
           /* size of text relocation */
    long    xe_drsize;
           /* size of data relocation */
    long    xe_drsize;
           /* size of data relocation */
    long    xe_dbase;
           /* data relocation base */
    long    xe_stksize;
           /* stack size (if XE_FS set) */
    long    xe_segpos;
           /* segment table position */
    long    xe_segsize;
           /* segment table size */
    long    xe_mdtpos;
           /* machine dependent table position */
    long    xe_mdtsize;
           /* machine dependent table size */
    char    xe_mdtttype;
           /* machine dependent table type */
    char    xe_pagesize;
           /* file pagesize, in multiples of 512 */
    char    xe_ostype;
           /* operating system type */
    char    xe_osvers;
           /* operating system version */
    unsigned short xe_eseg;
           /* entry segment (hardware dependent) */
    unsigned short xe_sres;
           /* reserved */
};

/*
 *   Definitions for xexec.x_renv (short).
 *
 *   vv    version compiled for
 *   xx    extra (zero)
 *   s     set if segmented x.out
 *   a     set if absolute (set up for physical address)
 *   i     set if segment table contains iterated text/data
 *   h     set if huge model data
 *   f     set if floating point hardware required
 *   t     set if large model text
 *   d     set if large model data
 *   o     set if text overlay
 *   f     set if fixed stack
 *   p     set if text pure
 *   s     set if separate I & D
 *   e     set if executable
 */
```

7

C User's Guide

```
#define XE_V2      0x4000
/* up to and including 2.3 */
#define XE_V3      0x8000
/* after version 2.3 */
#define XE_VERS    0xc000
/* version mask */

#define XE_SEG0x0800
/* segment table present */
#define XE_ABS0x0400
/* absolute memory image (standalone) */
#define XE_ITER    0x0200
/* iterated text/data present */
#define XE_HDATA   0x0100
/* huge model data */
#define XE_FPH0x0080
/* floating point hardware required */
#define XE_LTEXT   0x0040
/* large model text */
#define XE_LDATA   0x0020
/* large model data */
#define XE_OVER    0x0010
/* text overlay */
#define XE_FS      0x0008
/* fixed stack */
#define XE_PURE    0x0004
/* pure text */
#define XE_SEP0x0002
/* separate I & D */
#define XE_EXEC    0x0001
/* executable */

struct xseg {
/* x.out segment table entry */
  unsigned short xs_type;
/* segment type */
  unsigned short xs_attr;
/* segment attributes */
  unsigned short xs_seg;
/* segment number */
  unsigned short xs_sres;
/* unused */
  long xs_filpos;
/* file position */
  long xs_psize;
/* physical size (in file) */
  long xs_vsize;
/* virtual size (in core) */
  long xs_rbase;
/* relocation base address */
  long xs_lres;
/* unused */
  long xs_lres2;
/* unused */
};
```

```

struct xiter {
    /* x.out iteration record */
    long    xi_size;
           /* byte count */
    long    xi_rep;
           /* # of repetitions */
    long    xi_offset;
           /* destination offset in segment */
};

struct sym {
    /* x.out symbol table entry */
    unsigned short s_type;
    unsigned short s_seg;
    long          s_value;
};

/*
 *   Definitions for xe_mdtype
 */
#defineXE_MDTNONE    0
           /* no machine dependent table */
#defineXE_MDT286    1
           /* iAPX286 LDT */

/*
 *   Definitions for xe_ostype
 */
#defineXE_OSNONE    0
#defineXE_OSXENIX  1
           /* XENIX */
#defineXE_OSRMX    2
           /* iRMX */

/*
 *   Definitions for xe_osvers
 */
#defineXE_OSXV3    1
           /* XENIX */

/*
 *   Definitions for xs_type:
 *   Values from 64 to 127 are reserved.
 */
#defineXS_TNULL    0    /* unused segment */
#defineXS_TTEXT    1    /* text segment */
#defineXS_TDATA    2    /* data segment */
#defineXS_TSYMS    3    /* symbol table segment */
#defineXS_TREL4    4    /* relocation segment */

```

C User's Guide

```
/*
 *   Definitions for xs_attr:
 *       The top bit is set if the file segment represents
 *       a memory image. The other 15 bits' definitions
 *       depend on the type of file segment.
 */
#define XS_AMEM      0x8000
/* segment represents a memory image */
#define XS_AMASK     0x7fff
/* type specific field mask */

/*
 *   Definitions for xs_attr, built by or'ing the following
 *   bit patterns: these values are valid for XS_TTEXT and
 *   XS_TDATA file segments only.
 */
#define XS_ALTER     0x0001
/* contains iteration records */
#define XS_AHUGE     0x0002
/* contains huge element */
#define XS_ABSS      0x0004
/* contains implicit bss */
#define XS_APURE     0x0008
/* is read-only, may be shared */
#define XS_AEDOWN    0x0010
/* segment expands downward */

/*
 *   Definitions for xs_attr.
 *   These values are valid for XS_TSYMS file segments only.
 */
#define XS_SXSEG     0x0001
/* x.out segmented format */
```

When using the *xs_seg* field, note that if the XS_AMEM bit is set in the *xs_attr* field, the file segment represents a memory image, and the value placed in this field should be the segment number as used by the hardware to reference the segment. This is the actual value placed in the segment register. For the 286, it is simply an LDT selector (under XENIX, if the privilege level is not 3, the file will not be executed). Otherwise the segment is not a memory image, and the contents of the field is not defined. File segments other than memory images may define and use this field as needed.

There are two bits in the *xexec.x_cpu* field that are used to indicate the CURRENT byte and word ordering of the non-character data fields of the header, extended header, segment table and symbol table. These bits, XC_BSWAP and XC_WSWAP, do not indicate the byte and word ordering of the target cpu, XC_CPU.

The segment table is not block aligned. No individual segment table entry may straddle a block boundary.

Chapter 8

C Language Compatibility with Assembly Language

- 8.1 Introduction 8-1
- 8.2 C Calling Sequence for 8086/80286 8-1
- 8.3 Entering an 8086/80286 Assembly Routine 8-2
- 8.4 8086/80286 Return Values 8-2
- 8.5 Exiting an 8086/80286 Routine 8-2
- 8.6 8086/80286 Program Example 8-3
- 8.7 80386 C Language Calling Sequence 8-4
- 8.8 Entering an 80386 Assembly-Language Routine 8-4
- 8.9 80386 Return Values 8-5
- 8.10 Exiting a 80386 Routine 8-7
- 8.11 80386 Program Example 8-7

8.1 Introduction

This appendix explains how to use 8086/286/386 assembly language routines with C language programs and functions. In particular, it explains how to call assembly language routines from C language programs and how to call C language functions from an assembly language routine.

This assembly language interface is especially useful for those assembly language programmers whose wish to use the functions of the standard C library and other C libraries.

Note

Two different calling conventions are available. The 8086/80286 calling convention is established by configuring C language programs with the **-M0**, **-M1**, or **-M2** option. The 80386 calling convention is established by configuring C language programs with the **-M3** option.

8.2 C Calling Sequence for 8086/80286

To receive values from C language function calls or to pass values to C functions, assembly language routines must follow the C argument passing conventions. C language function calls pass their arguments to the given functions by pushing the value of each argument onto the stack. The call pushes the value of the last argument first and the first argument last. If an argument is an expression, the call computes the expression's value before pushing it onto the stack.

Arguments with **char**, **int**, or **unsigned** type occupy a single word (16 bits) on the stack. Arguments with **long** type occupy a double word (32 bits) with the value's high order word occupying the first word. Arguments with **float** type are converted to **double** type (64 bits). Note that **char** type arguments are zero-extended to **int** type before being pushed on the stack.

If an argument is a structure, the function call pushes the last word of the structure first and each successive word in turn until the first word is pushed.

After a function returns control to a routine, the calling routine is responsible for removing arguments from the stack.



C User's Guide

8.3 Entering an 8086/80286 Assembly Routine

Assembly language routines that receive control from C function calls should preserve the contents of the **bp**, **si**, and **di** registers and set the **bp** register to the current **sp** register value before proceeding with their tasks. The following example illustrates the recommended instruction sequence for entry to an assembly language routine:

```
entry:
    push    bp
    mov     bp, sp
    push    di
    push    si
```

This is the same sequence used by the C compiler.

If this sequence is used, the last argument passed by the function call (which is also the first argument given in the call's argument list) is at address "[bp+4]". Subsequent arguments begin at address "[bp+6]" or "[bp+8]" depending on the size of the first argument.

This sequence is strongly recommended even if the **si** and **di** registers are not modified, since it allows backtracing with the **adb** program during program debugging.

8.4 8086/80286 Return Values

Assembly language routines that wish to return values to a C language program or receive return values from C functions must follow the C return value conventions. C functions place return values that have **int**, **char**, or **unsigned** type in the **ax** register. They place values with **long** type in the **ax** and **dx** registers, with the high order word in **dx**.

To return a structure or a floating point value, C functions place the address of the given value in the **ax** register. The structure or floating point value must be in a static area in memory. Long addresses are returned in the **ax** and **dx** registers with the segment selector in **dx**.

8.5 Exiting an 8086/80286 Routine

Assembly language routines that return control to C programs should restore the values of the **bp**, **si**, and **di** registers before returning control. The following example illustrates the recommended instruction sequence for exiting a routine:

C Language Compatibility with Assembly Language

```
pop  si
pop  di
mov  sp, bp
pop  bp
ret
```

This sequence does not change the **ax**, **bx**, **cx**, or **dx** registers or any of the segment registers. The sequence does not remove arguments from the stack. This is the responsibility of the calling routine.

8.6 8086/80286 Program Example

To illustrate the assembly language interface, consider the following example of a C function:

```
add(i, j)
int i, j;
{
    return(i+j);
}
```

If written as an assembly language routine, this function must save the proper registers, retrieve the arguments from the stack, add the arguments, place the return value in the **ax** register, then restore registers and return control. The following is an example of how the routine can be written:

```
_add:
    push bp
    mov  bp, sp
    push di
    push si

    mov  ax, [bp+4]
    add  ax, [bp+6]

    pop  si
    pop  di
    mov  sp, bp
    pop  bp
    ret
```

If, on the other hand, the C function is to be called by an assembly language routine, the routine must contain instructions that push the argu-



C User's Guide

ments on the stack in the proper order, call the function, and clear the stack. It may then use the return value in the **ax** register. The following is an example of the instructions that can do this:

```
push <j value>
push <i value>
call _add
add sp, *4
```

Note that the C compiler does not preserve **es** over calls. Assembly language routines need not preserve **es** and should not assume that it will be preserved if they make calls to routines written in C.

8.7 80386 C Language Calling Sequence

To receive values from 80386 C language function calls, or to pass values to 80386 C language functions, assembly-language routines must follow the 80386 C language argument-passing conventions.

C language function calls pass arguments to the function by pushing each argument onto the stack. The call pushes the last function argument first and the first function argument last onto the stack. If an argument is an expression, the call computes the expression's value before pushing it onto the stack.

Arguments with **char**, **int**, **unsigned**, **short**, or **long** type occupy a doubleword (32 bits or 4 bytes) on the stack. Arguments with **float** type are converted to **double** type (64 bits or 8 bytes). Note that **char**, **unsigned char**, **short**, and **unsigned short** type arguments are sign extended or zero extended, respectively, to **int** type before being pushed onto the stack.

If an argument is a structure, the function call pushes the last word of the structure first and each successive word in turn until the first word of the structure is pushed onto the stack.

After a function returns control to the calling routine, the calling routine is responsible for removing all function arguments from the stack.

8.8 Entering an 80386 Assembly-Language Routine

Assembly-language routines that receive control from 80386 C function calls should preserve the contents of the **ebp**, **esi**, **edi**, and **ebx** registers. In addition, the routines should set the **ebp** register to the current **esp**

C Language Compatibility with Assembly Language

register value before proceeding with their tasks. The following example illustrates a recommended instruction sequence for entry to an assembly-language routine:

```
entry:
    push    ebp
    mov     ebp, esp
    push    edi
    push    esi
    push    ebx
```

Note that this is the same routine that the compiler uses after pushing the function arguments onto the stack.

If this sequence is used, the last function argument pushed by the function call (which is also the first argument in the function's argument list) is at address "[ebp+8]". Subsequent arguments are at address "[ebp+12]" or "[ebp+16]", depending on the size of the argument pushed onto the stack at "8(ebp)".

8.9 80386 Return Values

Assembly-language routines that return values to a 80386 C language program or receive return values from 80386 C language functions must follow the 80386 C language return-value conventions. C language functions place return values that have **int**, **char**, **unsigned**, **short**, and **long** types in the **eax** register.

Floating-point values are returned to the top of the ndp 80287 stack. The following example shows the recommended instruction sequence for passing floating-point values:

```
float func(), f;
f = func(f)
    fld    DWORD PTR f
    sub   esp, 8
    fstp  QWORD PTR [esp]
    call func      ; result in ST(0)
    add   esp, 8
    fstp  DWORD PTR f
```



C User's Guide

The following example shows the recommended instruction sequence for returning floating-point values:

```
float fvalue;
return (fvalue);
    fld  fvalue           ; result in ST(0)
    pop  edx
    pop  esi
    pop  edi
    leave
    ret
```

Far pointers are returned in the **eax** and **edx** registers. The offset is contained in **eax** and the segment is contained in **edx**.

C language structure returns are returned to a buffer whose address is passed as a hidden first parameter.

The following example shows the recommended instruction

```
struct shape
{
    int stuff, to, fill, it, with;
} in, out, them();
out = them(in);

    sub    esp,20
    mov    edi,esp
    lea   edi,in           ; structure copy input
    mov    ecx,5           ; struct onto stack
    rep   movsd
    lea   eax,out         ; pass address of
    push  eax             ; assignment as extra "hidden"
    call  them            ;parameter
    add   esp,24
```

The following example shows the recommended instruction sequence for returning C language structure returns:

```
struct shape source;
return shape;
    mov    edi,[ebp+8]
    mov    esi,source
    mov    ecx,5
    rep   movsd
    pop    ebx
    pop    esi
    pop    edi
    leave
    ret
```

8.10 Exiting a 80386 Routine

Before returning control from an assembly-language routine to a 80386 C language program, restore the **ebp**, **esi**, **edi**, and **ebx** registers. The following example illustrates the recommended instruction sequence for exiting a routine:

```
pop    ebx
pop    esi
pop    edi
leave
ret
```

This sequence does not save the **eax**, **ecx**, or **edx** register. These registers are scratch registers for use by the compiler. If the routine modifies segment register **es**, **ss**, or **ds**, the routine must preserve the modified segment registers. The sequence does not remove arguments from the stack. This is the responsibility of the calling routine.

8.11 80386 Program Example

The following example illustrates a 80386 C language function that can be written as an assembly-language routine. The function takes two integer arguments and adds them together, returning the resultant value.

```
int add(i, j)
int i, j;
{
return(i + j);
}
```

If written as an assembly-language routine, this function must save the proper registers, retrieve the arguments from the stack, add the arguments, place the return value in the **eax** register, then restore the proper registers and return control to the calling routine. The following is an example of how the routine can be written:

C User's Guide

```
_add:
    push  ebp
    mov   ebp, esp
    push  edi
    push  esi
    push  ebx

    mov   eax, [ebp+8]
    add   eax, [ebp+12]

    pop   ebx
    pop   esi
    pop   edi
    mov   esp, ebp
    pop   ebp
    ret
```

Note

In the above assembly-language routine, it is not necessary to save the contents of the **esi**, **edi**, and **ebx** registers because the routine does not modify their contents. If the **esi**, **edi**, or **ebx** register was modified by the routine, its contents must be saved.

If the C language function is to be called by an assembly-language routine, the routine must contain instructions that push the arguments onto the stack in the proper order, call the function, and clear the stack. It can then use the return value in the **eax** register. The following is an example of the instructions that perform this task:

```
push  <j value>
push  <i value>
call  _add
add   esp, 8
```

Chapter 9

Error Processing

- 9.1 Introduction 9-1
- 9.2 Using the Standard Error File 9-1
- 9.3 Using the errno Variable 9-2
- 9.4 Printing Error Messages 9-2
- 9.5 Using Error Signals 9-3
- 9.6 Encountering System Errors 9-4

9.1 Introduction

The XENIX system automatically detects and reports errors that occur when using standard C library functions. Errors range from problems with accessing files to allocating memory. In most cases, the system simply reports the error and lets the program decide how to respond. The XENIX system terminates a program only if a serious error has occurred, such as a violation of memory space.

This chapter explains how to process errors, and describes the functions and variables a program may use to respond to errors.

9.2 Using the Standard Error File

The standard error file is a special output file that can be used by a program to display error messages. The standard error file is one of three standard files (standard input, output, and error) automatically created for the program when it is invoked.

The standard error file, like the standard output, is normally assigned to the user's terminal screen. Thus, error messages written to the file are displayed on the screen. The file can also be redirected by using the shell's redirection symbol (>) For example, the following command redirects the standard error file to the file *errorlist*:

```
dc 2>errorlist
```

In this case, subsequent error messages are written to the given file.

The standard error file, like the standard input and standard output, has predefined file pointer and file descriptor values. The file pointer **stderr** may be used in stream functions to copy data to the error file. The file descriptor **2** may be used in low-level functions to copy data to the file. For example, in the following program fragment, **stderr** is used to write the message "Unexpected end of file" to the standard error file.

```
if ( (c=getchar()) == EOF)
    fprintf(stderr, "Unexpected end of file.\n");
```

The standard error file is not affected by the shell's pipe symbol (|). This means that even if the standard output of a program is piped to another program, errors generated by the program will still appear at the terminal screen (or in the appropriate file if the standard error is redirected).



C User's Guide

9.3 Using the `errno` Variable

The `errno` variable is a predefined external variable which contains the error number of the most recent XENIX system function error. Errors detected by system functions, such as access permission errors and lack of space, cause the system to set the `errno` variable to a number and return control to the program. The error number identifies the error condition. The variable may be used in subsequent statements to process the error.

The file `errno.h` contains manifest constant definitions for each error number, and the external declaration of `errno`. These constants may be used in any program in which the line:

```
#include <errno.h>
```

is placed at the beginning of the program. The meaning of each manifest constant is described in "Error Messages" of the *XENIX C Library Guide*.

The `errno` variable is typically used immediately after a system function has returned an error. In the following program fragment, `errno` is used to determine the course of action after an unsuccessful call to the `open` function:

```
if ( (fd=open("accounts", O_RDONLY)) == -1 )
    switch (errno) {
        case(EACCES):
            fd = open("/usr/tmp/accounts",O_RDONLY);
            break;
        default:
            exit(errno);
    }
```

In this example, if `errno` is equal to `EACCES` (a manifest constant), permission to open the file `accounts` in the current directory is denied, so the file is opened in the directory `/usr/tmp` instead. If the variable is any other value, the program terminates.

9.4 Printing Error Messages

The `perror` function copies a short error message describing the most recent system function error to the standard error file. The function call has the form:

```
perror (s);
```

where *s* is a pointer to a string containing additional information about the error.

The **perror** function places the given string before the error message and separates the two with a colon (:). Each error message corresponds to the current value of the **errno** variable. For example, in the following program fragment, **perror** displays the message:

```
accounts: Permission denied.
```

if **errno** is equal to the constant **EACCES**:

```
if ( errno == EACCES ) {
    perror("accounts");
    fd = open ("/usr/tmp/accounts", O_RDONLY);
}
```

All error messages displayed by **perror** are stored in an array named **sys_errno**, an external array of character strings. The **perror** function uses the variable **errno** as the index to the array element containing the desired message. For more information on the **perror** function, see the **perror(S)** manual page in the *XENIX Reference*.

9.5 Using Error Signals

Some program errors cause the XENIX system to generate error signals. These signals are passed back to the program that caused the error and normally terminate the program. The most common error signals are SIGBUS, the bus error signal, SIGFPE, the floating point exception signal, SIGSEGV, the segment violation signal, SIGSYS, the system call error signal, and SIGPIPE, the pipe error signal. Other signals are described in **signal(S)** in the *XENIX Reference*.

A program can, if necessary, catch an error signal and perform its own error processing by using the **signal** function. This function, as described in the "Using Signals" chapter of the *XENIX Programmer's Guide*, can set the action of a signal to a user-defined action. For example, the function call:

```
signal(SIGBUS, fixbus);
```

sets the action of the bus error signal to the action defined by the user-supplied function *fixbus*. Such a function usually attempts to remedy the problem, or at least display detailed information about the problem before terminating the program.



C User's Guide

For details about how to catch, redefine, and restore these signals, see "Using Signals" in the *XENIX Programmer's Guide*.

9.6 Encountering System Errors

Programs that encounter serious errors, such as hardware failures or internal errors, generally do not receive detailed reports on the cause of the errors. Instead, the XENIX system treats these errors as "system errors," and reports them by displaying a system error message on the system console. This section briefly describes some aspects of XENIX system errors and how they relate to user programs. For a complete list and description of XENIX system errors, see **messages(M)** in the *XENIX Reference*.

Most system errors occur during calls to system functions. If the system error is recoverable, the system will return an error value to the program and set the **errno** variable to an appropriate value. No other information about the error is available.

Although the system lets two or more programs share a given resource, it does not keep close track of which program is using the resource at any given time. When an error occurs, the system returns an error value to all programs regardless of which caused the error. No information about which program caused the error is available.

System errors that occur during routine I/O operations initiated by the XENIX system itself generally do not affect user programs. Such errors cause the system to display appropriate system error messages on the system console.

Some system errors are not detected by the system until after the corresponding function has returned successfully. Such errors occur when data written to a file by a program has been queued for writing to disk at a more convenient time, or when a portion of data to be read from disk is found to already be in memory and the remaining portion is not read until later. In such cases, the system assumes that the subsequent read or write operation will be carried out successfully and passes control back to the program along with a successful return value. If operation is not carried out successfully, it causes a delayed error.

When a delayed error occurs, the system usually attempts to return an error on the next call to a system function that accesses the same file or resource. If the program has already terminated or does not make a suitable call, then the error is not reported.

Appendix A

Converting from Previous Versions of the Compiler

- A.1 Introduction A-1
- A.2 Differences between Versions 5.0 and 4.0 A-1
 - A.2.1 Enhancements and Additions A-1
 - A.2.2 Changes to the Language Syntax A-2
 - A.2.3 New Features for the XENIX Implementation of C A-3
- A.3 Differences between Versions 4.0 and 3.0 A-5
 - A.3.1 Enhancements and Additions A-5
 - A.3.2 Changes in the Language Syntax A-5
 - A.3.3 New Features for the XENIX Implementation of C A-8

Converting from Previous Versions of the Compiler

A.1 Introduction

This appendix describes differences between Version 5.0 and Version 4.0, and between Version 4.0 and Version 3.0, of the XENIX C Compiler. If you have an earlier version of the compiler, or if you have written programs for an earlier version, this chapter can help you convert your previous source code. The actions necessary to convert source code depend on which of the earlier versions you have used.



Version 5.0 is an update of Version 4.0. Generally, the two versions are compatible: most C source code written for Version 4.0 should compile without change on the Version 5.0 compiler, although there are erroneous C constructs allowed in Version 4.0 that are not allowed in Version 5.0, and changes in the emerging ANSI C standard may force changes in source programs (for more information, see the *XENIX C Language Reference*). In some cases you may be able to enhance your programs by revising them to take advantage of new library functions and other features available with Version 5.0.

A.2 Differences between Versions 5.0 and 4.0

Changes in Version 5.0 since Version 4.0 fall into the following categories:

- Enhancements and additions to the compiler software to allow for more flexible programming, improved code generation, and increased support for the developing ANSI standard
- Changes in the language syntax
- Changes in function operations, primarily to conform to the specifications for these functions in the ANSI standard.

These features and the changes required to take advantage of them are discussed in the following sections.

A.2.1 Enhancements and Additions

Enhancements for Version 5.0 include the following:

- Improved code generation, including loop optimization; improved large-model code generation; and intrinsic functions
- Faster compilation speed

XENIX C Compiler User's Guide

- Support for code that will be loaded into read-only memory (ROM)
- New error-message numbering

A.2.2 Changes to the Language Syntax

Some Version 5.0 changes were made to the C language syntax to make it conform more closely to the new ANSI standard. Most of these changes do not affect source code written for the Version 4.0 compiler. The changes are summarized as follows:

- Full function prototyping is supported in Version 5.0. A function prototype is a forward declaration containing the types and, optionally, names of the parameters (if any) expected in the function call. It can also include identifiers for the arguments, though they go out of scope at the end of the prototype. Prototypes allow the compiler to perform type checking on the actual arguments passed when the function is called. If the compiler does not find a prototype, the first occurrence of the function (definition or call) is used as the basis of a prototype for that function. That prototype is used to perform type checking against subsequent calls, subsequent declarations, or the definition. For more information about function prototyping, see the *XENIX C Language Reference*.
- The **const** and **volatile** type specifiers have been implemented for Version 5.0. The **const** type specifier declares an object as an unmodifiable lvalue. It can be used for objects of any fundamental or aggregate type or for pointers to objects of any type. The **volatile** type specifier is implemented syntactically, but not semantically. For more information, see the *XENIX C Language Reference*.

Note

Programs that currently use **const** or **volatile** as identifiers must be recoded to use other names.

- In Version 5.0, variables of **enum** type are treated as if they were of **int** type in all cases. Therefore, **enum** variables can be used in indexing expressions and as operands of all relational and arithmetic operators.

Converting from Previous Versions of the Compiler

- String concatenation is supported in Version 5.0. This feature causes adjacent string literals to be concatenated into a single string literal. This means, for example, that instead of using a backslash before a new-line character to indicate continuation of a long string literal, the literal can simply be broken into two or more quoted string literals on separate lines. For more information, see the *XENIX C Language Reference*.
- New preprocessor features in Version 5.0 include the string operator (`#`), which allows arguments in macro expansions to be expanded into a string literal containing the expanded argument; and the concatenation operator (`##`), which concatenates the tokens on either side of the operator into a new token in macro expansions. For more information, see the *XENIX C Language Reference*.

Note

Previous versions of XENIX C allowed expansion of macro formal arguments appearing in string literals and character constants. Programs that rely on this feature *must* be recoded to use the stringizing operator. For information, see the discussion of string literals in the *XENIX C Language Reference*.

- The **long double** data type is now supported; the **long float** data type is no longer supported.
- The three-digit forms of hex escape sequences (`\ddd`) and octal escape sequences (`\ddd`) are now supported.
- The unary plus (`+`) operator is allowed, but ignored semantically.

A.2.3 New Features for the XENIX Implementation of C

The following new `cc` command options have been added to the XENIX implementation of the XENIX C Compiler for Version 5.0:

XENIX C Compiler User's Guide

Option	Effect
-Oi	Enables intrinsic code generation for all available functions
-Ol	Enables loop optimizations for an entire program
-Op	Forces consistent precision in floating-point math operations
-Sl	Specifies the line width for source listings
-Sp	Specifies the number of lines per page for source listings
-Ss	Specifies subtitles for source listings
-St	Specifies titles for source listings
-Tc	Tells the compiler that the following file is a C source file
-Zp	Packs structures on one-, two-, or four-byte boundaries

The following new pragmas have been added to the XENIX implementation of the XENIX C Compiler for Version 5.0 to control the specified features on a local basis:

Pragma	Effect
loop_opt	Turns loop optimizations on and off
pack	Specifies packing alignment for structures
intrinsic	Specifies which functions are compiled as intrinsic functions
function	Specifies which functions are compiled as standard function calls
same_seg	Tells the compiler to assume that specified variables are allocated in the same far data segment
alloc_text	Specifies modules to be grouped into a specified far code segment

Converting from Previous Versions of the Compiler

Note that the existing `check_stack` pragma uses the following new format for specifying arguments:

```
#pragma check_stack([[on|off]])
```



A.3 Differences between Versions 4.0 and 3.0

Changes between Versions 4.0 and 3.0 fall into the same categories as those between Versions 5.0 and 4.0.

- Enhancements and additions to the compiler software to allow for more flexible programming, improved code generation, and increased support for the developing ANSI standard
- Changes in the language syntax

These features and the changes required to take advantage of them are discussed in the following sections.

A.3.1 Enhancements and Additions

Enhancements for Version 4.0 include the following:

- New options for `cc` and `ld`
- Improved code optimization
- New memory models (`compact` and `huge`)
- Source listings
- Numbered error messages
- Huge arrays, allowing a single array to be larger than 64K

These changes should have no effect on Version 3.0 source code.

For information on changes to the syntax of the `cc` command line, see the “Compiling with the `cc` Command” chapter of this guide.

A.3.2 Changes in the Language Syntax

Some Version 4.0 changes were made to the C language syntax to make it conform more closely to the new ANSI standard. Most of these changes do not affect source code written for the Version 3.0 compiler. The changes are summarized as follows:

XENIX C Compiler User's Guide

- The `\a` escape sequence represents the bell (or alert) character in Version 4.0.

You can make your source code more portable by using `\a` instead of `\x7`. For more information, see the *XENIX C Language Reference*.

- The **signed** keyword was added.

The **signed** keyword can be used to specify signed items. This keyword is particularly useful for declaring signed **char** types in programs compiled with the **-J** option. (**-J** changes the default mode for the **char** type to unsigned.) For more information on signed types, see the *XENIX C Language Reference*.

- The syntax was changed for making function calls with a variable number of arguments.

The following two declarations contrast the Version 3.0 form and the Version 4.0 form:

```
int func (int,);          /* Forward declaration in
                          ** Version 3.0 syntax
                          */

int func (int,...);      /* Forward declaration in
                          ** Version 4.0 syntax
                          */
```

This change was made to conform to changes in the ANSI standard for the C language. Both forms are supported in Version 4.0 of the XENIX C Compiler. XENIX recommends the use of the Version 4.0 form in all programs.

- Prior to Version 4.0, the compiler allowed arbitrary strings of characters after a syntactically correct preprocessor command. To conform to the new ANSI standard, this was disallowed in Version 4.0.

Beginning with Version 4.0, the following usage, for example, causes the compiler to generate a warning message:

```
#endif    Block ends here
```

In Versions 4.0 and later, such strings must be enclosed in comment delimiters, as in the following example:

```
#endif    /* Block ends here */
```

Converting from Previous Versions of the Compiler

- Names of types defined with **typedef** are not keywords in Version 4.0, as they were in Version 3.0. In Version 4.0, these names are in the same naming class as names of functions and variables, and can be redefined in a nested block.

For more information, see the *XENIX C Language Reference*.

- Beginning with Version 4.0, the **#pragma** directive is supported.

A “pragma” is an instruction to the compiler. Its syntax is similar to the syntax of preprocessor directives, but its purpose is different. The syntax is as follows:

```
#pragma charstring
```

The only pragma instruction supported in the XENIX C Compiler, Version 4.0, is the **check_stack** pragma. This pragma is specific to XENIX, and is discussed in greater detail in the “Compiling with the cc Command” chapter of this guide.

- Hexadecimal and octal integer constants are handled differently in Version 4.0 than they are in Version 3.0.

For more information, see the *XENIX C Language Reference*.

- The extended keywords **fortran**, **pascal**, **cdecl**, **near**, and **huge** are enabled by default in Version 4.0. They can be disabled by giving the **-Za** option on the command line.
- Two new reserved words, **const** and **volatile**, were added but not implemented for Version 4.0.
- In Version 3.0, when a near pointer is converted to type **long int**, it is first converted to type **short int**, then to **long int**; as a result, in Version 3.0 the expression in the **if** statement evaluates as true in the following fragment:

```
char *ptr = NULL;
long i;

i = (long) ptr;
if (i == 0L) {
    .
    .
    .
}
```


XENIX C Compiler User's Guide

In Version 4.0, the conversion order of near pointers to long integers was changed so that it conforms to the order in which the compiler does all other conversions that increase the length of a variable: first the size, then the mode. (For example, the compiler converts a variable with type **char** to type **unsigned long** by first converting it to **signed long**, then to **unsigned long**.) Because of this change, the preceding code now converts *ptr* to a far pointer by loading the appropriate segment register value, then changing that to a long integer. The expression following the *if* statement would most likely be false in Version 4.0, since the segment registers do not usually contain 0.

A.3.3 New Features for the XENIX Implementation of C

The following features were added to the XENIX implementation of the C compiler for Version 4.0:

- Two new memory models: huge and compact
- The **huge**, **signed**, and **cdecl** keywords
- A pragma (**check_stack**) to control stack checking
- The **-J** option to change the default mode for the **char** type to unsigned
- The **-Gc** option to specify the alternative calling sequence and naming conventions used in XENIX Pascal and XENIX FORTRAN

These features are discussed in “Working with Memory Models.” In most cases, they will not affect existing Version 3.0 source code. However, you may be able to improve your existing programs by modifying them to take advantage of the new memory models or the **huge** keyword.

Appendix B

Writing Portable Programs

- B.1 Introduction B-1
- B.2 Program Portability B-2
- B.3 Machine Hardware B-2
 - B.3.1 Byte Length B-2
 - B.3.2 Word Length B-2
 - B.3.3 Storage Alignment B-3
 - B.3.4 Byte Order in a Word B-4
 - B.3.5 Bit Fields B-5
 - B.3.6 Pointers B-6
 - B.3.7 Address Space B-8
 - B.3.8 Character Set B-8
- B.4 Compiler Differences B-9
 - B.4.1 Signed/Unsigned char and Sign Extension B-9
 - B.4.2 Shift Operations B-9
 - B.4.3 Identifier Length B-10
 - B.4.4 Register Variables B-10
 - B.4.5 Type Conversion B-10
 - B.4.6 Functions with a Variable Number of Arguments B-12
 - B.4.7 Side Effects and Evaluation Order B-12
- B.5 Environment Differences B-13
- B.6 Portability of Data B-14
- B.7 Type-Size Summary B-14
- B.8 Byte-Ordering Summary B-16

B.1 Introduction

The standard definition of the C programming language leaves many details to be decided in specific implementations of the language. These unspecified features of the language detract from its portability and must be studied when attempting to write portable C code.

Most of the issues affecting C portability arise from differences either in target-machine hardware or in compilers. C was designed to compile efficient code for the target machine (initially a Digital Equipment Corporation PDP-11®), so many of the language features not precisely defined are those that reflect a particular machine's hardware characteristics.

This appendix highlights the various aspects of C that may not be portable across different machines and compilers. It also briefly discusses the portability of a C program in terms of its environment. The environment is determined by the system calls and library routines a program uses during execution, file path names it requires, and other items not guaranteed to be constant across different systems.

The C language has been implemented on many different computers with widely different hardware characteristics, from small eight-bit microprocessors to large mainframes. This appendix is concerned with the portability of C code in the MS-DOS and XENIX programming environments. This is a more restricted problem to consider, since all MS-DOS and XENIX operating systems to date run on hardware with the following basic characteristics:

- ASCII character set
- Eight-bit bytes
- Two-byte or four-byte integers
- Two's-complement arithmetic

These features are not formally defined for the language and may not be found in all implementations of C. However, the remainder of this appendix is devoted to those systems where these basic assumptions hold.

The C language definition contains no specification of how input and output are performed. These specifications are left to system calls and library routines on individual systems. Within XENIX systems there are system calls and library routines that can be considered portable. This version of the XENIX C Compiler includes system calls and library

XENIX C User's Guide

routines that can be considered portable across XENIX and MS-DOS systems. The run-time library for the XENIX C Compiler for MS-DOS is composed primarily of XENIX-compatible routines. By restricting the use of XENIX routines to those included in the MS-DOS library, the XENIX programmer can develop MS-DOS programs in the XENIX environment; C programs written on MS-DOS are easily portable to XENIX.

B.2 Program Portability

A program is “portable” if it can be compiled and run successfully on different machines without alteration. There are many ways to write portable programs. One way is to avoid using inherently nonportable language features. Another is to isolate any nonportable interactions with the environment, such as I/O to nonstandard devices. For example, programs should avoid hard-coded path names unless a path name is common to all systems.

Files required at compile time (such as include files) may also introduce nonportability if the path names used are not the same on all machines. In some cases, include files containing machine-specific definitions can be used to make the source code itself portable.

B.3 Machine Hardware

Differences in the hardware of the various target machines and differences in the corresponding C compilers cause the greatest number of portability problems. This section lists problems commonly encountered.

B.3.1 Byte Length

By definition, the **char** data type in C must be large enough to hold as positive integers all members of a machine's character set. For the machines described in this appendix, the **char** size is an eight-bit byte.

B.3.2 Word Length

The size of the basic data types for a given implementation are not formally defined in the C language. Therefore, they often follow the most natural size for the underlying machine. It is safe to assume that **short** is no longer than **long**. Beyond that, no assumptions are portable. For example, on some machines **short** is the same length as **int**, whereas on others **long** is the same length as **int**.

Two areas where different **int** sizes affect program portability are the following:

1. Array indexing. For very large arrays, a variable of type **int** may not be long enough to store the indices of the highest-numbered array elements.
2. Pointer subtraction. On some machines, an **int** variable may not be long enough to store the results of pointer subtraction. See the section on “Pointers,” for more information about this problem.



Programs that need to assume the size of a particular data type should avoid hard-coded constants where possible. Such information can usually be written in a fairly portable way. For example, the maximum positive integer (on a two's-complement machine) can be obtained with the following directive:

```
#define MAXPOS ((int)((unsigned)-1) >> 1)
```

This is preferable to the following code:

```
#ifdef PDP11
#define MAXPOS 32767
#else
.
.
.
#endif
```

To find the number of bytes in an **int**, use **sizeof(int)** rather than 2, 4, or some other nonportable constant.

B.3.3 Storage Alignment

The C language defines no particular layout for storage of data items relative to each other. The layout for storage of structure elements, or unions within the structure or union, is also left undefined by the language.

Some processors require that data types longer than one byte be aligned on even-byte address boundaries. Others, such as the 8086/8088, have no such hardware restriction. However, even with these machines, most compilers generate code that aligns words, structures, arrays, and long

XENIX C User's Guide

words on even addresses or on even long-word addresses. Therefore, the following code sequence may give different results, depending on specific alignment requirements on different machines:

```
struct s_tag {
    char c;
    int i;
};
printf("%d\n", sizeof(struct s_tag));
```

This variation in data storage has two major implications: data accessed as nonprimitive data types are not portable; and code that makes assumptions about the layout on a particular machine is not portable.

Therefore, unions containing structures are nonportable if the union is used to access the same data in different ways. Unions are only likely to be portable if they are used exclusively to store different data in the same space at different times. For example, if the following union were used to obtain four bytes from a long word, the code would not be portable:

```
union {
    char c[4];
    long lw;
} u;
```

The **sizeof** operator should always be used when reading and writing structures, as follows:

```
struct s_tag st;
.
.
.
write(fd, &st, sizeof(st));
```

Using the **sizeof** operator ensures portability of the source code, but does not produce a portable data file. Portability of data is discussed in the "Portability of Data" section.

B.3.4 Byte Order in a Word

The variation in byte order in a word affects the portability of data more than the portability of source code. However, any program that makes

use of knowledge of the internal byte order in a word is not portable. For example, on some XENIX systems there is an include file, **misc.h**, that contains the following structure declaration:

```

/*
 * structure to access an
 * integer in bytes
 */
struct {
    char  lobyte;
    char  hibyte;
};

```



With certain less-restrictive compilers, this declaration could be used to access the high- and low-order bytes of an integer separately and in a completely nonportable way. The correct way to do this is to use mask and shift operations to extract the required byte, as shown in the following example:

```

#define LOBYTE(i) (i & 0xff)
#define HIBYTE(i) ((i >> 8) & 0xff)

```

These definitions provide a portable way to extract the least-significant and the next-least-significant bytes of an integer. Since the **int** type can be either two or four bytes, depending on the machine, even these definitions do not provide a completely portable way to access the bytes of an **int**.

One result of the byte-ordering problem is that the following code sequence will not always perform as intended:

```

int c = 0;

read(fd, &c, 1);

```

On machines where the low-order byte is stored first, the value of *c* is the byte value read. On other machines, the byte is read into some byte other than the low-order one, so the value of *c* is different.

B.3.5 Bit Fields

Bit fields are not implemented in all C compilers. The XENIX C Compiler implements bit fields and allows them to have any length up to the size of a **long**. However, in many implementations no bit field may be larger than an **int**, and no bit field can overlap an **int** boundary. If necessary, the compiler will leave gaps and move to the next **int** boundary. To ensure portability no individual field should exceed 16 bits.

XENIX C User's Guide

The C language makes no guarantees about whether bit fields are assigned left to right or right to left. Therefore, although bit fields may be useful for storing flags and other small data items, their use in unions to dissect bits from other data is definitely nonportable.

B.3.6 Pointers

The C language is fairly generous in allowing manipulation of pointers, to the extent that most compilers do not generate warnings for nonportable pointer operations. A common nonportable use of pointers is the use of casts to assign one pointer to another pointer of a different data type. This practice usually makes some assumption about the internal byte ordering and layout of the data type, and is therefore nonportable. In the following code, the byte order in the array *c* is not portable:

```
char c[4];
long *lp;

lp = (long *)&c[0];
*lp = 0x12345678L;
```

Code like this is usually unnecessary or invalid. It is acceptable, however, when the **malloc** function is used to allocate space for variables that do not have **char** type. The routine is declared as type **char ***, and the return value is cast to the type to be stored in the allocated memory. If this type is not **char ***, then a compiler may issue a warning concerning illegal type conversion. In addition, the **malloc** function is designed always to return a starting address suitable for storing all types of data. A compiler may not know this, so it may give an additional warning about possible data-alignment problems. In the following example, **malloc** is used to obtain memory for an array of 50 integers:

```
extern char *malloc( );
int *ip;

ip = (int *)malloc(50 * sizeof(int));
```

This example will elicit a warning message from some compilers.

The *XENIX C Language Reference* states that a pointer can be assigned (or cast) to an integer large enough to hold it. Note that the size of the **int** type depends on the given machine and implementation. This type is

long on some machines and **short** on others. The size may also be modified by **near** and **far** declarations. In general, do not assume that the following statement is always true:

```
sizeof(char *) == sizeof(int)
```

For example, the following construction is nonportable, assuming that the function identifier *func* is not previously declared:

```
int p;
p = (char *)func( );
```

This example assumes that a **char** pointer has the same length as an **int**.

Another consequence of different-sized **int** types on different machines is that pointer subtraction may not give the expected results. As an example of this case, subtracting pointers to the beginning and end of a very large array may give a result that is too large to store in an **int** variable, as shown in the following example:

```
int arr[20000], *b = arr, *e = &arr[20000];
int diff;
diff = e - b; /* result too large to store in
              int variable diff */
```

To correct this problem, coerce the result of the pointer subtraction **long** type, then assign the result to a variable of **unsigned int** type, as shown in the following example:

```
unsigned int udiff;
udiff = (long) ((int huge *)e - (int huge *)b);
```

In most implementations, the null pointer value **NULL** is defined to be the **int** value 0. The length of the 0 value can lead to problems for functions that expect pointer arguments longer than an **int**. For portable code, always use the following form to pass a **NULL** value of the correct size:

```
func( (char *)NULL );
```



B.3.7 Address Space

The address space available to a program varies considerably from system to system. Some small processors allow only 64K for program text and data combined. Others allow up to 64K of data and 64K of program text. Larger machines may allow considerably more text and possibly more data as well.

Large programs, or programs that require large data areas, may have portability problems on small machines.

B.3.8 Character Set

The C language does not require the use of the ASCII character set. In fact, the only character-set requirements are that all characters must fit in the **char** data type, and all characters must have positive values.

In the ASCII character set, all characters have values between 0 and 127 and therefore can be represented in seven bits. On an eight-bits-per-byte machine they are all positive, regardless of whether **char** is treated as signed or unsigned.

A set of character-classification macros is included as part of the run-time library for the XENIX C Compiler. These macros should be used for most tests on character quantities. The macros are defined in the include file **ctype.h**, and described in the *XENIX C Library Guide*. They appear on the pages headed **isalnum-isascii** and **isctrl-isxdigit**.

The character-classification macros provide insulation from the internal structure of the character set. In addition, the names of the macros are often more meaningful than the equivalent line of code. Compare the following two lines:

```
if (isupper(c))  
  
if ((c >= 'A') && (c <= 'Z'))
```

With some of the other macros, such as **isxdigit** to test for a hexadecimal digit, the advantage is even greater. Also, the internal implementation of the macros makes them more efficient than an explicit test with an **if** statement.

B.4 Compiler Differences

There are a number of C compilers running under various operating systems. The main areas of differences between compilers are outlined in this section.

B.4.1 Signed/Unsigned char and Sign Extension



The current state of the signed versus unsigned **char** problem is best described as unsatisfactory. The sign-extension problem is a serious barrier to writing portable C, and the best solution at present is to write defensive code that does not rely on particular implementation features.

B.4.2 Shift Operations

The left-shift operator (`<<`) shifts its operand a number of bits left, filling vacated bits with zeros. This is called a logical shift. When the right-shift operator (`>>`) is applied to an unsigned quantity, it performs a logical-shift operation; when it is applied to a signed quantity, the vacated bits may be filled with zeros (logical shift) or with sign bits (arithmetic shift). The decision is implementation dependent, and code that assumes a particular implementation is nonportable.

With compilers that use arithmetic right shift, it is necessary to shift and mask the appropriate number of high-order bits to avoid sign extension, as follows:

```
char c;
c = (c >> 3) & 0x1f;
```

You can also avoid sign extension by using the divide operator (`/`) as follows:

```
char c;
c = c / 8;
```

B.4.3 Identifier Length

The use of long symbols and identifier names will cause portability problems with some compilers. To avoid these problems, a program should keep the following symbols as short as possible:

- C preprocessor symbols
- C local symbols
- C external symbols

Some loaders also place restrictions on the number of unique characters in C external symbols. Symbols unique in the first six characters are unique to most C-language processors.

In some C implementations, the case of letters in identifiers is not significant.

B.4.4 Register Variables

The number and type of register variables in a function depend on the machine hardware and the compiler. Excess and invalid register declarations are treated as nonregister declarations and should not cause a portability problem. On an 8086 or 8088 processor, up to two register declarations are significant, and they must be applied to types of size **int** or smaller.

Since the compiler ignores excess variables of **register** type, the most important **register**-type variables should be declared first. In this way, register variables that the compiler ignores will be those that are the least important.

B.4.5 Type Conversion

The C language has some rules for implicit type conversion; it also allows explicit type conversions by type casting. The most common portability problem in implicit type conversion is unexpected sign extension. This is a potential problem whenever something of type **char** is compared with an **int**.

The following example will never evaluate true on a machine that sign-extends **char** types but treats hexadecimal numbers as unsigned:

```
char c;

if(c == 0x80) {
    .
    .
    .
}
```



The following construction is also nonportable:

```
char c;
unsigned int u;

if (u == (unsigned)c) {
    .
    .
    .
}
```

Two problems can arise in the preceding example:

1. The **char** type may be considered either signed or unsigned, depending on the implementation.
2. For implementations that consider the **char** type to be signed, two different methods of carrying out the conversion are possible: the **char** value may be sign extended to **int** type first, then converted to **unsigned** type; or the **char** type may be converted to an unsigned type of the same size, then zero extended to **int** length.

The only safe comparison between **char** type and **int** is the following:

```
int c;

if(c == 'x') {
    .
    .
    .
}
```

This comparison is reliable because C guarantees all character constants to be positive.

Type conversion also occurs when arguments are passed to functions. Types **char** and **short** become **int**. Extending the **char** type can produce unexpected results. For example, the following program yields a result of -128 on some machines:

```
char c = 128;
printf("%d\n", c);
```

The unexpected negative value is produced because *c* is converted to **int** when it is passed to the **printf** function. The function itself has no knowledge of the original type of the argument and is expecting an **int**. The correct way to handle this situation is to code defensively and allow for the possibility of sign extension, as in the following example:

```
char c = 128;
printf("%d\n", c & 0xff);
```

B.4.6 Functions with a Variable Number of Arguments

Functions with a variable number of arguments present a particular portability problem if the type of the arguments is also variable. In such cases the code is dependent on the size of various data types. For portability, these cases should be avoided.

B.4.7 Side Effects and Evaluation Order

The C language makes few guarantees about the order of evaluation of operands in an expression or arguments to a function call. Therefore, the following statement is almost never portable:

```
func(i++, i++);
```

Even the following statement is unwise if **func** is ever likely to be replaced by a macro, since the macro may use *i* more than once:

```
func(i++);
```

Certain XENIX-compatible macros commonly appear in user programs; some of these use their argument only once, and therefore can safely be called with a side-effect argument. To determine whether a macro handles side effects correctly, examine the code for that macro to see whether or not the argument is evaluated more than once.

Operands to the following operators are guaranteed to be evaluated left to right:

```
,      &&   ||   ?:
```

Note that the comma operator here is a separator for two C statements. A list of items separated by commas in a declaration list is not guaranteed to be processed left to right. Therefore, the following declaration on an 8086 or 8038 processor, where only two register variables may be declared, could give any two of the four variables **register** type, depending on the compiler:

```
register int a, b, c, d;
```

To give register storage to the most important variables, use separate declaration statements and declare the most important variables first. The order of processing of individual declaration statements is guaranteed to be sequential in the following statements:

```
register int a;
register int b;
register int c;
register int d;
```

B.5 Environment Differences

Most programs make system calls and use library routines for various services. This section indicates some of those routines that are not always portable and those that particularly aid portability.

System calls specific to an operating system are not portable if they are not present on all other operating-system implementations of C. Most of the system calls defined in the XENIX run-time library are compatible with DOS system calls and are therefore portable to a DOS environment.

Any program is nonportable that contains hard-coded path names to files or directories, or that contains user identifier numbers, log-in names, terminal lines or other system-dependent parameters. These types of constants should be in header files, passed as command-line arguments, or obtained from the environment.

Note that the members of the **printf** and **scanf** families of functions, including **fprintf**, **fscanf**, **printf**, **sprintf**, **scanf**, **vfprintf**, **vprintf**, **vsprintf**, and **sscanf**, have evolved in several ways, and some features are not completely portable. Some of the format-conversion characters have changed their meanings, in particular those relating to uppercase and lowercase in the output of hexadecimal numbers and the specification of **long** integers on 16-bit word machines. The XENIX C specifications for these routines are given in the *XENIX C Library Guide*.

Users should be wary of porting object files that reference the **setjmp** or **longjmp** functions from XENIX to MS-DOS, unless these object files were

compiled with the `-dos` option. The MS-DOS versions of these functions use a larger buffer size and may cause memory to be overwritten. Such object files can be ported from MS-DOS to XENIX without problems, and the corresponding source files can be ported in either direction.

B.6 Portability of Data

Data files are almost always nonportable across different central-processing-unit (CPU) architectures. As mentioned above, structures, unions, and arrays have varying internal layout and padding requirements on different machines. In addition, byte ordering within words and actual word length may differ.

The only way to achieve data-file portability is to write and read data files as one-dimensional character arrays. This procedure prevents alignment and padding problems if the data are written and read as characters, and interpreted that way. Thus ASCII text files can usually be moved between different machine types without significant problems.

B.7 Type-Size Summary

Table B.1 summarizes the sizes of the various data types as defined in the XENIX C Compiler, Version 5.0.

Table B.1
C Type Sizes

Type Name (Alternate Names)	Storage	Range of Values
char (signed char)	1 byte	-128 to 127
int (signed) (signed int)	Implementation dependent (2 bytes in XENIX C 5.0)	(-32,768 to 32,767 for XENIX C Version 5.0)
short (short int) (signed short)	2 bytes	-32,768 to 32,767

(signed short int)		
long (long int) (signed long) (signed long int)	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned¹ char	1 byte	0 to 255
unsigned (unsigned int)	Implementation dependent (2 bytes in XENIX C 5.0)	(0 to 65,535 for XENIX C 5.0)
unsigned short (unsigned short int)	2 bytes	0 to 65,535
unsigned long (unsigned long int)	4 bytes	0 to 4,294,967,295
enum	Implementation dependent (2 bytes in XENIX C 5.0)	(0 to 65,535 for XENIX C 5.0)
float	4 bytes	Approximately 3.4E-38 to 3.4E+38 (7-digit precision)
double	8 bytes	Approximately 1.7E-308 to 1.7E+308 (15-digit precision)
long double	Implementation dependent (8 bytes in XENIX C 5.0)	Approximately 1.7E-308 to 1.7E+308 (15-digit precision)

¹ Any type size modified by the **unsigned** keyword can be modified by the **signed** keyword instead. The **signed** keyword is useful if the **-J** option has been used to change the default sign of the **char** type.

B.8 Byte-Ordering Summary

Tables B.2 and B.3 summarize byte ordering for **short** and **long** types, respectively. The following conventions are used in these tables:

1. The lowest physically addressed byte of the data item is **a0**; **a1** has the byte address **a0 + 1**, and so on.
2. The least-significant byte of the data item is **b0**; **b1** is the next least significant, and so on.

Since byte ordering is machine specific, any program that actually makes use of the following information is guaranteed to be nonportable:

Table B.2
Byte Ordering for Short Types

CPU	Byte Order
	a0 a1
8086	b0 b1
80286	b0 b1
PDP-11®	b0 b1
VAX-11®	b0 b1
M68000	b1 b0
Z8000®	b1 b0

Table B.3
Byte Ordering for Long Types

CPU	Byte Order
	a0 a1 a2 a3
8086	b0 b1 b2 b3
80286	b0 b1 b2 b3
PDP-11®	b2 b3 b0 b1
VAX-11®	b0 b1 b2 b3
M68000	b3 b2 b1 b0
Z8000®	b3 b2 b1 b0



Appendix C

Writing Programs

for Read-Only Memory

C.1 Introduction C-1

C.2 XENIX-Dependent Library Routines C-1

Writing Programs for Read-Only Memory

C.1 Introduction

This appendix presents information for developers who will be downloading code written with the XENIX C Compiler into read-only memory (ROM). Code of this type is more commonly known as “ROMable” code. Information is given about the run-time library routines that directly interface with XENIX.

C.2 XENIX-Dependent Library Routines

Because ROMable programs are often run outside a XENIX environment, they cannot include calls to run-time library routines that perform their operations through calls to XENIX functions. Table C.1 lists the library routines that call XENIX functions.



Table C.1
MS-DOS-Dependent Library Routines

abort	_exit	fwrite	read
access	ifclose	getch	rmdir
chdir	fgetc	getcwd	scanf
chmod	fgetchar	getpid	sopen
chsize	fgets	gets	sprintf
close	flush	getw	sscanf
creat	fopen	labs	stat
dup	fprintf	localtime	system
dup2	fputc	locking	tell
eof	fputchar	lseek	time
execl	fputs	mkdir	tmpfile
execle	fread	mktemp	unlink
execlp	freopen	open	utime
execlpe	fscanf	perror	vfprintf
execv	fseek	printf	vprintf
execve	fstat	putch	vsprintf
execvp	ftell	puts	write
execvpe	ftime	putw	

A program containing calls to any of these routines cannot run in a non-XENIX environment unless you do one of the following:

- Write replacements for these XENIX-dependent routines as needed.
- Edit the program to remove the calls to the listed routines.

XENIX C User's Guide

- Obtain the library source files from XENIX and edit them so that they do not include XENIX function calls, and write functional equivalents of the XENIX functions that can be called from your program.

Note that certain functions that are not listed above may call XENIX functions indirectly: that is, they may be part of a series of nested calls that call routines in the list.

Appendix D

C Error Messages and Exit Codes

- D.1 Introduction D-1
- D.2 Command-Line Error Messages D-1
 - D.2.1 Command-Line Fatal-Error Messages D-1
 - D.2.2 Command-Line Error Messages D-1
 - D.2.3 Command-Line Warning Messages D-4
- D.3 Compiler Error Messages D-5
 - D.3.1 Fatal-Error Messages D-7
 - D.3.2 Compilation-Error Messages D-12
 - D.3.3 Warning Messages D-28
 - D.3.4 Compiler Limits D-37
- D.4 Compiler Exit Codes D-39

D.1 Introduction

This appendix lists error messages you may encounter as you develop a program, and gives a brief description of actions you can take to correct the errors. It also describes the exit codes returned by the compiler.

D.2 Command-Line Error Messages

Messages that indicate errors on the command line used to invoke the compiler have one of the following formats:

<i>command line fatal error D1xxx: messagetext</i>	(fatal error)
<i>command line error D2xxx: messagetext</i>	(error)
<i>command line warning D4xxx: messagetext</i>	(warning error)

If possible, the compiler continues operation, printing a warning message. In some cases, command-line errors are fatal and the compiler terminates processing.



D.2.1 Command-Line Fatal-Error Messages

The following messages identify fatal errors. The compiler driver cannot recover from a fatal error; it terminates after printing the error message.

D1000 UNKNOWN COMMAND LINE FATAL ERROR

The compiler detected an unknown fatal-error condition.

D1001 could not execute '*filename*'

The compiler could not find the given file in the current working directory or any of the other directories named in the **PATH** variable.

D1002 too many open files, cannot redirect '*filename*'

No more file descriptors were available to redirect the output of the **-P** option to a file.

D.2.2 Command-Line Error Messages

When the compiler driver encounters any of the errors listed in this section, it continues compiling the program (if possible) and outputs additional error messages. However, no object file is produced.

XENIX C User's Guide

D2000 UNKNOWN COMMAND LINE ERROR

The compiler detected an unknown error condition.

D2001 too many symbols predefined with -D

Too many symbolic constants were defined using the **-D** option on the command line.

The limit on command-line definitions is normally 16; you can use the **-U** or **-u** option to increase the limit to 20.

D2002 a previously defined model specification has been overridden

Two different memory models were specified; the model specified later on the command line was used.

D2003 missing source file name

You did not give the name of the source file to be compiled.

D2007 bad *option* flag, would overwrite '*string1*' with '*string2*'

The specified option was given more than once, with conflicting arguments *string1* and *string2*.

D2008 too many *option* flags, '*string*'

Too many letters were given with the specified option (for example, with the **-O** option).

D2009 unknown *option character* in '*optionstring*'

One of the letters in the given option was not recognized.

D2012 too many linker flags on command line

You tried to pass more than 128 separate options and object files to the linker.


D2013 incomplete model specification

Not enough characters were given for the **-Astring** option. The option requires all three letters (to specify the data-pointer size, code-pointer size, and segment setup, respectively).

D2014 **-ND** not allowed with **-Ad**

You cannot rename the default data segment unless you give the **-Auxx** option (**SS != DS**, load **DS**) on the command line.

C Error Messages and Exit Codes

- D2015 assembly files are not handled
You gave a file name with an extension of **.asm** on the command line. Because the compiler cannot invoke the XENIX Macro Assembler (**masm**) automatically, it cannot assemble such files.
- D2016 **-Gw** and **-ND** *name* are incompatible
You tried to rename the default data segment to the given name when you specified the **-Gw** option.
- Renaming the default data segment is illegal in this case because the **-Gw** option requires the **-Awxx** option.
- D2017 **-Gw** and **-Au** flags are incompatible
You tried to specify the **-Auxx** option (**SS != DS**, load **DS**) with the **-Gw** option.
- Specifying **-Auxx** with **-Gw** is illegal because the **-Gw** option requires the **-Awxx** option.
- 
- D2018 cannot open linker cmd file
The response file used to pass object-file names and options to the linker could not be opened.
- This error may have occurred because another read-only file had the same name as the response file.
- D2019 cannot overwrite the source file, '*name*'
You specified the source file as an output-file name.
- The compiler does not allow the source file to be overwritten by one of the compiler output files.
- D2020 **-Gc** option requires extended keywords to be enabled (**-Ze**)
The **-Gc** option and the **-Za** option were specified on the same command line.
- The **-Gc** option requires the extended keyword **cdecl** to be enabled if library functions are to be accessible.
- D2021 invalid numerical argument '*string*'
A non-numerical string was specified following an option that required a numerical argument.
- D2022 cannot open help file, **cc.hlp**
The **-help** option was given, but the file containing the help

messages (**cc.hlp**) was not in the default directory (/usr/lib/286) or in any of the directories specified by the **PATH** environment variable.

D2023 invalid model specification - small model only

D.2.3 Command-Line Warning Messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking.

D4000 UNKNOWN COMMAND LINE WARNING

An unknown fatal condition has been detected by the compiler.

D4001 listing has precedence over assembly output

Two different listing options were chosen; the assembly listing is not created.

D4002 ignoring unknown flag '*string*'

One of the options given on the command line was not recognized and is ignored.

D4003 80186/286 selected over 8086 for code generation

Both the **-G0** option and either the **-G1** or **-G2** option were given; **-G1** or **-G2** takes precedence.

D4004 optimizing for time over space

This message confirms that the **-Ot** option is used for optimizing.

D4006 only one of **-P/-E/-EP** allowed, **-P** selected

Only one preprocessor output option can be specified at one time.

D4007 **-C** ignored (must also specify **-P** or **-E** or **-EP**)

The **-C** option must be used in conjunction with one of the preprocessor output flags, **-E**, **-EP**, or **-P**.

D4008 non-standard model -- defaulting to small model libraries

A nonstandard memory model was specified with the option. The library search records in the object model were set to use the small-model libraries.

C Error Messages and Exit Codes

D4009 threshold only for far/huge data, ignored
The **-Gt** option cannot be used in memory models that have near data pointers. It can be used only in compact, large, and huge models.

D4010 **-Gp** not implemented, ignored
The MS-DOS version of XENIX C does not support profiling.

D4011 preprocessing overrides source listing
Only a preprocessor listing was generated, since the compiler cannot generate both a source listing and a preprocessor listing at the same time.

D4012 function declarations override source listing
The compiler cannot generate both a source-listing file and the function prototype declarations at the same time.

D4013 combined listing has precedence over object listing
When **-Fc** is specified along with either **-Fl** or **-Fa**, the combined listing (**-Fc**) is created.

D4014 invalid value *number* for '*string*'. Default *number* is used
An invalid value was given in a context where a particular numerical value was expected.

D4017 conflicting stack checking options - stack checking disabled
Both the **-Ge** and the **-Gs** flags are given in one compile command (**-Ge** enables stack checking, **-Gs** disables it).

D.3 Compiler Error Messages

The error messages produced by the C compiler fall into three categories:

1. Fatal-error messages
2. Compilation-error messages
3. Warning messages

The messages for each category are listed below in numerical order, with a brief explanation of each error. To look up an error message, first



XENIX C User's Guide

determine the message category, then find the error number. All messages give the file name and line number where the error occurs.

Fatal-Error Messages

Fatal-error messages indicate a severe problem, one that prevents the compiler from processing your program any further. These messages have the following format:

filename(line) : fatal error C1xxx: messagetext

After the compiler displays a fatal-error message, it terminates without producing an object file or checking for further errors.

Compilation-Error Messages

Compilation-error messages identify actual program errors. These messages appear in the following format:

filename(line) : error C2xxx: messagetext

The compiler does not produce an object file for a source file that has compilation errors in the program. When the compiler encounters such errors, it attempts to recover from the error. If possible, it continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler stops processing.

Warning Messages

Warning messages are informational only; they do not prevent compilation and linking. These messages appear in the following format:

filename(line) : warning C4xxx: messagetext

You can use the **-W** option to control the level of warnings that the compiler generates. This option is described in the "Compiling with the cc Command" chapter of this guide.

D.3.1 Fatal-Error Messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it terminates after printing the error message.

C1000 UNKNOWN FATAL ERROR

An unknown error condition has been detected by the compiler.

C1001 Internal Compiler Error

The compiler detected an internal inconsistency.

Note that the file name refers to an internal compiler file, *not* your source file.

C1002 out of heap space

The compiler has run out of dynamic memory space. This usually means that your program has many symbols and/or complex expressions.



To correct the problem, divide the file into several smaller source files, or break expressions into subexpressions.

C1003 error count exceeds *n*; stopping compilation

Errors in the program were too numerous or too severe to allow recovery, and the compiler must terminate.

C1004 unexpected EOF

This message appears when you have insufficient space on the default disk drive for the compiler to create the temporary files it needs. The space required is approximately two times the size of the source file. This message can also occur when a comment does not have a closing delimiter (**/*), or when an **#if** directive occurs without a corresponding closing **#endif** directive.

C1005 string too big for buffer

A string in a compiler intermediate file overflowed a buffer.

C1006 write error on compiler intermediate file

The compiler was unable to create the intermediate files used in the compilation process.

XENIX C User's Guide

The following conditions commonly cause this error:

1. XENIX system file or inode table is full at time of compilation
2. Not enough space on a device containing a compiler intermediate file

C1007 unrecognized flag '*string*' in '*option*'

The *string* in the command-line *option* was not a valid option.

C1009 compiler limit possibly a recursively defined macro

The expansion of a macro exceeds the available space.

Check to see if the macro is recursively defined, or if the expanded text is too large.

C1010 compiler limit : macro expansion too big

The expansion of a macro exceeds the available space.

C1012 bad parenthesis nesting - missing '*character*'

The parentheses in a preprocessor directive were not matched; *character* is either a left or right parenthesis.

C1013 cannot open source file '*filename*'

The given file either did not exist, could not be opened, or was not found. Make sure your environment settings are valid and that you have given the correct path name for the file.

C1014 too many include files

Nesting of **#include** directives exceeds 10 levels.

C1016 #if[n]def expected an identifier

You must specify an identifier with the **#ifdef** and **#ifndef** directives.

C1017 invalid integer constant expression

The expression in an **#if** directive must evaluate to a constant.

C1018 unexpected '#elif'

The **#elif** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

C Error Messages and Exit Codes

C1019 unexpected `'#else'`
The **#else** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

C1020 unexpected `'#endif'`
An **#endif** directive appears without a matching **#if**, **#ifdef**, or **#ifndef** directive.

C1021 bad preprocessor command `'string'`
The characters following the number sign (**#**) do not form a valid preprocessor directive.

C1022 expected `'#endif'`
An **#if**, **#ifdef**, or **#ifndef** directive was not terminated with an **#endif** directive.

C1026 parser stack overflow, please simplify your program
Your program cannot be processed because the space required to parse the program causes a stack overflow in the compiler.

To solve this problem, try to simplify your program.

C1027 DGROUP data allocation exceeds 64K
More than 64K of variables was allocated to the default data segment.

For compact-, medium-, large-, or huge-model programs, use the **-Gt** option to move items into separate segments.

C1032 cannot open object listing file `'filename'`
One of the following statements about the file name or path name given (`filename`) is true:

1. The given name is not valid.
2. The file with the given name cannot be opened for lack of space.
3. A read-only file with the given name already exists.

C1033 cannot open assembly-language output file `'filename'`
One of the conditions listed under error message C1032 prevents the given file from being opened.



- C1034 cannot open source file *'filename'*
One of the conditions listed under error message C1032 prevents the given file from being opened.
- C1035 expression too complex, please simplify
The compiler cannot generate the code for a complex expression. Break the expression into simpler subexpressions and recompile.
- C1036 cannot open source listing file *'filename'*
One of the conditions listed under error message C1032 prevents the given file from being opened.
- C1037 cannot open object file *'filename'*
One of the conditions listed under error message C1032 prevents the given file from being opened.
- C1039 unrecoverable heap overflow in Pass 3
The post-optimizer compiler pass overflowed the heap and could not continue.
- Try recompiling with the **-Od** option (see "Compiling with the cc Command") or try rewriting the function containing the line that caused the error.
- C1040 unexpected EOF in source file *'filename'*
The compiler detected an unexpected end-of-file condition while creating a source listing or mingled source/object listing.
- This error probably occurred because the source file was edited during compilation.
- C1041 cannot open compiler intermediate file - no more files
The compiler could not create intermediate files used in the compilation process because no more file handles were available.
- C1042 cannot open compiler intermediate file - no such file or directory
The compiler could not create intermediate files used in the compilation process because the /tmp directory did not exist.
- C1043 cannot open compiler intermediate file
The compiler could not create intermediate files used in the compilation process. The exact reason is unknown.

C Error Messages and Exit Codes

C1044 out of disk space for compiler intermediate file

The compiler could not create intermediate files used in the compilation process because no more space was available.

To correct the problem, make more space available on the disk and recompile.

C1045 floating point overflow

The compiler generated a floating-point exception while doing constant arithmetic on floating-point items at compile time, as in the following example:

```
float fp_val = 1.0e100;
```

In this example, the double-precision constant *1.0e100* exceeds the maximum allowable value for a floating-point data item.



C1047 too many *option* flags, '*string*'

The *option* appeared too many times. The *string* contains the occurrence of the option that caused the error.

C1048 Unknown option '*character*' in '*optionstring*'

The *character* was not a valid letter for *optionstring*.

C1049 invalid numerical argument '*string*'

A numerical argument was expected instead of *string*.

C1050 code segment '*segmentname*' too large

A code segment grew to within 36 bytes of 64K during compilation.

A 36-byte pad is used because of a bug in some 80286 chips that can cause programs to exhibit strange behavior when, among other conditions, the size of a code segment is within 36 bytes of 64K.

C1052 too many *#if/#ifdef*'s

You have exceeded the maximum nesting level for *#if/#ifdef* directives.

C1053 compiler limit : struct/union nesting

Structure and union definitions were nested to more than 10 levels.

C1054 compiler limit : initializers too deeply nested

The compiler limit on nesting of initializers was exceeded. The limit ranges from 10 to 15 levels, depending on the combination of types being initialized.

To correct this problem, simplify the data type being initialized to reduce the levels of nesting, or assign initial values in separate statements after the declaration.

C1056 compiler limit : out of macro expansion space

The compiler has overflowed an internal buffer during the expansion of a macro; reduce the complexity of the macro.

C1057 unexpected EOF in macro expansion;
(missing ')')

The compiler has encountered the end of the source file while gathering the arguments of a macro invocation. Usually this is the result of a missing closing parenthesis () on the macro invocation.

C1059 out of near heap space

The compiler has run out of storage for items that it stores in the "near" (default data segment) heap. This usually means that your program has too many symbols or complex expressions. To correct the problem, divide the file into several smaller source files, or break expressions into smaller subexpressions.

C1060 out of far heap space

The compiler has run out of storage for items that it stores in the "far" heap. Usually this is the result of too many symbols in the symbol table.

D.3.2 Compilation-Error Messages

The messages listed below indicate that your program has errors. When the compiler encounters any of the errors listed in this section, it continues parsing the program (if possible) and outputs additional error messages. However, no object file is produced.

C2000 UNKNOWN ERROR

The compiler detected an unknown error condition.

C2001 newline in constant

A new-line character in a character or string constant was not in the correct escape-sequence format (\n).

C Error Messages and Exit Codes

- C2002 out of macro actual parameter space
Arguments to preprocessor macros exceeded 256 bytes.
- C2003 expected 'defined id'
The identifier to be checked in an **#if** directive was not enclosed in parentheses.
- C2004 expected 'defined(id)'
An **#if** directive caused a syntax error.
- C2005 #line expected a line number
A **#line** directive lacked the required line-number specification.
- C2006 #include expected a file name
An **#include** directive lacked the required file-name specification.
- C2007 #define syntax
A **#define** directive caused a syntax error.
- C2008 '*character*' : unexpected in macro definition
The given character was used incorrectly in a macro definition.
- C2009 reuse of macro formal '*identifier*'
The given identifier was used twice in the formal-parameter list of a macro definition.
- C2010 '*character*' : unexpected in formal list
The given character was used incorrectly in the formal-parameter list of a macro definition.
- C2011 '*identifier*' : definition too big
The given macro definitions exceeded 256 bytes.
- C2012 missing name following '<'
An **#include** directive lacked the required file-name specification.
- C2013 missing '>'
The closing angle bracket (>) was missing from an **#include** directive.



XENIX C User's Guide

- C2014 preprocessor command must start as first non whitespace
Non-white-space characters appear before the number sign (#) of a preprocessor directive on the same line.
- C2015 too many chars in constant
A character constant containing more than one character or escape sequence was used.
- C2016 no closing single quote
A character constant was not enclosed in single quotation marks.
- C2017 illegal escape sequence
The character or characters after the escape character (\) did not form a valid escape sequence.
- C2018 unknown character '0xcharacter'
The given hexadecimal number does not correspond to a character.
- C2019 expected preprocessor command, found 'character'
The given character followed a number sign (#), but it was not the first letter of a preprocessor directive.
- C2020 bad octal number 'character'
The given character was not a valid octal digit.
- C2021 expected exponent value, not 'character'
The given character was used as the exponent of a floating-point constant but was not a valid number.
- C2022 '*number*' : too big for char
The *number* was too large to be represented as a character.
- C2023 divide by 0
The second operand in a division operation (/) evaluated to zero, giving undefined results.
- C2024 mod by 0
The second operand in a remainder operation (%) evaluated to zero, giving undefined results.
- C2025 '*identifier*' : enum/struct/union type redefinition
The given identifier had already been used for an enumeration, structure, or union tag.

C Error Messages and Exit Codes

C2026 *'identifier'* : member of enum redefinition
The given identifier had already been used for an enumeration constant, either within the same enumeration type or within another enumeration type with the same visibility.

C2028 struct/union member needs to be inside a struct/union
Structure and union members must be declared within the structure or union.

This error may be caused by an enumeration declaration that contains a declaration of a structure member, as in the following example:

```
enum a {
    january,
    february,
    int march; /* structure declaration:
                ** illegal
                */
};
```



C2029 *'identifier'* : bit-fields allowed only in structs
Only structure types may contain bit fields.

C2030 *'identifier'* : struct/union member redefinition
The *identifier* was used for more than one member of the same structure or union.

C2031 *'identifier'* : function cannot be
struct/union member
The given function was declared to be a member of a structure.

To correct this error, use a pointer to the function instead.

C2032 *'identifier'* : base type with near/far/huge
not allowed
The given structure or union member was declared with the **near**, **far**, or **huge** keyword.

C2033 *'identifier'* : bit-field cannot have indirection
The given bit field was declared as a pointer (*), which is not allowed.

- C2034 *'identifier'* : bit-field type too small for number of bits
The number of bits specified in the bit-field declaration exceeded the number of bits in the given base type.
- C2035 enum/struct/union *'identifier'* : unknown size
The given structure or union had an undefined size.
- C2036 left of *'member'* must have struct/union type
The expression before the member-selection operator (->) was not a pointer to a structure or union type, or the expression before the member-selection operator (.) did not evaluate to a structure or union. In this message, *member* is a member designator in one of the following forms:
- >*identifier*
.*identifier*
- C2037 left of '->' or '.' specifies undefined struct/union *'identifier'*
The expression before the member-selection operator (-> or .) identified a structure or union type that was not defined.
- C2038 *'identifier'* : not struct/union member
The given identifier was used in a context that required a structure or union member.
- C2039 '->' requires struct/union pointer
The expression before the member-selection operator (->) was a structure or union name, not a pointer to a structure or union as expected.
- C2040 '.' requires struct/union name
The expression before the member-selection operator (.) was a pointer to a structure or union, not a structure or union name as expected.
- C2041 keyword *'enum'* illegal
The **enum** keyword appeared in a structure or union declaration, or an **enum** type definition was not formed correctly.
- C2042 signed/unsigned keywords mutually exclusive
The signed and unsigned keywords may not appear in the same declaration.
- C2043 illegal break
A **break** statement is legal only when it appears within a **do**, **for**, **while**, or **switch** statement.

C Error Messages and Exit Codes

- C2044 illegal continue
A **continue** statement is legal only when it appears within a **do**, **for**, or **while** statement.
- C2045 '*identifier*' : label redefined
The given label appeared before more than one statement in the same function.
- C2046 illegal case
The **case** keyword may appear only within a **switch** statement.
- C2047 illegal default
The **default** keyword may appear only within a **switch** statement.
- C2048 more than one default
A **switch** statement contained more than one **default** label.
- C2049 cast has illegal formal parameter list
A formal parameter list was given in a type-cast expression.
- C2050 non-integral switch expression
A switch expression was not integral.
- C2051 case expression not constant
Case expressions must be integral constants.
- C2052 case expression not integral
Case expressions must be integral constants.
- C2053 case value *number* already used
The given case value was already used in this **switch** statement.
- C2054 expected '(' to follow '*identifier*'
The context requires parentheses after the function *identifier*.
- C2055 expected formal parameter list, not a type list
An argument-type list appeared in a function definition instead of a formal parameter list.
- C2056 illegal expression
An expression was illegal because of a previous error. (The previous error may not have produced an error message.)



XENIX C User's Guide

- C2057 expected constant expression
The context requires a constant expression.
- C2058 constant expression is not integral
The context requires an integral constant expression.
- C2059 syntax error : *'token'*
The given token caused a syntax error.
- C2060 syntax error : EOF
The end of the file was encountered unexpectedly, causing a syntax error. This error can be caused by a missing closing curly brace (}) at the end of your program.
- C2061 syntax error : identifier *'identifier'*
The given identifier caused a syntax error.
- C2062 type *'type'* unexpected
The given type was misused.
- C2063 *'identifier'* : not a function
The given identifier was not declared as a function, but an attempt was made to use it as a function.
- C2064 term does not evaluate to a function
An attempt was made to call a function through an expression that did not evaluate to a function pointer.
- C2065 *'identifier'* : undefined
The given identifier was not defined.
- C2066 cast to function returning . . . is illegal
An object was cast to a function type.
- C2067 cast to array type is illegal
An object was cast to an array type.
- C2068 illegal cast
A type used in a cast operation was not a legal type.
- C2069 cast of *'void'* term to non-void
The **void** type was cast to a different type.
- C2070 illegal sizeof operand
The operand of a **sizeof** expression was not an identifier or a type name.

C Error Messages and Exit Codes

- C2071 *'class'* : bad storage class
The given storage class cannot be used in this context.
- C2072 *'identifier'* : initialization of a function
An attempt was made to initialize a function.
- C2073 *'identifier'* : cannot initialize array in function
An attempt was made to initialize the given array within a function. Arrays can be initialized only at the external level.
- C2074 cannot initialize struct/union in function
An attempt was made to initialize the given structure or union within a function. Structures and unions can be initialized only at the external level.
- C2075 *'identifier'* : array initialization needs curly braces
The braces ({ }) around the given array initializer were missing.
- C2076 *'identifier'* : struct/union initialization needs curly braces
The braces ({ }) around the given structure or union initializer were missing.
- C2077 non-integral field initializer *'identifier'*
An attempt was made to initialize a bit-field member of a structure with a nonintegral value.
- C2078 too many initializers
The number of initializers exceeded the number of objects to be initialized.
- C2079 *'expression'* uses undefined struct/union
The given identifier was declared as a structure or union type that had not been defined.
- C2082 redefinition of formal parameter *'identifier'*
A formal parameter to a function was redeclared within the function body.
- C2083 array *'identifier'* already has a size
The dimensions of the given array had already been declared.
- C2084 function *'identifier'* already has a body
The given function had already been defined.



XENIX C User's Guide

C2085 *'identifier'* : not in formal parameter list
The given parameter was declared in a function definition for a nonexistent formal parameter.

C2086 *'identifier'* : redefinition
The given identifier was defined more than once.

C2087 *'identifier'* : missing subscript
The definition of an array with multiple subscripts was missing a subscript value for a dimension other than the first dimension, as in the following example:

```
int func(a)
    char a[10][];          /* Illegal */
    {
        .
        .
        .
    }

int func(a)
    char a[][5];          /* Legal */
    {
        .
        .
        .
    }
```

C2088 use of undefined enum/struct/union *'identifier'*
The given identifier referred to a structure or union type that was not defined.

C2089 typedef specifies a near/far function
The use of the **near** or **far** keyword in a **typedef** declaration conflicted with the use of **near** or **far** for the declared item, as in the following example:

```
typedef int far FARFUNC ( );
FARFUNC near *fp;
```

C2090 function returns array
A function cannot return an array. (It can return a pointer to an array.)

C2091 function returns function
A function cannot return a function. (It can return a pointer to a function.)

C Error Messages and Exit Codes

- C2092 array element type cannot be function
Arrays of functions are not allowed; however, arrays of *pointers* to functions are allowed.
- C2093 cannot initialize a static or struct with address of automatic vars
You cannot use the address of an auto variable in the initializer of a static item.
- C2094 label '*identifier*' was undefined
The function did not contain a statement labeled with the given identifier.
- C2095 *function: actual has type void: parameter number*
An attempt was made to pass a **void** argument to a function. Formal parameters and arguments to functions cannot have type **void**; they can, however, have type **void *** (pointer to **void**).
- C2096 struct/union comparison illegal
You cannot compare two structures or unions. (You can, however, compare individual members within structures and unions.)
- C2097 illegal initialization
An attempt was made to initialize a variable using a nonconstant value.
- C2098 non-address expression
An attempt was made to initialize an item that was not an lvalue.
- C2099 non-constant offset
An initializer used a nonconstant offset.
- C2100 illegal indirection
The indirection operator (*****) was applied to a nonpointer value.
- C2101 '**&**' on constant
The address-of operator (**&**) did not have an lvalue as its operand.
- C2102 '**&**' requires lvalue
The address-of operator must be applied to an lvalue expression.



XENIX C User's Guide

- C2103 '&' on register variable
An attempt was made to take the address of a register variable.
- C2104 '&' on bit-field
An attempt was made to take the address of a bit field.
- C2105 '*operator*' needs lvalue
The given operator did not have an lvalue operand.
- C2106 '*operator*' : left operand must be lvalue
The left operand of the given operator was not an lvalue.
- C2107 illegal index, indirection not allowed
A subscript was applied to an expression that did not evaluate to a pointer.
- C2108 non-integral index
A nonintegral expression was used in an array subscript.
- C2109 subscript on non-array
A subscript was used on a variable that was not an array.
- C2110 '+' : 2 pointers
An attempt was made to add one pointer to another.
- C2111 pointer + non-integral value
An attempt was made to add a nonintegral value to a pointer.
- C2112 illegal pointer subtraction
An attempt was made to subtract pointers that did not point to the same type.
- C2113 '-' : right operand pointer
The right operand in a subtraction operation (-) was a pointer, but the left operand was not.
- C2114 '*operator*' : pointer on left; needs integral right
The left operand of the given operator was a pointer; the right operand must be an integral value.
- C2115 '*identifier*' : incompatible types
An expression contained incompatible types.
- C2116 *operator* : bad left (or right) operand
The specified operand of the given operator was illegal for that operator.

C2117 *'operator'* : illegal for struct/union
Structure and union type values are not allowed with the given operator.

C2118 negative subscript
A value defining an array size was negative.

C2119 *'typedefs'* both define indirection
Two **typedef** types were used to declare an item and both **typedef** types had indirection. For example, the declaration of *p* in the following example is illegal:

```
typedef int *P_INT;  
typedef short *P_SHORT;  
/* this declaration is illegal */  
P_SHORT P_INT p;
```

C2120 *'void'* illegal with all types
The **void** type was used in a declaration with another type.



C2121 typedef specifies different enum
An attempt was made to use a type declared in a **typedef** statement to specify both an enumeration type and another type.

C2122 typedef specifies different struct
An attempt was made to use a type declared in a **typedef** statement to specify both a structure type and another type.

C2123 typedef specifies different union
An attempt was made to use a type declared in a **typedef** statement to specify both a union type and another type.

C2125 *identifier* : allocation exceeds 64K
The given item exceeds the size limit of 64K.

The only items that are allowed to exceed 64K are huge arrays.

C2126 *identifier* : automatic allocation exceeds 32K
The space allocated for the local variables of a function exceeded the limit of 32K.

C2127 parameter allocation exceeds 32K
The storage space required for the parameters to a function exceeded the limit of 32K.

C2128 *identifier* : huge array cannot be aligned to segment boundary

The given array violated one of the restrictions imposed on huge arrays; see the "Working with Memory Models" chapter for more information on these restrictions.

C2129 static function '*identifier*' not found

A forward reference was made to a static function that was never defined.

C2130 #line expected a string containing the file name

A file name was missing from a #line directive.

C2131 attributes specify more than one near/far/huge

More than one **near**, **far**, or **huge** attribute was applied to an item, as in the following example:

```
typedef int near NINT;  
NINT far a;          /* Illegal */
```

C2132 syntax error : unexpected identifier

An identifier appeared in a syntactically illegal context.

C2133 array '*identifier*' : unknown size

An attempt was made to declare an unsized array as local variable, as in the following example:

```
int mat_add(array1)  
int array1[];      /* Legal */  
{  
int array2[];      /* Illegal */  
.  
.  
.  
}
```

C2134 *identifier* : struct/union too large

The size of a structure or union exceeded the compiler limit (2^{32} bytes).

C2135 missing ')' in macro expansion

A macro reference with arguments was missing a closing parenthesis ().

C2137 empty character constant

The illegal character constant '' was used.

C Error Messages and Exit Codes

C2138 unmatched close comment `'/*'`
The compiler detected an open-comment delimiter `(/*` without a matching close-comment delimiter `(*/)`.

This error usually indicates an attempt to use illegal nested comments.

C2139 type following `'type'` is illegal
An illegal type combination such as the following was used:

```
long char a;
```

C2140 argument type cannot be function
returning ...
A function was declared as a formal parameter of another function, as in the following example:

```
int func1(a)
    int a ( );    /* Illegal */
```



C2141 value out of range for enum constant
An enumeration constant had a value outside the range of values allowed for type `int`.

C2142 ellipsis requires three periods
The compiler detected the token `"..."` and assumed that `"..."` was intended.

C2143 syntax error : missing `'token1'` before `'token2'`
The compiler expected `token1` to appear before `token2`. This message may appear if a required closing curly brace `}`, right parenthesis `)`, or semicolon `;` is missing.

C2144 syntax error : missing `'token'` before type `'type'`
The compiler expected the given token to appear before the given type name. This message may appear if a required closing curly brace `}`, right parenthesis `)`, or semicolon `;` is missing.

C2145 syntax error : missing `'token'` before
identifier
The compiler expected the given token to appear before an identifier. This message may appear if a semicolon `;` does not appear after the last declaration of a block.

C2146 syntax error : missing 'token' before identifier 'identifier'

The compiler expected the given token to appear before the given identifier.

C2147 array : unknown size

An attempt was made to increment an index or pointer to an array whose base type has not yet been declared.

C2148 array too large

An array exceeded the maximum legal size (2^{32} bytes).

C2149 *identifier* : named bit-field cannot have 0 width

The given named bit field had a zero width. Only unnamed bit fields are allowed to have zero width.

C2150 *identifier* : bit-field must have type int, signed int, or unsigned int

The ANSI C standard requires bit fields to have types of **int**, **signed int**, or **unsigned int**. This message appears only if you compiled your program with the **-Za** option.

C2151 more than one cdecl/fortran/pascal attribute specified

More than one keyword specifying a function-calling convention was given.

C2152 *identifier* : pointers to functions with different attributes

An attempt was made to assign a pointer to a function declared with one calling convention (**cdecl**, **fortran**, or **pascal**) to a pointer to a function declared with a different calling convention.

C2153 hex constants must have at least 1 hex digit

At least one hexadecimal digit must follow the "x". The hexadecimal constants 0x and 0X are illegal.

C2154 '*name*' : does not refer to a segment

The *name* was the first identifier given in an **alloc_text** pragma argument list and it is already defined as something other than a segment name.

C2155 '*name*' : already in a segment

The function *name* appears in more than one **alloc_text** pragma.

C Error Messages and Exit Codes

- C2156 pragma must be at outer level
Certain pragmas must be specified at a global level, outside a function body, and there is an occurrence of one of these pragmas within a function.
- C2157 '*name*' : must be declared before use in pragma list
The function *name* in the list of functions for an **alloc text** pragma has not been declared prior to being referenced in the list.
- C2158 '*name*' : is a function
Name was specified in the list of variables in a **same_segment** pragma, but was previously declared as a function.
- C2159 more than one storage class specified
Illegal declaration—only one storage class is allowed.
- C2160 ## cannot occur at the beginning of a macro definition
A macro definition cannot begin with a token-pasting (##) operator.
- C2161 ## cannot occur at the end of a macro definition
A macro definition cannot end with a token-pasting (##) operator.
- 2162 expected macro formal parameter
The token following a stringizing operator (#) must be a formal parameter name.
- 2163 '*string*' : not available as an intrinsic
A function specified in the list of functions for an intrinsic or function pragma is not one of the functions available in intrinsic form.
- C2165 '*keyword*' : cannot modify pointers to data
Bad use of **fortran**, **pascal** or **cdecl** keyword to modify pointer to data.
- C2167 '*name*' : too many actual parameters for intrinsic
A reference to the intrinsic function *name* contains too many actual parameters.



- C2168 *'name'* : too few actual parameters for
intrinsic
A reference to the intrinsic function *name* contains too few actual parameters.
- C2169 *'name'* : is an intrinsic, it cannot be defined
An attempt was made to provide a function definition for a function already declared as an intrinsic.
- C2170 *identifier* : *intrinsic not declared as a function*
You tried to use the **intrinsic** pragma for an item other than a function, or for a function that does not have an intrinsic form.
- C2177 constant too big
Information was lost because a constant value was too large to be represented in the type to which it was assigned. (1)
- C2171 *'operator'* : bad operand
Illegal operand type for the specified unary operator.

D.3.3 Warning Messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking. The number in parentheses at the end of each warning-message description gives the minimum warning level that must be set for the message to appear.

- C4000 UNKNOWN WARNING
The compiler detected an unknown error condition.
- C4001 macro *'identifier'* requires parameters
The given identifier was defined as a macro taking one or more arguments, but it was used in the program without arguments. (1)
- C4002 too many actual parameters for macro *'identifier'*
The number of actual arguments specified with the given identifier was greater than the number of formal parameters given in the macro definition of the identifier. (1)
- C4003 not enough actual parameters for macro
'identifier'
The number of actual arguments specified with the given identifier was less than the number of formal parameters given in the macro definition of the identifier. (1)

C Error Messages and Exit Codes

- C4004 missing close parenthesis after 'defined'
The closing parenthesis was missing from an **#if defined** phrase. (1)
- C4005 '*identifier*' : redefinition
The given identifier was redefined. (1)
- C4006 #undef expected an identifier
The name of the identifier whose definition was to be removed was not given with the **#undef** directive. (1)
- C4009 string too big, trailing chars truncated
A string exceeded the compiler limit on string size. To correct this problem, break the string into two or more strings. (1)
- C4011 identifier truncated to '*identifier*'
Only the identifier's first 31 characters are significant. (1)
- C4014 '*identifier*' : bit-field type must be unsigned
The given bit field was not declared as an **unsigned** type.

Bit fields must be declared as **unsigned** integral types. A conversion has been supplied. (1)
- C4015 '*identifier*' : bit-field type must be integral
The given bit field was not declared as an integral type.

Bit fields must be declared as **unsigned** integral types. A conversion has been supplied. (1)
- C4016 '*identifier*' : no function return type
The given function had not yet been declared or defined, so the return type was unknown.

The default return type (**int**) is assumed. (2)
- C4017 cast of int expression to far pointer
A **far** pointer represents a full segmented address. On an 8086/8088 processor, casting an **int** value to a **far** pointer may produce an address with a meaningless segment value. (1)
- C4020 too many actual parameters
The number of arguments specified in a function call was greater than the number of parameters specified in the argument-type list or function definition. (1)



C4021 too few actual parameters

The number of arguments specified in a function call was less than the number of parameters specified in the argument-type list or function definition. (1)

C4022 pointer mismatch : parameter *n*

The pointer type of the given parameter was different from the pointer type specified in the argument-type list or function definition. (1)

C4024 different types : parameter *n*

The type of the given parameter in a function call did not agree with the type given in the argument-type list or function definition. (1)

C4025 function declaration specified variable argument list

The argument-type list in a function declaration ended with a comma or a comma followed by ellipsis dots (,...), indicating that the function could take a variable number of arguments, but no formal parameters were declared for the function. (1)

C4026 function was declared with formal argument list

The function was declared to take arguments, but the function definition did not declare formal parameters. (1)

C4027 function was declared without formal argument list

The function was declared to take no arguments (the argument-type list consisted of the word **void**), but formal parameters were declared in the function definition or arguments were given in a call to the function. (1)

C4028 parameter *n* declaration different

The type of the given parameter did not agree with the corresponding type in the argument-type list or with the corresponding formal parameter. (1)

C4029 declared parameter list different from definition

The argument-type list given in a function declaration did not agree with the types of the formal parameters given in the function definition. (1)

C Error Messages and Exit Codes

C4030 first parameter list is longer than the second

A function was declared more than once with different argument-type lists in the declarations. (1)

C4031 second parameter list is longer than the first

A function was declared more than once with different argument-type lists. (1)

C4032 unnamed struct/union as parameter

The structure or union type being passed as an argument was not named, so the declaration of the formal parameter cannot use the name and must declare the type. (1)

C4033 function must return a value

A function is expected to return a value unless it is declared as **void**. (2)

C4034 sizeof returns 0

The **sizeof** operator was applied to an operand that yielded a size of zero. (1)

C4035 *identifier* : no return value

A function declared to return a value did not do so. (2)

C4036 unexpected formal parameter list

A formal parameter list was given in a function declaration. The formal parameter list is ignored. (1)

C4037 '*identifier*' : formal parameters ignored

No storage class or type name appeared before the declarators of formal parameters in a function declaration, as in the following example:

```
int *f(a,b,c);
```

The formal parameters are ignored. (1)

C4038 '*identifier*' : formal parameter has bad storage class

The given formal parameter was declared with a storage class other than **auto** or **register**. (1)

C4039 '*identifier*' : function used as an argument

A formal parameter to a function was declared to be a function, which is illegal. The formal parameter is converted to a function pointer. (1)



- C4040 near/far/huge on *'identifier'* ignored
The **near** or **far** keyword has no effect in the declaration of the given identifier and is ignored. (1)
- C4041 formal parameter *'identifier'* is redefined
The given formal parameter was redefined in the function body, making the corresponding actual argument unavailable in the function. (1)
- C4042 *'identifier'* : has bad storage class
The specified storage class cannot be used in this context (for example, function parameters cannot be given **extern** class). The default storage class for that context was used in place of the illegal class. (1)
- C4043 *'identifier'* : void type changed to int
An item other than a function was declared to have **void** type. (1)
- C4044 huge on *'identifier'* ignored, must be an array
The **huge** keyword was used to declare the given nonarray item. (1)
- C4045 *'identifier'* : array bounds overflow
Too many initializers were present for the given array. The excess initializers are ignored. (1)
- C4046 *'&'* on function/array, ignored
An attempt was made to apply the address-of operator (**&**) to a function or array identifier. (1)
- C4047 *'operator'* : different levels of indirection
An expression involving the specified operator had inconsistent levels of indirection. (1)

The following example illustrates this condition:

```
char **p;  
char *q;  
.  
.  
.  
p = q;
```

C Error Messages and Exit Codes

- C4048 array's declared subscripts different
An array was declared twice with different sizes. The larger size is used. (1)
- C4049 *'operator'* : indirection to different types
The indirection operator (*) was used in an expression to access values of different types. (1)
- C4051 data conversion
Two data items in an expression had different types, causing the type of one item to be converted. (2)
- C4052 different enum types
Two different **enum** types were used in an expression. (1)
- C4053 at least one void operand
An expression with type **void** was used as an operand. (1)
- C4056 overflow in constant arithmetic
The result of an operation exceeded 0x7FFFFFFF. (1)
- C4057 overflow in constant multiplication
The result of an operation exceeded 0x7FFFFFFF. (1)
- C4058 address of frame variable taken, DS != SS
The program was compiled with the default data segment (**DS**) not equal to the stack segment (**SS**), and the program tried to point to a frame variable with a near pointer. (1)
- C4059 segment lost in conversion
The conversion of a **far** pointer (a full segmented address) to a **near** pointer (a segment offset) resulted in the loss of the segment address. (1)
- C4060 conversion of long address to short address
The conversion of a long address (a 32-bit pointer) to a short address (a 16-bit pointer) resulted in the loss of the segment address. (1)
- C4061 long/short mismatch in argument:
conversion supplied
The base types of the actual and formal arguments of a function were different. The actual argument is converted to the type of the formal parameter. (1)
- C4062 near/far mismatch in argument: conversion supplied



The pointer sizes of the actual and formal arguments of a function were different. The actual argument is converted to the type of the formal parameter. (1)

C4063 *'identifier'* : function too large for
post-optimizer

The given function was not optimized because not enough space was available. To correct this problem, reduce the size of the function by dividing it into two or more smaller functions. (0)

C4064 procedure too large, skipping *description* optimization and continuing

Some optimizations for a function were skipped because not enough space was available for optimization. (0)

To correct this problem, reduce the size of the function by dividing it into two or more smaller functions.

The *description* in this message may appear as any of the following:

```
loop inversion
branch sequence
cross jump
```

C4065 recoverable heap overflow in post-optimizer
- some optimizations may be missed

Some optimizations were skipped because not enough space was available for optimization. To correct this problem, reduce the size of the function by dividing it into two or more smaller functions. (0)

C4066 local symbol table overflow - some local symbols may be missing in listings

The listing generator ran out of heap space for local variables, so the source listing may not contain symbol-table information for all local variables.

C4067 unexpected characters following *'directive'*
directive - newline expected

Extra characters followed a preprocessor directive, as in the following example (1):

```
#endif    NO_EXT_KEYS
```

This is accepted in Version 3.0, but not in Versions 4.0 and 5.0. Versions 4.0 and 5.0 require comment delimiters, such as the following:

```
#endif /* NO_EXT_KEYS */
```

C4068 unknown pragma

The compiler did not recognize a pragma and ignored it. (1)

C4069 conversion of near pointer to long integer

A near pointer was converted to a long integer, which involves first extending the high-order word with the current data-segment value, *not* 0 as in Version 3.0. (1)

C4071 '*identifier*' : no function prototype given

The given function was called before the compiler found the corresponding function prototype. (3)

C4072 Insufficient memory to process debugging information

You compiled the program with the **-Zi** option, but not enough memory was available to create the required debugging information. (1)

C4073 scoping too deep, deepest scoping merged when debugging

Declarations appeared at a static nesting level greater than 13. As a result, all declarations will seem to appear at the same level. (1)

C4074 non standard extension used - '*extension*'

The given nonstandard language extension was used when the **-Ze** option was in effect. These extensions are given in the "Compiling with the cc Command" chapter of this guide. (If the **-Za** option is in effect, this condition generates an error.) (3)

C4075 size of switch expression or case constant too large - converted to int

A value appearing in a **switch** or **case** statement was larger than an **int**. The compiler converts the illegal value to an **int**. (1)

C4076 '*type*' : may be used on integral types only

The type modifiers **signed** and **unsigned** can be combined only with other integral types.



- C4077 unknown `check_stack` option
Unknown option given when using the old form of the **check_stack** pragma. The option must be empty, +, or -.
- C4079 unexpected char '*character*'
Unexpected separator *character* found in argument list of a pragma.
- C4080 missing segment name
The first argument in the argument list for the **alloc_text** pragma is missing a segment name. This happens if the first token in the argument list is not an identifier.
- C4081 expected a comma
There is a missing comma (,) between two arguments of a pragma.
- C4082 expected an identifier
There is a missing identifier in list of arguments to a pragma.
- C4083 missing '('
There is a missing opening parenthesis (() in the argument list for a pragma.
- C4084 expected a pragma keyword
The token following the **pragma** keyword is not an identifier.
- C4085 expected [onloff]
Bad argument given for new form of **check_stack** pragma.
- C4086 expected [1|2|4]
Bad argument given for **pack** pragma.
- C4087 '*name*' : declared with *void parameter list*
The function *name* was declared as taking no parameters, but a call to the function specifies actual parameters.
- C4090 different '*const*' attributes
The program passed a pointer to a *const* item to a function where the corresponding formal parameter is a pointer to a non-*const* item, which means the item could be modified by the function undetected.
- C4091 no symbols were declared
An empty declaration was detected. (2)

C Error Messages and Exit Codes

C4092 untagged enum/struct/union declared
 no symbols

An empty declaration was detected that used an untagged enum/struct/union. (2)

C4093 unescaped newline in character constant in non-active code

The constant expression of an **#if**, **#elif**, **#ifdef**, or **#ifndef** preprocessor directive evaluated to 0, making the following code inactive, and a new-line character appeared between a single or double quotation mark and the matching single or double quotation mark in that inactive code.

C4094 unexpected newline

A new-line character appeared in a pragma where a comma, right parenthesis, or identifier was expected, as in the following examples:

```
#pragma intrinsic (memset  
#pragma intrinsic (memset,
```

C4095 too many arguments for pragma

More than one argument was given for a pragma that can take only one argument.

D.3.4 Compiler Limits

To operate the XENIX C Compiler, you must have sufficient disk space available for the compiler to create temporary files used in processing. The space required is approximately two times the size of the source file.

Table D.1 summarizes the limits imposed by the C compiler. If your program exceeds one of these limits, an error message will inform you of the problem.



Table D.1
Limits Imposed by the C Compiler

Program Item	Description	Limit
String literals	Maximum length of a string, including the terminating null character (\0)	512 bytes
Constants	Maximum size of a constant is determined by its type; see the <i>XENIX C Language Reference</i> for a discussion of constants	
Identifiers	Maximum length of an identifier	31 bytes (additional characters are discarded)
Declarations	Maximum level of nesting for structure/union definitions	10 levels
Preprocessor directives	Maximum size of a macro definition	512 bytes
	Maximum number of actual arguments to a macro definition	8 arguments
	Maximum length of an actual preprocessor argument	256 bytes
	Maximum level of nesting for #if , #ifdef , and #ifndef directives	32 levels
	Maximum level of nesting for include files	10 levels

The compiler does not set explicit limits on the number and complexity of declarations, definitions, and statements in an individual function or in a program. If the compiler encounters a function or program that is too large or too complex to be processed, it produces an error message to that effect.

D.4 Compiler Exit Codes

All the programs in the XENIX C Compiler package return an exit code (sometimes called an “errorlevel” code) that can be used by other programs such as **make**. If the program finishes without errors, it returns a code of 0. The code returned varies depending on the error encountered.

Code Meaning

- | | |
|---|---|
| 0 | No fatal error |
| 2 | Program error (such as compiler error) |
| 4 | System level error (such as out of disk space or compiler internal error) |



Replace this Page
with Tab Marked:

Index



Index

- { } (braces) 1-5
- [] (brackets) 1-5
- | (bar) 1-5
- (dash) option character
 - linker 3-3
- (hyphen) option character, cc 2-4
- _ (underscore), in names 2-10, 2-21

A

- Address space B-8
- Addresses
 - components 5-3
 - far 5-4
 - huge 5-4
 - near 5-4
- Alias checking 2-38
- Alignment *See* Storage alignment
- alloc_text pragma 5-30
- argc variable 4-2
- Arguments
 - linker options 3-4
 - listing options 2-11
 - macros D-41
 - main function *See* main function
 - variable number of 2-55, 6-5, B-12
- Argument-type list 2-35
- argv variable 4-2
- Assembly language
 - interface 8-7
 - return values 8-5
 - routines
 - entry 8-4
 - exit 8-7
- Assembly language interface, described 8-1
- Assembly-listing files
 - creating 2-9, 2-10
 - extensions 2-11
 - format 2-20

B

- Bar (|) 1-5
- BASE 7-9
- Bibliography 1-8
- Bit fields B-5
- Bold font 1-5
- Braces ({}) 1-5
- Brackets ([]) 1-5
- Byte length B-2
- Byte order B-16, B-4

C

- C calling conventions
 - described 8-1
- C compiler
 - compiler pass options 2-58
 - d option 2-58
 - DOS Cross Development Options 2-57
 - dos option 2-57
 - FP option 2-57
 - functions of the passes 2-58
 - impure small model 5-6
 - M option 2-4
 - manifest defines 2-26
 - model and segment options 2-4
 - pure small model 5-7
 - z option 2-58
- C language
 - calling sequence 8-4
 - interface with assembly language 8-7
 - return values 8-5
 - c option 2-7
 - C option 2-29
- Call sequence 8-1
- Calling conventions
 - C 2-55, 6-5
 - controlling
 - cdecl keyword 2-56
 - fortran and pascal keywords 2-56
 - Gc option 2-56
 - FORTRAN/Pascal 2-55, 6-5
- Calling sequence
 - assembly language 8-4
 - C language 8-4
- Canonic Frame 7-6

Index

- Capital letters
 - small 1-5
 - use of 1-5
- Case significance
 - linker 3-5
- cc command
 - exit codes D-41
 - file processing 2-2
 - format 2-2
- cc options
 - A 5-22, 5-24, 5-25
 - assembly listing 2-9, 2-10
 - c 2-7
 - C 2-29
 - command line, order 2-4
 - comments, preserving 2-29
 - constants and macros, defining 2-24
 - D 2-24
 - data segments, naming 5-28, 5-30, 6-7
 - data threshold, setting 5-27
 - default char type, changing 2-55
 - default libraries 2-5
 - differences from linker options 3-4
 - E 2-28
 - EP 2-28
 - external names, restricting length of 2-53
 - F 3-6
 - Fa 2-10, 2-20
 - Fc 2-10
 - Fe 2-9
 - FI 2-10
 - Fm 2-10
 - Fo 2-7
 - format 2-4
 - FORTRAN/Pascal, calling convention 2-56
 - Fs 2-10
 - function declarations, generating 2-35
 - Gc 2-56
 - Gs 2-47, 6-4
 - Gt 5-27
 - help 2-6
 - I 2-30
 - include files, searching for 2-30
 - J 2-55
 - language extensions, disabling 2-49
 - line numbers 2-36
 - link 2-2, 3-1
 - linker information, passing 3-1
 - listing 2-6
 - maximum optimization 2-38
 - Mc 5-7
 - memory models
 - code-pointer size 5-24
 - compact 5-7
 - cc options (*continued*)
 - memory models (*continued*)
 - data-pointer size 5-24
 - default libraries 2-4
 - huge 5-9
 - large 5-9
 - medium 5-7
 - mixed 5-22, 5-24, 5-25
 - segments, setting up 5-25
 - small 5-6
 - Mh 5-9
 - MI 5-9
 - Mm 5-7
 - Ms 5-6
 - naming
 - executable files 2-9
 - modules 5-28
 - object files 2-7
 - ND 5-28, 5-30, 6-7
 - nl 2-53
 - NM 5-28
 - NT 5-28
 - o 2-9
 - Oa 2-38, 6-3
 - object files
 - labeling 2-53
 - naming 2-7
 - specifying 2-2
 - object listing 2-9, 2-10
 - Od 2-36, 2-42
 - Oi 2-38, 2-42, 6-3
 - Ol 2-38, 2-44, 6-4
 - Op 2-45
 - optimization
 - alias checking, relaxing 2-38, 6-3
 - code size 2-38, 2-46
 - disabling 2-36, 2-42
 - execution time 2-38, 2-46, 6-3
 - floating-point results, consistent 2-45
 - intrinsic functions 2-38, 2-42, 6-3
 - loops 2-38, 2-44, 6-4
 - maximum 2-46
 - Oi 6-3
 - program speed 6-2
 - option character
 - hyphen (-) 2-4
 - Os 2-46
 - Ot 2-46, 6-3
 - Ox 2-38, 2-46
 - P 2-28
 - predefined identifiers, removing definitions 2-27
 - preprocessed listing 2-28
 - preprocessor

- cc options (*continued*)
 - preprocessor (*continued*)
 - C 2-29
 - D 2-24
 - U and -u 2-27
 - source files, specifying 2-2, 2-6
 - source listing 2-10
 - source/object listing 2-10
 - special keywords, disabling 5-14
 - Ss 2-13
 - St 2-13
 - stack probes, removing 2-47, 6-4
 - standard places, ignoring 2-30
 - structure members, packing 2-51
 - subtitle 2-13
 - suppressing
 - library selection 2-54
 - linking 2-7
 - syntax checking 2-34
 - Tc 2-3, 2-6
 - text segments, naming 5-28
 - titles 2-13
 - U and -u 2-27
 - V 2-53
 - Version 4.0, new for A-8
 - Version 5.0, new for A-3
 - W0, -W1, -W2, and -W3 2-33
 - warning level 2-33
 - X 2-30
 - Za 2-49, 5-14
 - Zd 3-6
 - Zg 2-35
 - Zi 2-36
 - Zl 2-54
 - Zp 2-51
 - Zs 2-34
- cdecl keyword
 - defined 2-56
 - Gc option, used with 6-6
 - include files, used in 2-50
 - Za option, used with 2-49
- char type, changing default 2-55
- Character
 - classification, macros B-8
 - set B-8
 - types
 - signed B-9
 - unsigned B-9
- check_stack pragma 2-47, 6-4
- Class name, LSEG 7-6
- Code pointers, mixed memory models 5-24
- Code size, optimization 2-38, 2-46
- Combination Attribute 7-21
- Command line (*continued*)
 - arguments
 - executable file 4-1
 - cc 2-2
 - error messages D-1
 - length, maximum 2-2
- Commands
 - notational conventions 1-5
- COMMENT 7-40
- RECORD 7-40
- Comments, preserving 2-29
- Compact memory models *See* Memory models
- Compilation
 - conditional 2-50
 - error messages D-6
- Compiler
 - differences, other compilers
 - portability problems B-9
 - differences, Version 4.0
 - cc options A-8
 - enhancements and additions A-5
 - language changes A-5
 - differences, Version 5.0
 - enhancements and additions A-1
 - language changes A-2
 - new cc options A-3
 - pragmas, new A-4
 - documentation 1-1
 - error messages *See* Error messages, compiler
 - limits D-40
 - naming conventions 2-21
 - stopping 2-2
- Compiler, converting from previous versions
 - See* Compiler differences
- Compiler guide, organization 1-2
- Compiler options *See* cc options
- Complete name, LSEG 7-6
- Conditional compilation 2-24, 2-50
- Constants
 - defining 2-24
 - manifest *See* Constants, symbolic
 - size, maximum D-40
 - symbolic 2-24
- Controlling
 - linker 3-3
 - preprocessor 2-27
 - segments 3-5
 - stack size 3-6
- Conventions, notational 1-5
- Conversion
 - near pointers to long integers A-7
 - pointer arguments 5-20
- Correctable error messages D-6
- cr0.o *See* Start-up routine

Index

ctype.h macros B-8
Customized memory models *See* Mixed memory models

D

-D option 2-24
d option
 cc 2-58
Dash (-)
 linker option character 3-3
Data
 passing to programs 4-1
 portability B-14
 segment
 data threshold, setting 5-27
 default, contents 5-27
 default name 5-28
 mixed memory models 5-25
 naming 5-28
 types, size of B-2
Data pointers, mixed memory models 5-24
_DATA segment 5-28
Data Structures
 x.out symbol table 7-50
Data threshold, setting 5-27
Debugging, preparing for
 -Zi and -Od options 2-36
Declarations, maximum level of nesting D-40
Default libraries
 object files, used in 3-2
 suppressing selection 2-54
DGROUP group 5-28
Differences from previous versions *See*
 Compiler differences
Directory names, notational conventions 1-5
Documentation, compiler 1-1
DOS Cross Development
 C compiler 2-57
dos option
 cc 2-57
dosld command 2-57
DS register 5-25

E

-E option 2-28
eax register 8-5, 8-7
ebp register 8-4, 8-7

ebx register 8-7
ecx register 8-7
edi register 8-4, 8-7
edx register 8-5, 8-7
EIGHT
 LEAF
 DESCRIPTOR 7-25
EIGHT LEAF DESCRIPTOR 7-24
Ellipses, use of 1-5
environ variable 4-3
Environment
 portability problems B-13
 table
 pointer to 4-2
 variable names, notational conventions 1-variables
 INCLUDE 2-30
 LIB 3-2
 PATH 4-1
 SET 4-1
envp variable 4-2
-EP option 2-28
errno variable
 defined 9-2
 described 9-2
Error messages
 compiler
 command line D-1
 compilation D-6
 correctable D-6
 fatal D-6, D-7
 identifying 2-31
 redirecting 2-31
 warning D-30, D-6
 format *See* Error messages, compiler
 source listings 2-14
 warning messages, setting level of 2-33
Errorlevel codes *See* Exit codes
Errors
 catching signals 9-3
 delayed 9-4
 errno variable 9-2
 error constants 9-2
 error numbers 9-2
 printing error messages 9-2
 processing 9-1
 routine system I/O 9-4
 sharing resources 9-4
 signals 9-3
 standard error file 9-1
 system 9-4
esi register 8-4, 8-7
esp register 8-4
Evaluation order B-12

exec function 4-1
 Executable files
 cc command and 2-3
 command-line arguments 4-1
 extensions 2-9
 naming, default 2-9
 naming with cc 2-9
 passing data to 4-1
 running 4-1
 Executable Format 7-50
 Execution-time optimization 2-38, 2-46, 6-3
 Exit code D-41
 Extensions
 executable files 2-9
 listing files, defaults for 2-10
 map files 2-11
 object files 2-8
 object-listing files 2-11
 source-listing files 2-11
 source/object-listing files 2-11
 External names 2-53

F

-F option 3-6
 -Fa option 2-10, 2-20
 Far keyword 5-18
 far keyword
 default addressing conventions 5-12
 effects
 data declarations 5-14, 6-6
 function declarations 5-18
 library routines, used with 5-14
 small-model programs, used in 5-6
 -Za option, used with 2-49
 Far pointers 5-12
 Fatal-error messages D-6, D-7
 -Fc option 2-10
 -Fe option 2-9
 File names
 notational conventions 1-5
 Files
 assembly listing 2-10, 2-20
 executable *See* Executable files
 listing, preprocessed 2-28
 map
 creating 2-10, 2-13, 3-5, 3-6
 default names 2-11
 listing formats 2-22
 -MAP linker option 3-5
 object
 cc command, used with 2-2, 2-3

Files (*continued*)
 object (*continued*)
 listing 2-10, 2-11, 2-19
 source 2-2
 source listing *See* Source-listing files
 source/object listing *See* Source/object-listing files
 temporary
 space requirements D-40
 FIXUP
 RECORD 7-34
 FIXUPP 7-34
 Fixups
 definition 7-8
 segment.....relative 7-10, 7-14
 self.....relative 7-10, 7-13
 -Fl option 2-10
 Floating point
 operations
 optimizing for consistency in 2-45
 -Fm option 2-10
 -Fo option 2-7
 fortran keyword 2-49, 2-56, 6-6
 FP option
 cc 2-57
 FRAME
 definition 7-4
 specifying 7-11
 FRAME NUMBER 7-5
 -Fs option 2-10
 function pragma 2-42
 Functions
 arguments, variable number of 2-55, 6-5, B-12
 calling conventions
 C 2-55, 6-5
 FORTRAN/Pascal 2-55, 6-5
 declarations
 generating 2-35
 near and far keywords 5-18

G

-Gc option 2-56
 getenv function 4-2
 Global symbols *See* Public symbols
 GROUP 7-5
 Group Definition Record 7-23
 GRPDEF 7-23
 -Gs option 2-47, 6-4
 -Gt option 5-27

Index

H

Hardware Reference Numbers 7-56
-help option
 cc 2-6
HIBYTE 7-9
Huge arrays 5-9
huge keyword 2-49
 data declarations, effects in 5-14, 6-6
 default addressing conventions 5-12
 library routines, used with 5-14
 small-model programs, used in 5-6
Huge memory model *See* Memory models
Huge pointers 5-12
Hyphen (-), cc option character 2-4

I

-I option 2-30
iAPX.....286,386
 address translation
 logical to physical 7-2
 descriptor tables 7-1
 GDT 7-1
 LDT 7-1
 logical address space 7-1
 memory management 7-1
 pointers
 to logical addresses 7-1
 protected mode 7-1
 segment selector 7-2
 INDEX field 7-2
 RPL field 7-2
 TI field 7-2
 system architecture 7-1
Identifier length *See* Names, length
Identifiers
 length, maximum D-40
 predefined
 listed 2-26
 M_I86 2-26
 M_I86xM 2-26
 M_XENIX 2-26
 removing definitions of 2-27
Implicit bss 7-49
Include files
 directory specification 2-30
 nesting, maximum level of D-41
 portability problems B-2
 search path 2-30
INCLUDE variable

INCLUDE variable (*continued*)
 overriding 2-30
Index fields 7-8
Indices 7-8
intrinsic pragma 2-42
Italics 1-5
Iterated Segments 7-48

J

-J option 2-55

K

Key sequences, notational conventions 1-:
Keywords
 cdecl 2-49, 2-56, 6-6
 far 5-18
 far *See* far keyword
 fortran 2-49, 6-6
 huge *See* huge keyword
 near 5-18
 near *See* near keyword
 pascal 2-49, 6-6
 special 2-49
 Version 4.0, new for A-8

L

Language extensions
 disabling 2-49
 listed 2-49
Large memory model *See* Memory models
 large
Large Model 7-49
LIB variable 3-2
Libraries
 creating
 -Zl, compiling modules with 2-54
 default
 ignoring 3-3
 -M options 2-5, 3-1
 overriding 3-3
 suppressing selection 2-54
 mixed-model programs 5-26
 names in object files 3-1
 search

Libraries (*continued*)
 search (*continued*)
 path 3-2, 3-3
 specifying 3-2
 standard places 3-2
 Library
 routines
 exec 4-1
 getenv 4-2
 intrinsic forms 2-43
 putenv 4-2
 system 4-1
 XENIX dependent C-1
 LIDATA 7-32
 Limits
 compiler D-40
 LINE
 NUMBERS
 RECORD 7-30
 -LINENUMBERS (-LI) linker option 3-6
 -link option 2-2, 3-1
 Linker
 error messages 2-31
 Linker options
 abbreviations 3-4
 case sensitivity 3-5
 cc options, differences from 3-4
 command line, order on 3-5
 default libraries, ignoring 3-3
 line numbers, displaying 3-6
 -LINENUMBERS (-LI) 3-6
 map file 3-5
 -MAP (-M) 3-5
 -NODEFAULTLIBRARYSEARCH (-NOD)
 overriding default libraries 3-3
 numerical arguments 3-4
 rules 3-3
 segments
 number of 3-5
 -SEGMENTS (-SE) 3-5
 stack size, setting 2-52, 3-6
 -STACK (-ST) 3-6
 -T 6-7
 translating far calls 6-7
 LINNUM 7-30
 List of Names Record 7-19
 Listing cc options 2-6
 Listing files
 assembly 2-9, 2-10, 2-20
 map 2-10
 object 2-9, 2-10, 2-19
 preprocessed 2-28
 source 2-9, 2-10, 2-14
 source/object 2-10, 2-20

LNames 7-19
 LOBYTE 7-10
 LOCATION, types 7-9
 LOGICAL
 ITERATED
 DATA
 RECORD 7-32
 Logical Segment 7-5
 Long pointers *See* Far pointers
 Loop optimization 2-44, 6-4
 loop_opt pragma 2-37, 2-44, 6-4
 LSEG 7-5

M

M option
 cc 2-4
 Macro definitions D-40
 Macros
 arguments, maximum number D-41
 character classification B-8
 defined 2-24
 notational conventions 1-5
 main function
 arguments to 4-1
 Manifest constants, notational conventions 1-5
 Manifest defines
 C compiler 2-26
 Map files
 creating 2-10, 2-13, 3-5
 extensions 2-11, 3-6
 -Fm option 2-13
 format 2-22
 -MAP linker option 3-5
 program entry point 2-23
 segment lists 2-22
 symbol tables 2-22
 -MAP linker option 3-5
 MAS 7-4
 -Mc option 5-7
 Medium memory model *See* Memory models
 Memory Address Space 7-4
 Memory addresses *See* Addresses
 Memory models
 compact 5-7
 default 5-2, 5-6
 huge 5-9
 large 5-9
 medium 5-7
 mixed *See* Mixed memory models
 options
 code-pointer size 5-24

Index

Memory models (*continued*)

- options (*continued*)
 - compact model 5-7
 - data-pointer size 5-24
 - default libraries 2-5
 - huge model 5-9
 - large model 5-9
 - medium model 5-7
 - segment setup 5-25
 - small model 5-6
- small 5-2, 5-6, 5-18
- standard
 - advantages 5-5
 - common features 5-5
 - disadvantages 5-5
 - Version 4.0, new for A-8
- Memory models, customized *See* Mixed memory models
- Mh option 5-9
- M_I86 identifier 2-26
- M_I86xM identifier 2-26
- Mixed memory models
 - code pointers 5-24
 - creating 5-22
 - data pointers 5-24
 - library support 5-26
 - near, far, huge keywords 5-12
 - segment setup options 5-25
- Ml option 5-9
- Mm option 5-7
- MODE 7-10
- MODEND 7-39
- MODULE 7-4
- END
- RECORD 7-39
- Module header record 7-6
- Modules, naming 5-28
- Ms option 5-6
- M_XENIX identifier 2-26

N

Names

- conventions 2-57
- executable files 2-9
- external 2-53
- global 2-10, 2-21
- length B-10
- modules, changing 5-28
- object files 2-7
- segments, changing 5-28
- underscores (`_`), using in 2-10, 2-21

Naming conventions

- compiler 2-21
- segments 5-29
- ND option 5-28, 5-30, 6-7
- Near keyword 5-18
- near keyword
 - data declarations, effects in 5-14, 6-6
 - default addressing conventions 5-12
 - function declarations, effects in 5-18
 - library routines, used with 5-14
- Near pointer 5-12
- Nesting
 - declarations D-40
 - include files D-41
 - preprocessor directives D-41
- nl option 2-53
- NM option 5-28
- NODEFAULTLIBRARYSEARCH (-NOD)
 - linker option
 - default libraries, overriding 3-3
- Non-Iterated Segments 7-49
- Notational conventions 1-5
- NT option 5-28
- Numeric record types 7-42

O

- O (optimization) options 2-37
- o option 2-9
- Oa option, cc 2-38, 6-3
- Object File Format
 - Executable 7-45
- Object files
 - cc command 2-2, 2-3
 - default extension 2-2, 2-6
 - extensions 2-8
 - labeling 2-53
 - library names in 3-1
 - naming 2-7
 - specifying to cc 2-2
- Object listing *See* Object-listing files
- Object Module Formats 7-3, 7-4
- Object-listing files
 - creating 2-10
 - extensions 2-11
 - format 2-19
- Od option 2-36
- OFFSET 7-9
- Oi option 2-38, 6-3
- Ol option 2-38, 2-44, 6-4
- OMF 7-4
- omf Subset 7-46

- Op option 2-45
- Optimization
 - alias checking, relaxing 2-38, 6-3
 - code size 2-38, 2-46
 - consistent floating-point results 2-38, 2-45
 - default 2-1, 2-46
 - disabling 2-36, 2-38, 2-42
 - execution time 2-38, 6-3
 - intrinsic functions 2-42
 - intrinsic pragmas 6-3
 - listing files 2-12
 - loops 2-44, 6-4
 - maximum 2-38, 2-46
 - options 2-37
 - stack probes, removing 2-47, 6-4
- Optimizing *See* Optimization
- Optional fields, notational conventions 1-5
- Options, cc *See* cc options
- Options, linker *See* Linker options
- Os option 2-46
- Ot option 2-46, 6-3
- Overlay Name, LSEG 7-6
- Overview 1-1
- Ox option 2-38, 2-46

P

- P option 2-28
- pack pragma 2-51
- Packing
 - structure members 2-51
- PARAGRAPH NUMBER 7-5
- pascal keyword 2-49, 2-56, 6-6
- Path names
 - notational conventions 1-5
 - portability problems B-2
- PATH variable 4-1
- perfor function 9-2
- Physical Segment 7-5
- Placeholders 1-5
- Pointers
 - arguments, size conversion 5-20
 - code 5-24
 - far 5-12, 5-24
 - huge 5-12
 - manipulation B-6
 - near
 - conversion to long integers A-7
 - customized memory models 5-24
 - near keywords, used with 5-12
 - subtracting in huge-model programs 5-9
- Portability

- Portability (*continued*)
 - address space B-8
 - bit fields B-5
 - byte length B-2
 - byte order B-16, B-4
 - case distinctions B-10
 - character set B-8
 - data B-14
 - data types, size of B-2
 - environment differences B-13
 - evaluation order B-12
 - functions with variable number of arguments
 - B-12
 - guidelines B-2
 - hardware B-2
 - identifier length B-10
 - include files B-2
 - path names B-2
 - pointer manipulation B-6
 - register variables B-10
 - shift operations B-9
 - side effects B-12
 - sign extension B-9
 - signed and unsigned char types B-9
 - storage alignment B-3
 - type conversion B-10
 - word length B-2
- Pragmas
 - alloc_text 5-30
 - check_stack 2-47, 6-4
 - function 2-42
 - intrinsic 2-42
 - loop_opt 2-37, 2-44, 6-4
 - pack 2-51
 - same_seg 5-30, 6-7
 - Version 4.0, new for A-8
 - Version 5.0, new for A-4
- Preprocessor
 - macro arguments, maximum number of D-41
 - macro definitions, maximum size of D-40
 - nesting, maximum level of D-41
 - options
 - comments, preserving 2-29
 - D 2-24
 - predefined identifiers, removing definitions
 - of 2-27
 - use 2-24
- Product names, notational conventions 1-5
- Prompts 1-5
- PSEG
 - definition 7-5
 - NUMBER 7-5
- PUBDEF 7-26
- PUBLIC

Index

PUBLIC (*continued*)

NAMES

DEFINITION

RECORD 7-26

Public names *See* External names

Public names *See* Public symbols

Public symbols, listing 2-13, 3-5

putenv function 4-2

Q

Quotation marks, use of 1-5

R

Record format, sample 7-16

Record formats 7-3

Record order 7-15

Record types 7-43

numeric 7-42

Register variables 6-1, B-10

Registers

eax 8-5, 8-7

ebp 8-4, 8-7

ebx 8-7

ecx 8-7

edi 8-4, 8-7

edx 8-5, 8-7

esi 8-4, 8-7

esp 8-4

Relocatable memory images 7-3

Return codes *See* Exit Codes

Return values 8-2

assembly language 8-5

Routine entry sequence 8-2

Routine exit sequence 8-2

Routines

assembly language

entry 8-4

exit 8-7

Run file *See* Executable file

S

same_seg pragma 5-30, 6-7

Sample x.out File 7-48

Search paths

Search paths (*continued*)

changing

include files 2-30

libraries 3-3

include files 2-30

libraries 3-2

SEGDEF 7-19

Segment addressing 7-7

Segment definition 7-7

Segment definition record 7-19

Segment lists

map files 2-22

source listings 2-19

Segment Name, LSEG 7-6

Segment Numbers 7-56

Segment registers 8-7

SegmentRelative fixups 7-10

SegmentRelative Fixups 7-14

Segments

data

default name 5-28

mixed memory models 5-25

names 5-28

naming 5-28

threshold, effect of 5-27

default 5-3

defined 5-3

names, changing 5-28

naming conventions 5-29

number allowed 3-5

setting up 5-25

source listing 2-19

stack 5-25

text

default name 5-28

naming 5-28

-SEGMENTS (-SE) linker option 3-5

SelfRelative fixups 7-10, 7-13

SET variable 4-1

Shift operations B-9

Short pointers *See* Near pointers

Side effects B-12

Sign extension B-9

Signals

catching 9-3

on program errors 9-3

Signed char type B-9

sizeof operator 5-9

Small capitals, use of 1-5

Small memory model *See* Memory models

Small model 5-18

impure 5-6

pure 5-7

Source files

Source files (*continued*)
 default extension 2-2, 2-6
 specifying to cc 2-2
 Source listing *See* Source-listing files
 Source-listing files
 creating 2-10
 described 2-9
 error messages 2-14
 extensions 2-11
 format 2-14, 2-15
 segment lists 2-19
 subtitles 2-13
 symbol tables 2-17
 titles 2-13
 Source/object-listing files
 creating 2-10
 extensions 2-11
 format 2-20
 Special Header Fields 7-49
 Special keywords, disabling 5-14
 -Ss option 2-13
 SS register 5-25
 -St option 2-13
 Stack
 probes 2-47, 6-4
 segments, mixed memory models 5-25
 size
 default for C programs 2-52
 setting 2-52, 3-6
 Stack order 8-1
 -STACK (-ST) linker option 3-6
 Standard files
 redirecting 9-1
 Standard places
 changing 2-30
 ignoring 2-30
 libraries 3-2
 stderr, the standard error file 9-1
 Storage alignment B-3
 Strings
 length, maximum D-40
 notational conventions 1-5
 Structures, packing 2-51
 Subtitles, source listings 2-13
 Switches *See* Options
 Symbol definition 7-8
 Symbol Table 7-50
 Symbol tables
 map files, used in 2-22
 object files, used in (-Zi option) 2-36
 source listings, used in 2-17
 Syntax conventions *See* Notational conventions
 sys_erro array, described 9-3
 System errors

System errors (*continued*)
 described 9-4
 reporting 9-4
 system function 4-1

T

-T linker option 6-7
 TARGET 7-10
 -Tc option 2-3, 2-6
 _TEXT segment 5-28
 Text segments
 default name 5-28
 naming 5-28
 THEADR 7-18
 Titles, source listings 2-13
 T.....MODULE 7-4
 T.....Module Header Record (THEADR)
 7-18
 TYPDEF 7-24
 Types
 checking 2-35
 conversion B-10

U

-U and -u options 2-27
 Underscore (_) in names 2-10, 2-21
 Unsigned char type B-9
 Uppercase letters, use of 1-5

V

-V option 2-53
 Variables, register *See* Register variables
 Vertical bar (|) 1-5

W

-W0, -W1, -W2, and -W3 options 2-33
 Warning error messages 2-33, D-30, D-6
 Wild card
 characters 2-7

Index

X

- X option 2-30
- x.out
 - file layout 7-48
 - general description 7-46
 - implicit bss 7-49
 - iterated segments 7-48
 - large model 7-49
 - noniterated segments 7-49
 - special fields 7-49
 - symbol table 7-50
- x.out Examples 7-52
- x.out Executable Format 7-50
- x.out Format 7-45
- x.out Include Files 7-52
- x.out Segmented OMF Specification 7-45

Z

- z option
 - cc 2-58
- Za option 2-49, 5-14
- Zd option 3-6
- Zg option 2-35
- Zi option 2-36
- Zl option 2-54
- Zp option 2-51
- Zs option 2-34

Replace this Page
with Tab Marked:

**C LIBRARY
GUIDE**



XENIX[®] System V

Development System

C Library 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.

Contents

1 Introduction

- 1.1 About the CLibrary 1-1
- 1.2 About This Manual 1-1
- 1.3 Notational Conventions 1-4

2 Using C Library Routines

- 2.1 Introduction 2-1
- 2.2 Identifying Functions and Macros 2-1
- 2.3 Including Files 2-3
- 2.4 Declaring Functions 2-4
- 2.5 Stack Checking on Entry 2-5
- 2.6 Argument-Type Checking 2-5
- 2.7 Error Handling 2-6
- 2.8 Filenames and Pathnames 2-7
- 2.9 Floating-Point Support 2-8
- 2.10 Using Huge Arrays with Library Functions 2-9

3 Global Variables and Standard Types

- 3.1 Introduction 3-1
- 3.2 The daylight, timezone, and tzname Variables 3-1
- 3.3 errno, sys_errlist, sys_nerr 3-2
- 3.4 environ 3-2
- 3.5 Standard Types 3-2

4 Run-Time Routines by Category

- 4.1 Introduction 4-1
- 4.2 Buffer Manipulation 4-1
- 4.3 Character Classification and Conversion 4-2
- 4.4 Database-Manipulation Routines 4-3
- 4.5 Data Conversion 4-3
- 4.6 Directory Operation 4-4
- 4.7 File Handling 4-5
- 4.8 Group and Password File Control Routines 4-6
- 4.9 Input and Output Routines 4-7
- 4.10 Math 4-11
- 4.11 Memory Allocation 4-13
- 4.12 Message-Control Routines 4-14
- 4.13 Pipes 4-14
- 4.14 Process Control 4-16

- 4.15 Random-Number Generation 4-18
- 4.16 Screen Processing 4-18
- 4.17 Searching and Sorting 4-22
- 4.18 Semaphore-Control Routines 4-23
- 4.19 Shared-Memory Routines 4-23
- 4.20 String Manipulation 4-24
- 4.21 System Accounting 4-26
- 4.22 Terminal Control 4-27
- 4.23 Time 4-27
- 4.24 Miscellaneous 4-28

5 Include Files

- 5.1 Introduction 5-1
- 5.2 /usr/include Files 5-1
- 5.3 /usr/include/sys Files 5-8

6 Using the Standard I/O Functions

- 6.1 Introduction 6-1
- 6.2 Using Command Line Arguments 6-4
- 6.3 Using the Standard Files 6-5
- 6.4 Using the Data Stream Functions 6-12
- 6.5 Using More Data Stream Functions 6-24
- 6.6 Using the Low-Level Functions 6-28
- 6.7 Using File Descriptors 6-28
- 6.8 Controlling Terminal Lines Using `termio` and `ioctl()` 6-37

7 Screen Processing

- 7.1 Introduction 7-1
- 7.2 Preparing the Screen 7-7
- 7.3 Using the Standard Screen 7-11
- 7.4 Creating and Using Windows 7-20
- 7.5 Using Other Window Functions 7-34
- 7.6 Combining Movement with Action 7-40
- 7.7 Controlling the Terminal 7-40
- 7.8 Advanced Topics 7-44

8 Character and String Processing

- 8.1 Introduction 8-1
- 8.2 Using the Character Functions 8-1
- 8.3 Testing for Punctuation 8-5
- 8.4 Using the String Functions 8-7

9 Using Process Control

- 9.1 Introduction 9-1
- 9.2 Using Processes 9-1
- 9.3 Calling a Program 9-2
- 9.4 Stopping a Program 9-3
- 9.5 Starting a New Program 9-3
- 9.6 Executing a Program Through a Shell 9-7
- 9.7 Duplicating a Process 9-7
- 9.8 Waiting for a Process 9-8
- 9.9 Inheriting Open Files 9-9
- 9.10 Program Example 9-10

10 Using the Event Manager

- 10.1 Introduction 10-1
- 10.2 Using the Event Manager 10-1
- 10.3 Events 10-3
- 10.4 Event Manager Calls 10-5
- 10.5 Configuration Files 10-10
- 10.6 Event Manager C Language Definitions and Syntax 10-13
- 10.7 Summary of Event Manager C Syntax 10-14
- 10.8 A Sample Program 10-18

11 Writing and Using Pipes

- 11.1 Introduction 11-1
- 11.2 Opening a Pipe to a New Process 11-1
- 11.3 Reading and Writing to a Process 11-2
- 11.4 Closing a Pipe 11-3
- 11.5 Opening a Low-Level Pipe 11-3
- 11.6 Program Examples 11-5
- 11.7 Named Pipes 11-8

12 Using System Resources

- 12.1 Introduction 12-1
- 12.2 Allocating Memory 12-1
- 12.3 Overview of File Locking 12-8
- 12.4 Locking Files Under XENIX 12-9

12.5	Locking Files Under UNIX System V	12-11
12.6	Message Operations	12-23
12.7	Overview of Semaphores	12-50
12.8	Using Semaphores Under XENIX	12-50
12.9	Using Semaphores Under UNIX System V	12-57
12.10	Getting Semaphores	12-63
12.11	Overview of Shared Memory	12-86
12.12	Using Shared Memory	12-87
12.13	Using Shared Memory Under UNIX System V	12-96
12.14	Shared Memory Data Structures	12-97

A Library Routine Error Messages

A.1	Introduction	A-1
A.2	errno Values	A-1
A.3	Math Errors	A-8

B Common Libraries

B.1	Introduction	B-1
B.2	Run-Time Routines	B-1
B.3	Global Variables	B-5
B.4	Include Files	B-6
B.5	Differences Between Routines Common to MS-DOS	B-7

C XENIX to DOS: A Cross Development System

C.1	Introduction	C-1
C.2	Creating Source Files	C-2
C.3	Compiling a DOS Source File	C-2
C.4	Using Assembly Language Source Files	C-4
C.5	Creating and Linking Object Files	C-4
C.6	Running and Debugging a DOS Program	C-5
C.7	Transferring Programs Between Systems	C-5
C.8	Creating DOS Libraries	C-7
C.9	Common Run-Time Routines	C-7
C.10	Common System-Wide Variables	C-9
C.11	Common Include Files	C-10
C.12	Differences Between Common Routines	C-11
C.13	Differences in Definitions	C-21

Chapter 1

Introduction

- 1.1 About the C Library 1-1
- 1.2 About This Manual 1-1
- 1.3 Notational Conventions 1-4

1.1 About the C Library

The Microsoft® C Run-Time Library is a set of more than 200 predefined functions and macros designed for use in C programs. The run-time library makes programming easier by providing the following:

- An interface to operating-system functions (such as opening and closing files)
- Fast and efficient functions to perform common programming tasks (such as string manipulation), sparing the programmer the time and effort needed to write such functions

The run-time library provides many basic functions that are not provided by the C language, including input and output, storage allocation, and process control, among others.

The run-time library is further designed to be compatible with the Draft Proposed American National Standard — Programming Language C (referred to as ANSI C), except for the internationalization functions. Appendix B also lists the functions that conform to the ANSI C standard.

The functions in the Microsoft C Run-Time Library are designed to maintain compatibility between XENIX® and UNIX and MS-DOS® systems. Unless otherwise noted, all XENIX routines may be assumed to be compatible with the UNIX system and most other UNIX-like systems.

In general, compatible functions share the same name. If you are interested in writing portable programs, refer to Appendix B, “Common Libraries.” This appendix lists those functions that are common to both the XENIX and MS-DOS libraries, and describes any significant differences in the operation of common libraries. (Note that the term “MS-DOS” is used in this manual to refer to both MS-DOS and PC-DOS.)

1.2 About This Manual

The *C Library Guide* describes the contents of the C Run-Time Library. To use this manual, you should be familiar with the C language and with XENIX. You should also know how to compile and link C programs on the XENIX system. To learn about the C language, refer to the *C Language Reference*. If you have questions about compiling or linking C programs, see the *XENIX C User's Guide*.

This manual should be used with the Subroutines(S) section of the *XENIX Programmer's Reference*. While the *C Library Guide* provides general information about using the library routines, and describes the routines

C Library Guide

according to different categories of functions, the Subroutines(S) section gives detailed descriptions of the run-time routines in alphabetical order. Once you become familiar with the types of routines available and the rules for using them, you will likely use the *XENIX Programmer's Reference* most often.

Note

Throughout this manual, references to the Subroutines(S) section of the *XENIX Programmer's Reference* will be given simply as *name(S)* where *name* is the name of the library routine.

The chapters of this manual are organized as follows:

Chapter 2, "Using C Library Routines," gives general rules for understanding and using C library routines and mentions special considerations that apply to certain routines. This chapter will likely become valuable as a reference.

Chapter 3, "Global Variables and Standard Types," describes variables and types that are declared and defined in the run-time library and used by the library routines. This chapter cross-references to the include file that defines or declares these variables and types. You may find them useful in your own routines.

Chapter 4, "Run-Time Routines by Category," breaks down the run-time library routines by category, lists the routines that fall into each category, and discusses considerations that apply to each category as a whole. This chapter complements the Subroutines(S) section of the *XENIX Reference Manual*, making it easier to locate routines by task. Once you decide on the routines you want, simply turn to the appropriate manual pages in the Subroutines(S) section for a detailed description.

Chapter 5, "Include Files," summarizes the contents of each include file provided with the run-time library, and lists the routines that use it.

Chapter 6, "Using the Standard I/O Functions," describes the input and output functions already provided by the system. Further, this chapter explains how to use these I/O functions.

Chapter 7, "Screen Processing," describes the functions of the **curses** and **terminfo** libraries, and explains how to use these functions to control the terminal screen.

Chapter 8, “Character and String Processing,” describes the system-provided functions for character and string processing.

Chapter 9, “Using Process Control,” describes the process control functions available with the standard C library.

Chapter 10, “Using the Event Manager,” describes the the Event driver routines available with the XENIX.

Chapter 11, “Creating and Using Pipes,” describes how to create and use pipes. Further, the functions provided in the standard library for controlling pipes are described.

Chapter 12, “System Resources” describes system resource functions. These functions let a program dynamically allocate memory, share memory with other programs, lock files against access by other programs, and use semaphores.

The appendixes for this guide provide more detailed information about error messages and about MS-DOS-compatible routines. Appendix A, “Error Messages,” describes the error values and messages that can appear when using library routines. Appendix B, “Common Libraries,” lists routines of the XENIX C library that are compatible both with routines of the same name on MS-DOS systems and with routines that conform to the ANSI C standard. Appendix B also describes differences (if any) between the XENIX and MS-DOS versions of the routines and discusses common global variables and include files.

Appendix C, “XENIX to DOS: A Cross Development System,” provides a variety of tools to create programs that can be executed under control of the DOS operating system. The DOS cross development system lets you create, compile, and link DOS programs on the XENIX system and transfer these programs to a DOS system for execution and debugging.

C Library Guide

1.3 Notational Conventions

The following notational conventions are used throughout this manual:

Example of Convention	Description of Convention
Examples	<p>The typeface shown in the left column is used to simulate the appearance of information that would be printed on the screen or by a printer. For example, the following command line is printed in this special typeface:</p> <pre>cc -Foout.o -DTRUE=1 file.c</pre> <p>When discussing this command line in text, items appearing on the command line, such as <i>out.o</i>, also appear in a special typeface.</p>
Language elements	<p>Bold type indicates elements of the C language that must appear in source programs as shown. Text that is normally shown in bold type includes operators, keywords, library functions, commands, options, and preprocessor directives. Examples are shown below:</p> <pre>+= #if defined() int if -Fa fopen main sizeof</pre>
ENVIRONMENT, VARIABLES, and MACROS	<p>Bold capital letters are used for environment variables, symbolic constants, and macros.</p>
<i>placeholders</i>	<p>Words in italics are placeholders that you must supply in command-line and option specifications and in the text for types of information. Consider the following option:</p> <pre>-H <i>number</i></pre> <p>Note that <i>number</i> is italicized to indicate that it represents a general form for the -H option. In an actual command, you would supply a particular number for the placeholder <i>number</i>.</p> <p>Occasionally, italics are also used to emphasize particular words in the text.</p>

Missing code

Vertical ellipses are used in program examples to indicate that a portion of the program is omitted. For instance, in the following excerpt, the ellipses between the statements indicate that intervening program lines occur but are not shown:

```
count = 0;
.
.
.
*pc++;
```

[*optional items*]

Brackets enclose optional fields in command-line and option specifications. Consider the following option specification:

-D*identifier*[=*string*]

The placeholder *identifier* indicates that you must supply an identifier when you use the **-D** option. The outer brackets indicate that you are not required to supply an equal sign (=) and a string following the identifier. The inner brackets indicate that you are not required to enter a string following the equal sign, but if you do supply a string, you must also supply the equal sign.

Single brackets are used in C-language array declarations and subscript expressions. For instance, *a[10]* is an example of brackets in a C subscript expression.

Repeating elements...

Horizontal ellipses are used in syntax examples to indicate that more items having the same form may be entered. For example, in the Bourne shell, several paths can be specified in the **PATH** command, as shown in the following syntax:

PATH[=*path*;*path*]...



C Library Guide

`{choice1|choice2}`

Braces and a vertical bar indicate that you have a choice between two or more items. Braces enclose the choices, and vertical bars separate the choices. You must choose one of the items unless all of the items are also enclosed in double square brackets.

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

```
-W {0 | 1 | 2 | 3}
```

You can use **-W1**, **-W2**, or **-W3** to display different levels of warning messages or **-W0** to suppress all warning messages.

“Defined terms”

Quotation marks set off terms defined in the text. For example, the term “far” appears in quotation marks the first time it is defined.

Some C constructs require quotation marks. Quotation marks required by the language have the form " " rather than “ ”. For example, a C string used in an example would be shown in the following form:

```
"abc"
```

KEY+KEY

Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+C. Key sequences to be pressed simultaneously are indicated by the key names in small caps separated by a plus sign (CTRL+C).

Chapter 2

Using C Library Routines

- 2.1 Introduction 2-1
- 2.2 Identifying Functions and Macros 2-1
- 2.3 Including Files 2-3
- 2.4 Declaring Functions 2-4
- 2.5 Stack Checking on Entry 2-5
- 2.6 Argument-Type Checking 2-5
- 2.7 Error Handling 2-6
- 2.8 Filenames and Pathnames 2-7
- 2.9 Floating-Point Support 2-8
- 2.10 Using Huge Arrays with Library Functions 2-9

2.1 Introduction

To use a C library routine, simply call it in your program, just as if the routine were defined in your program. The C library functions are stored, in compiled form, in the library files that accompany your C compiler software.

At link time, your program must be linked with the appropriate C library file or files to resolve the references to the library functions and provide the code for the called library functions. Information about the procedures for linking with the C library can be found in the *XENIX C User's Guide*.

In most cases you must prepare for the call to the run-time library function by performing one or both of the following steps:

1. Include a given file in your program. Many routines require definitions and declarations that are provided by an include file.
2. Provide declarations for library functions that return values of any type other than **int**. The compiler expects all functions to have **int** return type unless declared otherwise. You can provide these declarations by including the C library file containing the declarations or by explicitly declaring the functions within your program.

These are the minimum steps required; you may also want to take other steps, such as enabling type checking for the arguments in function calls.

This chapter discusses the procedures for preparing to use run-time library routines, and special rules (such as filename and pathname conventions) that may apply to some routines.

2.2 Identifying Functions and Macros

The words “function” and “routine” are used interchangeably throughout this manual, and in fact most of the routines in the C run-time library are C functions; that is, they consist of compiled C statements. However, some routines are implemented as “macros.” A macro is an identifier defined with the C preprocessor directive **#define** to represent a value or expression. Like a function, a macro can be defined to take zero or more arguments, which replace formal parameters in the macro definition. For more information on defining and using macros, see the *XENIX C Language Reference*.

The macros defined in the C run-time library behave like functions: they take arguments and return values, and they are invoked in a similar manner. The primary advantage of using macros is that they execute

C Library Guide

faster. Macros are expanded (replaced by their definitions) during preprocessing, so the overhead required for a function call is eliminated. However, unlike a function, which is defined only once, regardless of the number of times it is called, each occurrence of a macro is expanded before the program is compiled. Macros can therefore increase the size of a program, particularly when they appear many times. In several cases, the C library offers both macro and function versions of the same library routine. This allows you to opt between speed of execution and compact program size, whichever is more important to the application.

Some important differences between functions and macros include the following:

- Some macros may treat arguments with side effects incorrectly if the macro is defined so that arguments are evaluated more than once. See the example that follows this list.
- A macro identifier does not have the same properties as a function identifier. In particular, a macro identifier does not evaluate to an address, as a function identifier does. You cannot, therefore, use a macro identifier in contexts requiring a pointer. For instance, if you give a macro identifier as an argument in a function call, the *value* represented by the macro is passed; if you give a function identifier as an argument in a function call, the *address* of the function is passed.
- Since macros are not functions, they cannot be declared, nor can pointers to macro identifiers be declared. Thus, type checking cannot be performed on macro arguments. The compiler does, however, detect cases where the wrong number of arguments is specified for the macro.
- The library routines implemented as macros are defined through preprocessor directives in the library include files. To use a library macro, you must include the appropriate file, or the macro will be undefined.

The routines that are implemented as macros are noted in the Subroutines(S) section of the *XENIX Reference*. You can examine particular macro definitions in the corresponding include file to determine whether arguments with side effects will cause problems.

For example, the following program fragment uses the **toupper** routine from the standard C library:

```
#include <ctype.h>

int a = 'm';
a = toupper(a++);
```

The **toupper** routine is implemented as a macro; its definition in **ctype.h** is as follows:

```
#define toupper(c)  ( (islower(c)) ? _toupper(c) : (c) )
```

The definition uses the conditional operator (**? :**). In the conditional expression, the argument *c* is evaluated twice: once to determine whether or not it is lowercase, and once to return the appropriate result. This causes the argument *a++* to be evaluated twice, thus increasing *a* twice rather than once. As a result, the value operated on by **islower** differs from the value operated on by **_toupper**.

Not all macros have this effect; you can determine whether a macro will handle side effects properly by examining the macro definition before using it.

2.3 Including Files

Many run-time routines use macros, constants, and types that are defined in separate include files. To use these routines, you must incorporate the specified file (using the preprocessor directive **#include**) into the source file being compiled.

The contents of each include file are different, depending on the needs of specific run-time routines. However, in general, include files contain combinations of the following:

- Definitions of manifest constants

For example, the constant **BUFSIZ**, which determines the hardware-dependent size of buffers for buffered input and output operations, is defined in **stdio.h**.

- Definitions of types

Some run-time routines take data structures as arguments or return values with structure types. Include files set up the required structure definitions. For example, most stream input and output operations use pointers to a structure of type **FILE**, defined in **stdio.h**.

2

C Library Guide

- Function declarations

Declarations provide the return type of a function; this is required for any function that returns a value with type other than **int**. (See “Declaring Functions.”)

- Macro definitions

Some routines in the run-time library are implemented as macros. The definitions for these macros are contained in the include files. To use one of these macros, you must include the appropriate file.

The include file or files needed by each routine can be found in the Subroutines(S) section of the *XENIX Programmer's Reference* on the manual page for the routine.

2.4 Declaring Functions

Whenever you call a library function that returns any type of value but an **int**, you should make sure that the function is declared before it is called. The easiest way to do this is to include the file containing declarations for that function, causing the appropriate declarations to be placed in your program. The function declaration in the include file provides the return type of the function.

Your program can contain more than one declaration of the same function as long as the declarations are compatible. This is an important feature to remember if you have older programs whose function declarations do not contain argument-type lists. For instance, if your program contains the declaration

```
char *calloc( );
```

you can also include the following declaration:

```
char *calloc(unsigned, unsigned);
```

Although the two declarations are not identical, they are compatible, so no conflict occurs.

If you wish, you can provide your own function declarations instead of using the declarations in the library include files. However, you should consult the declarations in the include files to make sure that your declarations are correct.

2.5 Stack Checking on Entry

Stack checking means that, on entry to a routine, the stack is first checked to determine whether or not there is room for the local variables used by that routine. If there is, space is allocated by adjusting the stack pointer. Otherwise, a “stack overflow” run-time error occurs. If stack checking is disabled, the compiler assumes there is enough stack space. If in fact there is not sufficient space on the stack, you may overwrite memory locations in the data segment and receive no warning.

2

All XENIX library routines are compiled with stack checking enabled.

2.6 Argument-Type Checking

Microsoft C offers a type-checking feature for the arguments in a function call. Type checking is performed whenever an argument-type list is present in a function declaration and the declaration appears before the definition or use of the function in a program. For information on the form of the argument-type list and the type-checking method, see the *XENIX C Language Reference*.

For functions that you write yourself, you must set up argument-type lists that invoke type checking. You can also use the **-Zg** command-line option to cause the compiler to generate a list of function declarations for all functions defined in a particular source file; the list can then be incorporated into your program. See the *XENIX C User's Guide* for details on using the **-Zg** option.

For functions in the C run-time library, type checking is always enabled. Only limited type checking can be performed on functions that take a variable number of arguments. The following run-time functions are affected by this limitation:

- In calls to **printf** and **scanf**, type checking is performed only on the first argument: the format string.
- In calls to **fprintf**, **fscanf**, **sprintf**, and **sscanf**, type checking is performed on the first two arguments: the file or buffer and the format string.
- In calls to **open**, only the first two arguments are type checked: the pathname and the open flag.

C Library Guide

- In calls to **execl**, **execle**, **execlp**, and **execlepe**, type checking is performed on the first two arguments: the pathname and the first argument pointer.

2.7 Error Handling

When calling a function, it is a good idea to provide for detection and handling of error returns, if any. Otherwise, your program may produce unexpected results.

For run-time library functions, you can determine the expected return value from the return-value discussion on each library page. In some cases no established error return exists for a function. This usually occurs when the range of legal return values makes it impossible to return a unique error value.

The description of some functions in the Subroutines(S) section of the *XENIX Programmer's Reference* indicates that when an error occurs, a global variable named **errno** is set to a value indicating the type of error. Note that you cannot depend on **errno** being set unless the description of the function explicitly mentions the **errno** variable.

When using functions that set **errno**, you can test the **errno** values against the error values defined in **errno.h**, or you can use the **perror** function if you want to print the system error message to standard error (**stderr**). For a list of **errno** values and the associated error messages, see "Error Messages" in this guide.

When you use **errno** and **perror** remember that the value of **errno** reflects the error value for the last call that set **errno**. To prevent misleading results, you should always test the return value before accessing **errno**, to verify that an error actually occurred. Once you determine that an error has occurred, use **errno** or **perror** immediately. Otherwise, the value of **errno** may be changed by intervening calls.

The math functions set **errno** upon error in the manner described on the manual page for each math function in the Subroutines(S) section of the *XENIX Programmer's Reference*. Math functions handle errors by invoking a function named **matherr**. You can choose to handle math errors differently by writing your own error function and naming it **matherr**. When you provide your own **matherr** function, that function is used in place of the run-time library version. You must follow certain rules when writing your own **matherr** function, as outlined in *matherr(S)*.

You can check for errors in stream operations by calling the **ferror** function. The **ferror** function detects whether the error indicator has been set

for a given stream. When the stream is closed or rewound, the error indicator is cleared automatically; or you can reset it by calling the **clearerr** function.

Errors in low-level input and output operations cause **errno** to be set.

The **feof** function tests for end-of-file on a given stream. An end-of-file condition in low-level input and output can be detected with the **eof** function or when a **read** operation returns 0 as the number of bytes read.

2

2.8 Filenames and Pathnames

Many functions in the run-time library accept strings representing pathnames and filenames as arguments. The functions process the arguments and pass them to the operating system, which is ultimately responsible for creating and maintaining files and directories. Thus, it is important to keep in mind not only the C conventions for strings, but also the operating system rules for filenames and pathnames and (where portability to MS-DOS systems is an issue) the differences between XENIX and MS-DOS rules. There are three considerations:

1. Case sensitivity
2. Subdirectory conventions
3. Delimiters for pathname components

Both the C language and the XENIX operating system are case-sensitive, which means that they distinguish between uppercase and lowercase letters. The MS-DOS operating system, however, does not use case differences to distinguish between otherwise identical names. So, while "FILEA" and "fileA" refer to two different files on a XENIX system, they refer to the same file on an MS-DOS system. If you want to prepare portable code, do not take advantage of the case-sensitivity of C and XENIX when specifying filenames.

By convention, some include files are stored in a subdirectory named **sys** on XENIX systems. If portability to MS-DOS systems is a concern, be aware that this XENIX convention is not used on all MS-DOS systems.

The XENIX and MS-DOS operating systems differ in the way they handle pathnames. XENIX uses the forward slash (/) to delimit the components of pathnames, while MS-DOS ordinarily uses the backslash (\). Note, however, that MS-DOS recognizes the forward slash as a delimiter in situations where a pathname is expected. Thus, you can produce portable code by using the forward slash, as long as the context is not ambiguous and a pathname is clearly expected in the program.

C Library Guide

2.9 Floating-Point Support

The math functions supplied in the C run-time library require floating-point support to perform calculations with real numbers. This support can be provided by the floating-point libraries that accompany your compiler software or by an 8087 or 80287 coprocessor. (For information on selecting and using a floating-point library with your program, see the *XENIX C User's Guide*.) The names of the functions that require floating-point support are listed below:

acos	cabs	modf	log	hypot
asin	ceil	ecvt	tan	strtod
atan	cos	exp	tanh	sin
atan2	cosh	fabs	pow	sinh
atof	floor	fcvt	frexp	sqrt
bessel ¹	fmod	ldexp	gcvt	

¹The **bessel** function does not correspond to a single function, but to six functions named **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**.

In addition, the **printf** family of functions (**fprintf**, **printf**, **sprintf**, **vfprintf**, **vprintf**, and **vsprintf**) requires support for floating-point input and output if used to print floating-point values.

The C compiler tries to detect whether floating-point values are used in a program so that supporting functions are loaded only if required. This behavior saves a considerable amount of space for programs that do not require floating-point support.

When you use a floating-point type character in the format string for a **printf** or **scanf** call (**fprintf**, **printf**, **sprintf**, **vfprintf**, **vprintf**, **vsprintf**, **cscanf**, **fscanf**, **scanf**, or **sscanf**), make sure that you specify floating-point values or pointers to floating-point values in the argument list to correspond to any floating-point type characters in the format string. The presence of floating-point arguments allows the compiler to detect floating-point values. If a floating-point type character is used to print an integer argument, for example, floating-point values will not be detected because the compiler does not actually read the format string used in the **printf** and **scanf** functions. For instance, the following program results in incorrect results at run-time:

```
main( )    /* THIS EXAMPLE PRODUCES AN ERROR */
{
    long l = 10L;
    printf("%f", l);
}
```

In the preceding example, the functions for floating-point I/O are not loaded for the following reasons:

- No floating-point arguments are given in the call to **printf**.
- No floating-point values are used anywhere else in the program.

As a result, the *%f* is not recognized as a valid format, and the system simply displays the letter *f*.

The following is a corrected version of the above call to **printf**:

```
main( )      /* THIS EXAMPLE WORKS JUST FINE */
{
    long l = 10L;
    printf("%f", (double) l);
}
```

This version corrects the error by casting the long integer value to **double**.

2.10 Using Huge Arrays with Library Functions

In programs that use small, compact, medium, and large memory models, Microsoft C lets you use arrays exceeding the 64K (kilobyte) limit of physical memory in these models by explicitly declaring the arrays as **huge**. (See your compiler guide for a complete discussion of memory models and the **near**, **far**, and **huge** keywords.) However, you cannot generally pass **huge** data items as arguments to C library functions. In the compact-model library used by compact-model programs, and in the large-model library used by both large-model and huge-model programs, only the following functions use argument arithmetic that works with **huge** items:

fread	memccpy	memcmp	memicmp
fwrite	memchr	memcpy	memset

With this set of functions, you can read from, write to, copy, initialize, or compare **huge** arrays; a **huge** array can be passed without difficulty to any of these functions in a compact-, large-, or huge-model program.

Note that there is a semantic difference between the function and intrinsic versions of the **memset**, **memcpy**, and **memcmp** library routines. The function versions of these routines support huge pointers in compact and large model, but the intrinsic versions do not support huge pointers.

Chapter 3

Global Variables and Standard Types

- 3.1 Introduction 3-1
- 3.2 The daylight, timezone, and tzname Variables 3-1
- 3.3 errno, sys_errlist, sys_nerr 3-2
- 3.4 environ 3-2
- 3.5 Standard Types 3-2

3.1 Introduction

The C run-time library contains definitions for a number of variables and types used by library routines. You can access these variables and types by including in your program the files in which they are declared or by giving appropriate declarations in your program, as shown in this chapter.

3.2 The `daylight`, `timezone`, and `tzname` Variables

The `daylight`, `timezone`, and `tzname` variables are used by several of the time and date functions to make adjustments for local time. The variables are declared as follows in the include file `time.h`:

```
int daylight;
long timezone;
char *tzname[2];
```

The values of the variables are determined by the setting of an environment variable named `TZ`. You can adjust local time by setting the `TZ` environment variable. The value of the environment variable `TZ` must be a three-letter time zone, followed by a signed or unsigned number giving the difference in hours between Greenwich mean time and local time. The number is positive west of Greenwich, and negative east of Greenwich. The number may be followed by a three-letter daylight-saving-time (DST) zone. For example, the following shell environment statement specifies that the local time zone is EST (Eastern standard time), that local time is five hours earlier than Greenwich mean time, and that EDT is the name of the time zone when daylight saving time is in effect:

```
SET TZ=EST5EDT
```

Omitting the DST zone means that daylight saving time is never in effect:

```
SET TZ=EST5
```

When you call the `ftime` or `localtime` function, the values of the three variables `daylight`, `timezone`, and `tzname` are determined from the `TZ` setting. The `daylight` variable is given a nonzero value if a DST zone is present in the `TZ` setting; otherwise, `daylight` is 0. The `timezone` variable is assigned the difference in seconds (calculated by converting the hours given in the `TZ` setting) between Greenwich mean time and local time. The first element of the `tzname` variable is the string value of the three-letter time zone from the `TZ` setting; the second element is the string value of the DST zone. If the DST zone is omitted from the `TZ` setting, `tzname[1]` is an empty string.



C Library Guide

The **ftime** and **localtime** functions call another function, **tzset**, to assign values to the three global variables from the **TZ** setting. You can also call **tzset** directly if you like; see the **tzset** reference in the “Time” section of “Run-Time Routines by Category.”

3.3 **errno**, **sys_errlist**, **sys_nerr**

The **errno**, **sys_errlist**, and **sys_nerr** variables are used by the **perror** function to print error information. When an error occurs in a system-level call, the **errno** variable is set to an integer value to reflect the type of error. The **perror** function uses the **errno** value to look up (index) the corresponding error message in the **sys_errlist** table. The value of the **sys_nerr** variable is defined as the number of elements in the **sys_errlist** array. For a listing of the **errno** values and the corresponding error messages, see “Error Messages” in this guide.

3.4 **environ**

The **environ** variable provides access to memory areas containing process-specific information. This variable is an array of pointers to the strings that constitute the process environment. The environment consists of one or more entries of the form

name=string

where *name* is the name of an environment variable and *string* is the value of that variable. The string may be empty. The initial environment settings are taken from the shell environment at the time of program execution.

The **getenv** and **putenv** routines use the **environ** variable to access and modify the environment table. When **putenv** is called to add or delete environment settings, the environment table changes size. The table’s location in memory may also change, depending on the program’s memory requirements. The **environ** variable is adjusted in these cases and will always point to the correct table location.

3.5 Standard Types

A number of run-time library routines use values whose types are defined in include files. These types are listed and described as follows, and the include file that defines each type is given. For a list of the actual type definitions, see the description of the appropriate include file in the “Include Files” chapter.

Global Variables and Standard Types

Standard Type	Description
clock_t	The clock_t type, defined in time.h , stores time values and is used by the clock function.
FILE	The FILE structure, defined in stdio.h , is the structure used in all stream input and output operations. The fields of the FILE structure store information about the current state of the stream.
jmp_buf	The jmp_buf type, defined in setjmp.h , is an array type rather than a structure type. It defines the buffer used by the setjmp and longjmp routines to save and restore the program environment.
size_t	The size_t type, defined in stddef.h and several other include files, is the unsigned integral result of the sizeof operator.
stat	The stat structure, defined in sys/stat.h , contains file-status information returned by the stat and fstat routines.
time_t	The time_t type, defined in time.h , represents time values in the time routine.
timeb	The timeb structure, defined in sys/timeb.h , is used by the ftime routine to store the current system time in a broken-down format.
tm	The tm structure, defined in time.h , is used by the asctime , gmtime , and localtime functions to store and retrieve time information.
utimbuf	The utimbuf structure, defined in sys/utime.h , stores file access and modification times used by the utime function to change file-modification dates.



C Library Guide

va_list

The **va_list** array type, defined in **varargs.h**, is used to hold information needed by the **va_arg** macro and the **va_end** routine. The called function declares a variable of type **va_list**, which may be passed as an argument to another function.

Chapter 4

Run-Time Routines

by Category

- 4.1 Introduction 4-1
- 4.2 Buffer Manipulation 4-1
- 4.3 Character Classification and Conversion 4-2
- 4.4 Database-Manipulation Routines 4-3
- 4.5 Data Conversion 4-3
- 4.6 Directory Operation 4-4
- 4.7 File Handling 4-5
- 4.8 Group and Password File Control Routines 4-6
- 4.9 Input and Output Routines 4-7
 - 4.9.1 Standard I/O Routines 4-8
 - 4.9.2 Stream I/O Routines 4-8
 - 4.9.3 Low-Level I/O Routines 4-10
- 4.10 Math 4-11
- 4.11 Memory Allocation 4-13
- 4.12 Message-Control Routines 4-14
- 4.13 Pipes 4-14
- 4.14 Process Control 4-16
- 4.15 Random-Number Generation 4-18

- 4.16 Screen Processing 4-18
- 4.17 Searching and Sorting 4-22
- 4.18 Semaphore-Control Routines 4-23
- 4.19 Shared-Memory Routines 4-23
- 4.20 String Manipulation 4-24
- 4.21 System Accounting 4-26
- 4.22 Terminal Control 4-27
- 4.23 Time 4-27
- 4.24 Miscellaneous 4-28

4.1 Introduction

This chapter describes the major categories of routines included in the C run-time libraries. The discussions of these categories are intended to give a brief overview of the capabilities of the run-time library. Some categories of routines, such as “Input and Output,” are discussed in some detail, to help show how the routines are used in programs. For a complete description of the syntax and use of each routine, see the Subroutines(S) section of the *XENIX Programmer’s Reference*. Another source of more detailed information is found in the “Using System Resources” chapter of the *XENIX Programmer’s Guide*.

4.2 Buffer Manipulation

The following buffer-manipulation routines are useful for working with areas of memory on a character-by-character basis. Buffers are arrays of characters (bytes). However, unlike strings, they are not usually terminated with a null character (`\0`). Therefore, the buffer-manipulation routines always take a length or count argument.

4

Routine	Use
memccpy	Copies characters from one buffer to another, until a given character or a given number of characters has been copied.
memchr	Returns a pointer to the first occurrence, within a specified number of characters, of a given character in the buffer.
memcmp	Compares a specified number of characters from two buffers.
memcpy	Copies a specified number of characters from one buffer to another.
memset	Uses a given character to initialize a specified number of bytes in the buffer.

Function declarations for the buffer-manipulation routines are given in the include file **memory.h**.

For more information on the buffer manipulation routines, see **memory (S)** in the *XENIX Programmer’s Reference*.

C Library Guide

4.3 Character Classification and Conversion

The following character classification and conversion routines let you test individual characters in a variety of ways and convert between uppercase and lowercase characters.

Routine	Use
isalnum	Tests for alphanumeric character (letters and digits)
isalpha	Tests for alphabetic character (uppercase and lowercase letters)
isascii	Tests for ASCII character (0-127)
isctrl	Tests for control character (ASCII 0-31 and 127)
isdigit	Tests for decimal digit (0-9)
isgraph	Tests for printable character except space (ASCII 33-126)
islower	Tests for lowercase character
isprint	Tests for printable character (ASCII 32-126)
ispunct	Tests for punctuation character (neither control nor alphanumeric)
isspace	Tests for white space character (space, tab, carriage return, newline, vertical tab, or form feed)
isupper	Tests for uppercase character
isxdigit	Tests for hexadecimal digit (0-9, a-f, A-F)
toascii	Converts character to ASCII code
tolower	Tests character and converts to lowercase if uppercase
_tolower	Converts character to lowercase (unconditional)
toupper	Tests character and converts to uppercase if lowercase
_toupper	Converts character to uppercase (unconditional)

For more information on the character classification and conversion routines, see the “Character and String Processing” chapter in this guide and **ctype**(S) and **conv**(S) in the *XENIX Programmer’s Reference*.

4.4 Database-Manipulation Routines

The following routines are available when you specify the library **dbm** on the compile line. They are provided to give you the tools to perform simple manipulations of a very large database. For more information, see **dbm** (S) in the *XENIX Programmer's Reference*.

Routine	Use
dbmopen	Opens a database file for accessing
delete	Deletes a key and its associated contents
fetch	Accesses data stored under a key
firstkey	Returns the first key in the database
nextkey	Returns the next key following any specified key in the database
store	Stores data under a key

4

4.5 Data Conversion

The data-conversion routines convert numbers to strings of ASCII characters and vice versa. These routines are implemented as functions; all are declared in the include file **stdlib.h**. For details on the use of these functions, see the appropriate manual pages in the Subroutines(S) section of the *XENIX Programmer's Reference*.

Routine	Use
a64l	Converts a base-64 representation to a long
atof	Converts string to float
atoi	Converts string to int
atol	Converts string to long
ecvt	Converts double to string
fcvt	Converts double to string
gcvt	Converts double to string

C Library Guide

itoa	Converts int to string
ltoa	Converts long to string
ltoa3	Converts a list of long integers to a list of 3-byte integers
l3toa	Converts a list of 3-byte integers to a list of long integers
l64a	Converts a long into a base-64 representation
sgetl	Returns a long stored with sputl
sputl	Stores a long in memory
strtod	Converts string to double
strtol	Converts string to a long integer
strtoul	Converts string to an unsigned long integer
ultoa	Converts unsigned long to string

4.6 Directory Operation

The following routines provide control over the special files called directories. For a full description of their use, see the manual entries **directory(S)** and **getdents(S)**.

Routine	Use
closedir	Closes the named directory stream and frees the structure associated with the directory pointer
opendir	Opens the directory named by a filename and associates a directory stream with it
readdir	Returns a pointer to the next directory entry
rewinddir	Resets the position of the named directory stream to the beginning of the directory
seekdir	Sets the position of the next readdir operation on the directory stream
telldir	Returns the current location associated with the named directory stream

4.7 File Handling

The following file-handling routines work on a file designated by a path-name, or by a “file descriptor.” A descriptor is a file-management structure obtained from an **open**, **creat**, **dup**, **fcntl**, or **pipe** system call. The file-handling routines provide or modify information about the designated file. Directories and devices are treated as special files by the XENIX system, so the file-handling routines control their use as well.

Routine	Use
access	Checks file-permission setting
chdir	Changes current working directory
chmod	Changes file-permission setting
chown	Changes file owner and group
chsize	Changes file size
fcntl	Controls open files
fstat	Gets file-status information
getcwd	Gets current working directory
ioctl	Controls character devices
isatty	Checks for character device
link	Links an existing file to a new pathname
locking	Locks or unlocks areas of a file
mknod	Creates a directory, special file, or ordinary file
mktemp	Creates a unique filename
mount	Mounts a file system on a directory
stat	Gets file-status information on named file
umask	Sets default-permission mask
umount	Unmounts file system mounted by mount



C Library Guide

unlink	Deletes a file
ustat	Gets status information about a file system
utime	Sets file access and modification times

The **access**, **chmod**, **chown**, **chroot**, **link**, **mknod**, **stat**, **unlink**, and **utime** routines operate on files specified by a pathname or filename. The **stat** routine is declared in **sys/stat.h**.

The **chsize**, **fcntl**, **fstat**, **ioctl**, **isatty**, and **locking** routines work with files designated by a file descriptor.

The **mount** and **umount** routines accept pointers to pathnames to mount and unmount removable file systems on device files.

The **ustat** routine, which returns information about mounted file systems, works with devices specified by device numbers. To use **ustat**, you must include **sys/types.h** and **ustat.h**.

The **mktemp** and **umask** routines have slightly different functions than the above routines. The **mktemp** routine creates a unique filename. Programs can use **mktemp** to create unique filenames that do not conflict with the names of existing files. The **umask** routine sets the default permission mask for any new files created in a program.

For additional information on any of the file-handling routines, see the Subroutines(S) section of the *XENIX Programmer's Reference*.

4.8 Group and Password File Control Routines

The group and password file control routines provide you with low-level control of the group and password files. Access to these files is restricted to the system administrator. However, you can still search the files. For information on both the group and the password files, see the "Include Files" chapter. For information on a specific routine, see the Subroutines(S) section in the *XENIX Programmer's Reference*.

Routine	Use
endgrent	Closes the group file
endpwent	Closes the password file
getgrent	Reads the next line of the group file

Run-Time Routines by Category

getgrgid	Searches the group file from the beginning for a match to group ID
getgrnam	Searches the group file from the beginning for a match to a name
getpass	Reads a password from <code>/dev/tty</code> , or from the standard input if <code>/dev/tty</code> cannot be opened
getpw	Searches the password file for the specified user ID, and returns the matching line to the buffer
getpwent	Reads the next line in the password file
getpwnam	Searches the password file from the beginning for a matching name
getpwuid	Searches the password file from the beginning for a matching user ID
putpwent	Writes a line on the stream in the same format as that of <code>/etc/passwd</code>
setgrent	Rewinds the group file
setpwent	Rewinds the password file

4

4.9 Input and Output Routines

The input and output routines of the C run-time library let you read and write data to and from files and devices. In C there are no predefined file structures; all data are treated as sequences of bytes. This section provides information on using the input and output (I/O) routines; three basic categories of functions are discussed:

- Standard I/O Routines
- Stream I/O Routines
- Low-Level I/O Routines

C Library Guide

4.9.1 Standard I/O Routines

The standard I/O routines let you read from and write to the standard input and output files. The following sections explain how to read from and write to the standard input and output.

Routine	Use
getchar	Reads a character from stdin
gets	Reads a line from stdin
printf	Writes formatted data to stdout
putchar	Writes a character to stdout
puts	Writes a line to stdout
scanf	Reads formatted data from stdin

4.9.2 Stream I/O Routines

The standard I/O routines described earlier allow programs to read from the standard input and write to the standard output. Use the stream I/O routines to access files not already connected to the program. The stream I/O routines allow a program to open and access ordinary files as if they were a stream of characters.

Routine	Use
clearerr	Clears the error indicator for a stream
fclose	Closes a stream
fdopen	Opens a stream using its descriptor
feof	Tests for end-of-file on a stream
ferror	Tests for error on a stream
fflush	Flushes a stream
fgetc	Reads a character from a stream (function version)

Run-Time Routines by Category

fgets	Reads a string from a stream
fileno	Gets file descriptor associated with a stream
fopen	Opens a stream
fprintf	Writes formatted data to a stream
fputc	Writes a character to a stream (function version)
fputs	Writes a string to a stream
fread	Reads unformatted data from a stream
freopen	Reassigns a FILE pointer
fscanf	Reads formatted data from a stream
fwrite	Writes unformatted data items to a stream
getc	Reads a character from a stream (macro version)
getchar	Reads a character from stdin (macro version)
gets	Reads a line from stdin
getw	Reads a binary int item from stream
printf	Writes formatted data to stdout
putc	Writes a character to a stream (macro version)
putchar	Writes a character to stdout (macro version)
puts	Writes a line to a stream
putw	Writes a binary int item to a stream
scanf	Reads formatted data from stdin
setbuf	Controls stream buffering
setvbuf	Controls stream buffering and buffer size
sprintf	Writes formatted data to string

C Library Guide

sscanf	Reads formatted data from string
tmpfile	Creates a temporary file
ungetc	Places a character in the buffer
vfprintf	Writes formatted data to a stream
vprintf	Writes formatted data to stdout
vsprintf	Writes formatted data to a string

4.9.3 Low-Level I/O Routines

The following low-level routines provide direct access to files and peripheral devices, such as drives and printers. They are actually direct calls to the routines used in XENIX to read from and write to files and peripheral devices. The low-level functions give a program the same control over a file or device as the system, letting it access the file or device in ways that the stream functions do not. However, low-level functions, unlike stream functions, do not provide buffering or any other useful services of the stream functions. This means that any program that uses the low-level functions must handle input and output.

Routine	Use
close	Closes a file
creat	Creates a file
dup	Creates a second descriptor for a file
dup2	Reassigns a descriptor to a file
eof	Tests for end-of-file
fseek	Repositions FILE pointer to a given location
ftell	Gets current FILE pointer position
lseek	Repositions file pointer to a given location
open	Opens a file
read	Reads data from a file

rewind	Repositions FILE pointer to beginning of a stream
write	Writes data to a file

4.10 Math

The following math routines let you perform common mathematical calculations. All math routines work with floating-point values and therefore require floating-point support (see “Floating-Point Support” in the “Using C Library Routines” chapter). Function declarations for the math routines are given in the include file **math.h**.

Routine	Use
abs	Calculates absolute value of an integer
acos	Calculates arc cosine
asin	Calculates arc sine
atan	Calculates arc tangent
atan2	Calculates arc tangent
bessel	Bessel functions (see j0 , j1 , jn , y0 , y1 , yn , below)
cabs	Finds absolute value of a complex number
ceil	Finds integer ceiling
cos	Calculates cosine
cosh	Calculates hyperbolic cosine
erf	Calculates error function
erfc	Calculates complementary error function
exp	Calculates exponential function
fabs	Finds absolute value
floor	Finds largest integer less than or equal to argument
fmod	Finds floating-point remainder

C Library Guide

frexp	Calculates an exponential value
gamma	Calculates log gamma
hypot	Calculates hypotenuse of right triangle
j0, j1, jn, y0, y1, yn	Calculates Bessel functions of the first and second kinds for real arguments and integer orders
ldexp	Calculates argument times 2^{exp}
log	Calculates natural logarithm
log10	Calculates base-10 logarithm
matherr	Handles math errors
modf	Breaks down argument into integer and fractional parts
pow	Calculates a value raised to a power
rand, srand	Generates a pseudo-random number (srand generates the seed)
sin	Calculates sine
sinh	Calculates hyperbolic sine
sqrt	Finds square root
tan	Calculates tangent
tanh	Calculates hyperbolic tangent

The **matherr** routine is invoked by the math functions when errors occur. This routine is defined in the library, but can be redefined by the user if different error-handling procedures are desired. The user-defined **matherr** function, if given, must conform to the specifications given in **matherr(S)**.

You are not required to supply a definition for **matherr**. If no definition is present, the default error returns for each routine are used. For a description of the routine's error returns, see **matherr(S)** in the *XENIX Programmer's Reference*.

The trigonometric functions, **sin**, **cos**, **tan**, **asin**, **acos**, **atan**, and **atan2**, are described in detail in **trig(S)** in the *XENIX Programmer's Reference*. For explanations of **exp**, **log**, **log 10**, **pow**, and **sqrt**, see **exp(S)**.

4.11 Memory Allocation

The following memory-allocation routines let you allocate, free, and reallocate blocks of memory. They are declared in the include file **malloc.h**.

Routine	Use
calloc	Allocates storage for array
free	Frees a block allocated with calloc , malloc , or realloc
malloc	Allocates a block
realloc	Reallocates a block
sbrk	Resets break value

4

The **calloc** and **malloc** routines allocate memory blocks. The **malloc** routine allocates a given number of bytes, while **calloc** allocates and initializes to 0 an array with elements of a given size.

The **realloc** routine reallocates a memory block, either by changing its size or changing its location; the contents of the block remain unchanged.

A low-level memory-allocation routine is provided by **sbrk**. It increases the program's break value (the address of the first location beyond the end of the default data segment), allowing the program to take advantage of available unallocated memory.

Note

In general, a program that uses the **sbrk** routine should not use the other memory-allocation routines, although their use is not prohibited. In particular, using **sbrk** to decrease the break value may cause unpredictable results from subsequent calls to the other memory-allocation routines.

4.12 Message-Control Routines

The following message-control routines provide the medium for interprocess communication. To use the message-control routines, you must include **sys/types.h**, **sys/ipc.h**, and **sys/msg.h** at the beginning of your program. Message operations are outlined in **msgop(S)** in the *XENIX Programmer's Reference*, and in the *XENIX System V/386 Programmer's Guide*.

Routine	Use
msgctl	Provides for message-control operations
msgget	Returns a message queue identifier
msgsnd	Sends a message to a queue
msgrcv	Reads a message from a queue

4.13 Pipes

A "pipe" is an artificial file that a program can create and use to pass information to other programs. A pipe is similar to a file in that it has a file pointer or a file descriptor, or both, and can be read from or written to using the input and output functions of the standard library. Unlike a file, a pipe does not represent a specific file or device. Instead, a pipe represents temporary storage in memory that is independent of the program's own memory and is controlled entirely by the system.

Routine	Use
pclose	Closes a pipe opened by popen
pipe	Opens a pipe for reading and writing
popen	Opens a pipe between a calling process and a command

Pipes pass information between programs, just as the shell pipe symbol (`|`) passes the output of one program to the input of another. This eliminates the need to create temporary files to pass information to other programs. A pipe can also be used as a temporary storage place for a single program. A program can write to the pipe, then read that information back at a later time.

The standard library provides several pipe functions. The **popen** and **pclose** functions control both a pipe and a process. The **popen** function opens a pipe and creates a new process at the same time, making the new pipe the standard input or output of the new process. This function is typically used in programs that need to call another program and pass substantial amounts of data to that program.

4

The stream I/O functions, including **fscanf** and **fprintf**, can read from or write to a pipe opened by **popen**. Stream I/O functions are outlined in “Stream I/O Routines” in this chapter.

The **pclose** function closes a pipe opened by **popen** and waits for termination of the corresponding process.

The **pipe** function gives low-level access to a pipe. This function is similar to **open** (`S`), but opens the pipe for both reading and writing, returning two file descriptors instead of one. The program can either use both sides of the pipe or close the one it does not need. This function typically opens a pipe in preparation for linking it to a child process.

The low-level input and output functions **read** and **write** can read from and write to a pipe. The low-level I/O functions are described in the “Low-Level I/O Routines” section. Pipe file descriptors are used in the same way as other file descriptors.

C Library Guide

4.14 Process Control

The term “process” refers to a program being executed by XENIX. A process consists of instructions and data and a table of information about the program, such as its allocated memory, open files, and current execution status. Whenever you execute a program, you start a process.

The system identifies each process with a unique ID number. These process ID numbers allow the system to run several processes simultaneously without confusing them. The following are process control routines:

Routine	Use
abort	Aborts a process
alarm	Sets the alarm of the calling process
execl	Executes child process with argument list
execle	Executes child process with argument list and given environment
execlp	Executes child process using PATH variable and argument list
execv	Executes child process with argument array
execve	Executes child process with argument array and given environment
execvp	Executes child process using PATH variable and argument array
exit	Terminates process
_exit	Terminates process without flushing buffers
fork	Creates a new process
getpgrp	Gets group process ID number
getpid	Gets process ID number
getppid	Gets parent process ID number
signal	Raises the signal; used with ssignal

Run-Time Routines by Category

kill	Sends a signal to a process or group of process
lock	Locks a process in main memory
monitor	Prepares an execution profile; used with profil
nap	Suspends a process for a period of time, or until a signal is received
nice	Decreases CPU priority of a process
pause	Suspends a process until a specified signal is received
proct1	Controls active processes or groups of processes
profil	Creates an execution-time profile of a section of core memory
ptrace	Allows parent process to trace execution of a child process
rdchk	Checks to see if there is data to be read
sbrk	Alters amount of space allocated to the data segment of the calling process
setpgrp	Sets group ID of a process
signal	Allows a process to handle signals
sleep	Suspends execution of calling process for a period of time
ssignal	Implements software signals
system	Executes a XENIX command
times	Gets execution times of processes and child processes
ulimit	Provides control over process limits
wait	Suspends the calling process until it traps a specified signal or until a child process stops

C Library Guide

4.15 Random-Number Generation

The following random-number routines generate pseudo-random numbers using the linear congruent algorithm and 48-bit integer arithmetic. The other three routines, **srand48**, **seed48**, and **lcong48**, are complex in nature. For a full description of the use of these pseudo-random-number generators, see **drand48** (S) in the *XENIX Programmer's Reference*.

Routine	Use
drand48 , erand48	Returns a non-negative double-precision floating-point value uniformly distributed over the interval [0.0, 1.0]
lrand48 , nrand48	Returns a non-negative long integer uniformly distributed over the interval $[0, 2^{31}]$
mrand48 , jrand48	Returns a signed long integer uniformly distributed over the interval $[-2^{31}, 2^{31}]$

4.16 Screen Processing

The screen processing functions allow you to use the capabilities provided by the **curses** and **terminfo** libraries. These libraries provide functions for creating and updating windows on the screen, getting input from the terminal, setting terminal modes, and optimizing the motion of the cursor on the screen.

A discussion of the screen processing functions is beyond the scope of this section. Chapter 7, "Screen Processing" describes in detail the functions listed and explains how to call the appropriate libraries in the compile command line. For further information, refer to **curses** (S) and **terminfo** (S) in the *XENIX Programmer's Reference*.

Routine	Use
addch	Adds a character to the standard screen
addstr	Adds a string to the standard screen
addkey	Defines a new function key.
box	Draws a box, using the specified characters
clear	Clears the standard screen and sets the <i>clear</i> flag

Run-Time Routines by Category

clearok	Sets or clears the <i>clear</i> flag
clrrobot	Clears the standard screen from the current position to the bottom of the screen
clrtoeol	Clears the standard screen from the current position to the end of the current line
crmode, nocrmode	Sets or clears CBREAK mode for the terminal
delch	Deletes a character from the standard screen
deleteln	Deletes a line from the standard screen
delwin	Deletes a window
dmpwin	Saves the contents of a window to a file.
echo, noecho	Sets or clears ECHO mode for the terminal
endwin	Terminates screen processing
erase	Clears the screen without setting the <i>clear</i> flag
getch	Gets a character from the standard input
getstr	Gets a string from the standard input
gettmode	Gets the <i>ty</i> mode
getyx	Saves the current line and column positions
inch	Reads a character from the standard screen
initscr	Initializes the standard screen
insch	Inserts a character on the standard screen
insertln	Inserts a line on the standard screen
keypad	This macro allows function key sequences to be considered as a single token.
leaveok	Sets or clears the <i>cursor</i> flag

C Library Guide

longname	Returns the full name of the terminal corresponding to a termcap or terminfo identifier
move	Moves the pointer to a specified position
mv<func>	Moves the cursor and performs the function call
mvcur	Moves the cursor
mvwin	Moves a window
newwin	Creates a new window
nl, nonl	Sets or clears NEWLINE mode for the terminal
overlay	Lays one window over another, without destroying the lower window
overwrite	Writes the contents of one window over another, destroying the contents of the lower window
printw	Prints formatted data to the standard screen
raw, noraw	Sets or clears RAW mode for a terminal
refresh	Updates the standard screen to show any changes
resetty	Restores terminal flags saved with savetty
savetty	Saves current terminal flags
scanw	Reads formatted data from the standard input
scroll	Scrolls a window up a line
scrollok	Sets or clears the <i>scroll</i> flag
setterm	Sets the terminal type
standend	Restores normal attribute for the standard screen

Run-Time Routines by Category

standout	Sets <i>standout</i> attribute for the standard screen
subwin	Creates a subwindow
touchwin	“Touches” a window for a subsequent refresh or wrefresh
waddch	Adds a character to a window
waddstr	Adds a string to a window
wclear	Clears a window and sets the <i>clear</i> flag
wclrto bot	Clears a window from the current position to the bottom of the screen
wclrtoeol	Clears a window from the current position to the end of the line
wdelch	Deletes a character from a window
wdeleteln	Deletes a line from a window
werase	Clears a window without setting the <i>clear</i> flag
wgetch	Gets a character from the standard input
wgetstr	Gets a string from the standard input
winch	Reads a character from a window or screen
winsch	Inserts characters in a window
winsertln	Inserts a line in a window
wmove	Moves a window
wprintw	Prints formatted data to a window
wrefresh	Updates the screen to show changes in a window
wscanw	Gets formatted data from the standard input

C Library Guide

wstandend	Clears standout mode for a window or a screen
wstandout	Sets standout mode for a window or a screen

4.17 Searching and Sorting

The following routines provide the means to perform searches and/or sorts using a number of different algorithms.

Routine	Use
bsearch	Performs binary search
ftw	Walks a hierarchical file tree
hcreate	Allocates sufficient space for the hash table
hdestroy	Destroys the hash table
hsearch	Manages a hash table
lfind	Performs linear search for given value
lsearch	Performs linear search for given value, which is added to array if not found
qsort	Performs quick sort
tdelete	Deletes a node from a binary tree
tfind	Searches a binary tree for a datum
tsearch	Builds and accesses a binary tree
twalk	Traverses a binary tree

The **bsearch**, **lfind**, **lsearch**, and **qsort** functions provide helpful binary-search, linear-search, and quick-sort utilities. For detailed information on these routines, see **bsearch** (S), **hsearch** (S), **lsearch** (S), and **tsearch** (S) in the *XENIX Programmer's Reference*.

4.18 Semaphore-Control Routines

The following semaphore routines control the semaphores that signal when a resource is available or locked. For detailed information, see **semctl(S)** and other appropriate pages in the Subroutines(S) section of the *XENIX Programmer's Reference*, and the *XENIX Programmer's Guide*.

Routine	Use
creatsem	Creates a binary semaphore
nbwaitsem	Provides the calling process with access to the semaphore; returns the error ENAVAIL if the resource is in use
opensem	Opens a semaphore for use by a process
semget	Returns the semaphore identifier associated with a key
semct	Provides a variety of semaphore-control operations
semop	Allows the execution of an array of semaphore operations on a set of semaphores
sigsem	Signals a process waiting for a semaphore that it may proceed and use the resource governed by the semaphore
waitsem	Provides the calling process with access to the semaphore; puts the calling process to sleep if the resource is in use

4

4.19 Shared-Memory Routines

The following shared memory routines provide control functions for the use of shared memory segments. For details, see **shmop(S)** in the *XENIX Programmer's Reference*.

Routine	Use
ftok	Forms a key to provide to the msgget , semget , and shmget system calls (for interprocess communication).

C Library Guide

sdenter	Indicates that the current process is about to access the contents of a shared data segment
sdfree	Detaches the current process from the shared data segment that is attached at the specified address
sdget	Attaches a shared data segment to the data space of the current process
sdgetv	Returns the version number of the data segment at the specified address
sdleave	Indicates that the current process has finished modifying the contents of the shared data segment; alters the version number on exiting
sdwaitv	Forces the current process to sleep until the version number of the indicated segment is no longer equal to the value <i>vnum</i>
shmat	Attaches the shared memory segment associated with the shared memory identifier specified by <i>shmid</i> to the data segment of the calling process
shmctl	Provides control of various shared-memory operations
shmdt	Detaches the calling process's data segment from the shared memory segment located at a specified address
shmget	Gets a shared memory segment associated with a key

4.20 String Manipulation

The following string functions concatenate, compare, copy, and count the number of characters in a string. Many string functions have two forms:

- a form that manipulates all characters in the string
- a form that manipulates a given number of characters

This gives a program very fine control over all or part of a string.

Run-Time Routines by Category

All string functions work on null-terminated character strings. When working with character arrays that do not end with a null character, you can use the buffer-manipulation routines, described earlier in this chapter.

Routine	Use
strcat	Appends a string
strchr	Finds first occurrence of a given character in string
strcmp	Compares two strings
strcpy	Copies one string to another
strcspn	Finds first occurrence of a character from given character set in string
strdup	Duplicates string
strlen	Finds length of string
strncat	Appends characters of string
strncmp	Compares characters of two strings
strncpy	Copies characters of one string to another
strpbrk	Finds first occurrence of character from one string in another
strrchr	Finds last occurrence of given character in string
strspn	Finds first substring from given character set in string
strtok	Finds next token in string

4

The sections that follow describe the string functions; for further information, refer to **string (S)** in the *XENIX Programmer's Reference*.

C Library Guide

4.21 System Accounting

The following system accounting routines are typically used by the system administrator to check and manipulate the contents of the system accounting files. For additional information, see **getut(S)** and other pages in the *XENIX Programmer's Reference*.

Routine	Use
acct	Enables or disables system accounting.
cuserid	Returns a pointer to a string that represents the login name of the owner of the current process.
endutent	Closes the currently opened file.
getutent	Reads the next entry from a system accounting file.
getlogin	Returns a pointer to the login name as found in the file <i>/etc/utmp</i> .
getuid	Searches forward from the current file position until it encounters an entry of the specified identification.
getuline	Searches forward from the current file position until it encounters an entry of the specified line.
putuline	Writes an entry (in the <i>utmp</i> format) in the system accounting file.
setutent	Resets the input stream to the beginning of the file.
utmpname	Allows the user to alter the name of the file examined. Default is <i>/etc/utmp</i> .
ttyslot	Returns the index of the current user's entry in the <i>/etc/utmp</i> file.

4.22 Terminal Control

The terminal-control routines let you use the capabilities provided by the terminal capability database, *termcap* (M). For more details, see *termcap* (S) in the *XENIX Programmer's Reference*.

Routine	Use
tgetent	Extracts the entry for a terminal buffer
tgetflag	Returns 1 if the specified identification capability is present in the terminal's entry in the <i>/etc/termcap</i> file; returns zero if it is not
tgetnum	Returns the numeric value of the specified identification capability. Returns -1 if the terminal is not in the <i>/etc/termcap</i> file
tgetstr	Gets the string value of the specified identification capability and places it in a buffer
tgoto	Returns a cursor-addressing string
tputs	Decodes the leading padding information of the string

4

4.23 Time

The following time routines let you obtain the current time, then convert and store it according to your particular needs. The current time is always taken from the system time.

Routine	Use
asctime	Converts time from structure to character string
clock	Returns the elapsed CPU time for a process
ctime	Converts time from long integer to character string
ftime	Gets current system time as structure
gmtime	Converts time from integer to structure

C Library Guide

localtime	Converts time from integer to structure with local correction
stime	Sets the system time
time	Gets current system time as long integer
tzset	Sets external time variables from environment time variable

The **time** and **ftime** functions return the current time as the number of seconds elapsed since Greenwich mean time, January 1, 1970. This value can be converted, adjusted, and stored in a variety of ways, using the **asctime**, **ctime**, **gmtime**, and **localtime** functions.

The **clock** function returns the elapsed CPU time for the calling process.

The **ftime** function requires two include files: **sys/types.h** and **sys/timeb.h**. The **ftime** function is declared in **sys/timeb.h**. The remainder of the time functions are declared in the include file **time.h**.

When you want to use **ftime** or **localtime** to make adjustments for local time, you must define an environment variable named **TZ**. See Section 3.2 on the global variables **daylight**, **timezone**, and **tzname** for a discussion of the **TZ** variable; **TZ** is also described in **tzset(S)** in the *XENIX Programmer's Reference*.

4.24 Miscellaneous

The “miscellaneous” category covers a number of commonly used routines that do not fit easily into any of the other categories.

Routine	Use
assert	Checks the validity of a given expression.
ctermid	Returns a pointer to a string that contains the filename of the controlling terminal of the calling process.
defopen	Opens the default file specified by filename. Calling defopen with NULL closes the default file.

Run-Time Routines by Category

defread	Reads the previously opened file from the beginning until it encounters a line beginning with a specified pattern; then returns a pointer to the first character in the line following the pattern.
fxlist	Performs the same function as xlist , except that fxlist accepts a pointer to a previously opened file instead of the filename of a file.
getenv	Searches the environment list for a string and returns the associated value.
getopt	Returns the next option letter that matches a letter in a string of recognized option letters.
logname	Returns a pointer to the null-terminated login name (determined by the environment variable).
longjmp	Restores the environment saved by the last call of setjmp (see setjmp below).
nlist	Examines the executable output file and extracts a list of values that is matched to a specified name list; matches the name type and value to be inserted into the next two fields in the output file.
perror	Produces a short message on the standard error, <i>stderr</i> , describing the last error encountered during a system call from a C program.
putenv	Changes or adds the value of an environment variable.
regex	Executes a compiled regular expression against a string.
regcmp	Compiles a regular expression and returns a pointer to the compiled form.
setgid	Sets the real and effective group IDs of the calling process.
setjmp	Performs a nonlocal goto , saves its stack environment, and returns zero.

C Library Guide

setuid	Sets the real and effective user IDs of the calling process.
shutdn	Flushes all information in the core memory and halts the CPU.
swab	Swaps bytes.
sync	Updates the super-block; causes all information in memory that should be on disk to be written out.
tmpfile	Creates a temporary file and returns a corresponding file pointer.
tmpnam	Generates a unique filename for a temporary file.
ttyname	Returns a pointer to the null-terminated pathname of the terminal device associated with the file descriptor.
uname	Returns a null-terminated character string naming the current XENIX system.
xlist	Functions identically to nlist , but uses different data structures with more information, such as segment value and longer symbol names (see nlist).

The **assert** routine is a macro and is defined in **assert.h**. The **setjmp.h** and **longjmp.h** functions are declared in **setjmp.h**.

The **assert** macro is typically used to test for program logic errors; it prints a message when a given “assertion” fails to hold true. Defining the identifier **NDEBUG** to any value causes occurrences of **assert** to be removed from the source file, thus allowing you to turn off assertion checking without modifying the source file.

The **getenv** and **putenv** routines provide access to the environment table. The global variable **environ** also points to the environment table, but it is recommended that you use the **getenv** and **putenv** routines to access and modify environment settings rather than accessing the environment table directly.

Run-Time Routines by Category

The **perror** routine prints the system error message, along with an optional user-supplied message, for the last system-level call that produced an error. The error number is obtained from the **errno** variable. The system message is taken from the **sys_errlist** array. The **errno** variable is guaranteed to be set upon error for only those routines that explicitly mention the **errno** variable in the "Return Value" section of the manual pages in the Subroutines(S) section of the *XENIX Programmer's Reference*.

The **setjmp** and **longjmp** functions save and restore a stack environment. These routines let you execute a nonlocal goto.

Chapter 5

Include Files

- 5.1 Introduction 5-1
- 5.2 /usr/include Files 5-1
- 5.3 /usr/include/sys Files 5-8

5.1 Introduction

The include files in the XENIX system are divided into two groups:

- those that reference system information (kept in */usr/include/sys*)
- those that may be useful to individual users (kept in */usr/include*)

This demarcation is not absolute and you may find yourself using a number of the include files in the */usr/sys* directory.

This chapter briefly describes all the XENIX include files. Descriptions of include files are divided into two sections, user include files and system include files.

5.2 /usr/include Files

The following section describes the function of each *include* file and lists the routines that may be found in each. The *include* files may also contain macro and constant definitions, type definitions, function declarations, and structure definitions.

Declarations or definitions of special interest will be noted. For detailed information on a particular routine, see the appropriate page in Subroutines(S) in the *XENIX Programmer's Reference*, or File Formats(F) in the *XENIX User's Reference*.

5

ar.h

ar.h defines file archiving. It sets the value of the archive file's unique identifier, the "archive number." The structure **ar_hdr** defines the header inserted before each file in an archive.

assert.h

Defines a macro that is useful in verifying the validity of a specified C statement. For more information, see **assert(S)** in the *XENIX Programmer's Reference*.

C Library Guide

core.h

Defines the location and size of a core-image file. For detailed information on the structure of core files, see **core(F)** in the *XENIX User's Reference*.

ctype.h

Defines a number of macros that classify ASCII-coded integer values by doing a table lookup. For a complete list of the available macros, see **ctype(S)** in the *XENIX Programmer's Reference*.

curses.h

Provides a number of routines that control screen and cursor functions. For a complete list of all the available functions, see **curses(S)** in the *XENIX Programmer's Reference*.

dbm.h

Defines the following routines:

dbminit	firstkey
delete	nextkey
fetch	store

These routines are used for handling very large (up to one billion blocks) databases. For more detailed information, see **dbm(S)** in the *XENIX Programmer's Reference*.

dumprestor.h

Defines the format of the header record and the first record of each description. When incremental dumps are made onto magnetic tapes, the files that are dumped are preceded by information defined by the structure **spcl**.

The structure **idates** describes an entry to the file where the dump history is kept.

errno.h

This file contains definitions of error codes that are passed to the external variable **errno**. When an error condition occurs during a system call, the kernel sets the **errno** variable to the appropriate value. For a complete list of these error codes and descriptions of how they occur, see “System Error Values.”

For information on error handling, see **perror(S)** in the *XENIX Programmer's Reference*.

execargs.h

Provides information for the shell. Not for use by the user.

fcntl.h

Provides the values for the file-control function **fcntl**. For a description of the values, see **fcntl(S)** in the *XENIX Programmer's Reference*.

ftw.h

Contains predefined values for an integer used by the system call **ftw**. These values represent the status of the object that **ftw** is examining.

grp.h

Defines a structure, **group**, which returns pointers to information about entries in the file */etc/group*. For more information, see **getgrent(S)** in the *XENIX Programmer's Reference*.

macros.h

Defines a number of useful macros (some for string handling, others for library routines).



C Library Guide

malloc.h

Defines the **mallinfo** structure (which contains information on memory allocation). Defines the following routines:

free	mallinfo
malloc	mallopt
realloc	

For more information, see **malloc(S)** in the *XENIX Programmer's Reference*.

math.h

Defines the following math routines:

acos	erfc	j1	sin
asin	exp	jn	sinh
atan	fabs	ldexp	sqrt
atan2	floor	log	tan
atof	fmod	log10	tanh
ceil	frexp	matherr	y0
cos	gamma	modf	y1
cosh	hypot	pow	yn
erf	j0		

It also defines a number of useful mathematical constants.

For detailed information on the math functions, see **bessel(S)**, **exp(S)**, **floor(S)**, **gamma(S)**, **hypot(S)**, **sinh(S)**, and **trig(S)** in the *XENIX Programmer's Reference*.

For information on **matherr** return values, see "System Error Values."

memory.h

Defines the following routines:

memccpy	memcpy
memchr	memset
memcmp	movedata

These routines are used for buffer manipulation.

mnttab.h

The structure **mnttab** defines the format of the */etc/mnttab* file. This file keeps a record of special files mounted using the **mount** command. For more information, see **mount(S)** in the *XENIX Programmer's Reference*.

mon.h

Defines two structures, **mon** and **monhdr**. These structures determine the format of the buffer in which **monitor** stores information on the execution profile of a specified program. For more information, see **monitor(S)** and **profil(S)** in the *XENIX Programmer's Reference*.



pwd.h

Defines two structures, **passwd** and **comment**, which determine the format of the entries in the */etc/passwd* file and the format of the comments associated with these entries. For details on the structure of the entries, see **getpwent(S)** in the *XENIX Programmer's Reference*.

regexp.h

Defines the following routines:

advance	getrnge
compile	step
ecmp	

C Library Guide

These functions compile regular C language expressions and return pointers to the compiled forms. For a detailed description, see **regexp(S)** in the *XENIX Programmer's Reference*.

sd.h

Defines a number of flags for the **sdget** system call. Defines the **sdget** system call. For more information, see **sdget(S)** in the *XENIX Programmer's Reference*.

search.h

Defines a structure, **entry**, and an enumeration type, **action**, for the **hsearch** system call. Defines an enumeration type, **visit**, for the **tsearch** system call.

setjmp.h

Provides data to ensure that the **setjmp** and **longjmp** system calls are machine independent.

sgtty.h

Defines the structure **sgttyb** for the **stty** and **gtty** system calls. Defines the **stty** and **gtty** system calls, terminal modes, delay algorithms, speeds, and **ioctl** arguments. Defines the structure **tchars**, which handles special characters. For more information, see **ioctl(S)**, **stty(C)**, and **tty(M)** in the *XENIX Programmer's Reference*.

signal.h

Defines the values that can be assigned to **signal** by the kernel. These values are returned to the calling process upon receipt of an error. For more details, see **signal(S)** in the *XENIX Programmer's Reference*.

stand.h

Provides the necessary information and structures for the operation of the system in STANDALONE mode.

stdio.h

Defines the standard buffered input and output routines. The files *stdin*, *stdout*, and *stderr* are defined. The following routines are defined:

ftell	rewind
getchar	setbuf
putchar	

Macros are defined for **clearerr**, **feof**, **ferror**, and **fileno**.

For details on how to use the standard I/O routines, see the following routines in Subroutines(S) in the *XENIX Programmer's Reference*.

close	ferror	putc
ctermid	open	puts
cuserid	popen	read
fclose	printf	scanf

string.h

Defines the following string-manipulation routines:

strcmp	strncmp
strcspn	strspn
strlen	

For details, see **string(S)** in the *XENIX Programmer's Reference*.

termio.h

Defines characters and modes for the terminal interface. In addition, a structure is defined for the **ioctl** system calls. For more information, see **ioctl(S)** *XENIX Programmer's Reference* and **tty(M)** in the *XENIX User's Reference*.

time.h

Defines the structure for the conversion of time to ASCII format. Defines the routine **tzset** and the variables *timezone*, *daylight*, and *tzname*. For details, see **ctime(S)** in the *XENIX Programmer's Reference*.

C Library Guide

unistd.h

Defines the flag values for the **lock** system call. For details, see **lock(S)** in the *XENIX Programmer's Reference*.

ustat.h

Defines the structure **ustat**, which returns information about a given mounted file system. For details, see **ustat(S)** in the *XENIX Programmer's Reference*.

utmp.h

Defines the format for the */etc/utmp* system accounting file. For details, see **utmp(M)** in the *XENIX User's Reference*.

values.h

Defines various values for machine-dependent variables.

varargs.h

Contains macros for use in variable-argument functions. Provided to allow portability of C language code.

5.3 */usr/include/sys* Files

The following include files are system files. Many of them define system parameters or contain information used by the kernel.

a.out.h

Declares the following structures:

aexec	xexec	xlist
bexec	xext	xseg
nlist	xiter	

These structures define (respectively): the **a.out** header, the **b.out** header, the structure for the **nlist** library call, the **x.out** header, the **x.out** header

extension, the **x.out** iteration record, the structure for the **xlist** library call, and the **x.out** segment-table entry.

For more detailed information, see the *XENIX C User's Guide* and **a.out(F)** in the *XENIX User's Reference*.

acct.h

Defines the structure **acct**, used in system accounting. For more information, see **acct(F)** in the *XENIX User's Reference*. and **accton(ADM)** in the *XENIX System Administrator's Guide*.

assert.h

Defines the **assert** macro. For more details, see **assert(S)** in the *XENIX Programmer's Reference*.

brk.h

Defines the commands for break control.

5

buf.h

Defines the structures **buf** and **hbuf**. The **buf** structure provides access to an I/O buffer for device drivers, and the **hbuf** structure provides fast access to the buffers through hashing.

callo.h

Defines the structure **callo**, which allows a clock interrupt for a specific period.

conf.h

Defines the structures **linesw**, **bdevsw**, and **cdevsw**, which are an array of function declarations to a line discipline switch, a block device switch, and a character device switch, respectively.

C Library Guide

dir.h

Defines the structure **direct**, which contains the value for the maximum directory size.

errno.h

Contains the values for the **errno** variable. The kernel sets the **errno** variable upon encountering an error. Math routines also set it. For more information, see **perror(S)** in the *XENIX Programmer's Reference*.

fbk.h

Defines the structure **fbk**, which contains the address of the next free block.

file.h

Defines a structure, **file**, which holds the read/write pointer associated with each open file.

filsys.h

Defines the structure of the super-block and a number of fundamental system variables.

ino.h

Defines the structure of the inode as it appears on a disk block.

inode.h

Contains definitions of the structures **iisem**, **iisd**, and **inode**. The **iisem** structure provides information about the semaphores related to a given inode. The **iisd** structure provides information about the shared data segments allocated to the inode. The **inode** structure provides information about the inode itself.

iobuf.h

Defines the structure of the I/O buffer for each block device.

ioctl.h

Defines macros for I/O control.

ipc.h

Provides constant definitions for the interprocess communications (IPC) report. For more information, see **ipc(S)** in the *XENIX Programmer's Reference*.

lock.h

Defines flag values for the locking of resources.

locking.h

Defines flag values for the **locking** system call. Defines the structure **locklist**, which provides the structure for the linked list of lock regions.



machdep.h

Defines machine-dependent variables (such as the number of descriptor table entries and clock timing).

map.h

Defines the structure **map**, which holds the location of the swapmap.

mmu.h

For memory-management purposes, defines constants for the descriptor tables.

C Library Guide

mount.h

Defines the structure **mount**. One is allocated for every device mounted. For more information, see **mount(S)** in the *XENIX Programmer's Reference*.

msg.h

Defines the structures **msqid_ds**, **msg**, **msgbuf**, and **msginfo**, which provide (respectively) the data structure for interprocess messages, a structure for each message in the queue, the user message buffer for the **msgsnd** and **msgrcv** system calls, and a structure containing information about the state of the message queue.

For more information on interprocess communication, see **ipcs(ADM)**, **msgctl(S)**, **msgget(S)**, and **msgop(S)** in the *XENIX Programmer's Reference*.

param.h

Contains a number of parameters vital to the system: the system's adjustable parameters, priorities, signals, MMU (Memory Management Unit) constants, macros for unit conversion, and definitions of the fundamental constants of the implementation.

proc.h

Defines the structures **proc** and **xproc**, which, when they may be swapped out, hold all the vital information on processes.

reg.h

Defines constants that provide an index of the available registers relative to AX.

relysym.h

Provides definitions for the following structure types associated with the executable-file format:

asym	sym
bsym	xreloc
reloc	

relysym86.h

Contains the declarations for the 8086/80286 symbol table and relocation record structures. The structures **dosexec**, **desctab**, and **srel86** are defined. The **dosexec** structure is provided for MS-DOS support, **desctab** provides the structure of the descriptor table, and **srel86** provides the structure for segment relocation (which is necessary for medium- and large-model memory support).

sd.h

Defines the shared data table. See the following reference to *sdu.h*.



sdu.h

Defines values for the shared data flags, which are used by the shared data system calls:

sdenter	sdfree
sdget	sdgetv
sdleave	sdwaitv

For more information, see Subroutines(S) in the *XENIX Programmer's Reference*.

C Library Guide

sem.h

Defines the structures used by the semaphore operations system call, **semop**. The structures are as follows:

sem	seminfo
sembuf	sem_undo
semid_ds	

For detailed information, see **semop(S)** in the *XENIX Programmer's Reference*.

signal.h

Defines the values for the signal constants. For more information, see **signal(S)** in the *XENIX Programmer's Reference*.

sites.h

Provides values for system constants that are used in the structure defined in **utsname.h**.

stat.h

Defines the structure **stat**, which returns a structure to both the **stat** and **fstat** system calls. Also defines a number of constants.

sysinfo.h

Defines the structures **sysinfo** and **syswait**, which hold information about the state of the system and its processes.

sysmacros.h

Defines a number of machine-dependent macros.

sysm.h

Defines the structures **sysent** and **idt**, which define the format for the system-entry table and the interrupt descriptor table. It also defines a number of random variables and functions used by more than one routine.

text.h

Defines the structure **text**, which provides the format for text segments. It also defines a number of constants.

timeb.h

Defines the structure **timeb**, which is returned by the **ftime** system call. For more information, see **time(S)** in the *XENIX Programmer's Reference*.

times.h

Defines the structure **tms**, which is returned by the routine **times**. For more information, see **times(S)** in the *XENIX Programmer's Reference*.



ttold.h

Defines the structures **sgtty** and **tc**, which contain information for the **stty** and **gtty** system calls. It also defines the terminal modes.

tty.h

Defines the structures:

cblock **inter**
chead **tty**
clist

The **tty** structure formats the information for I/O for each character device. The remaining structures define a number of internal state variables and device commands.

C Library Guide

types.h

Defines the structure **saddr** and numerous machine-dependent variables.

ulimit.h

Defines values passed to the **ulimit** system call.

user.h

Defines the structure **user**, which contains all the data on a user process that doesn't need to be referenced (and is swapped with the process). The standard error codes are also redefined here.

utsname.h

Defines the structure **utsname**, which provides general information about system characteristics.

var.h

Defines the structure **var**.

Chapter 6

Using the

Standard I/O Functions

- 6.1 Introduction 6-1
 - 6.1.1 Preparing for the I/O Functions 6-1
 - 6.1.2 Special Names 6-1
 - 6.1.3 Special Macros 6-3
- 6.2 Using Command Line Arguments 6-4
- 6.3 Using the Standard Files 6-5
 - 6.3.1 Reading From the Standard Input 6-6
 - 6.3.2 Writing to the Standard Output 6-9
 - 6.3.3 Program Example 6-11
- 6.4 Using the Data Stream Functions 6-12
 - 6.4.1 Using File Pointers 6-13
 - 6.4.2 Opening a File 6-14
 - 6.4.3 Reading a Single Character 6-15
 - 6.4.4 Reading a String from a File 6-15
 - 6.4.5 Reading Records from a File 6-16
 - 6.4.6 Reading Formatted Data From a File 6-17
 - 6.4.7 Writing a Single Character 6-18
 - 6.4.8 Writing a String to a File 6-19
 - 6.4.9 Writing Formatted Output 6-19
 - 6.4.10 Writing Records to a File 6-20
 - 6.4.11 Testing for the End of a File 6-21
 - 6.4.12 Testing For File Errors 6-21
 - 6.4.13 Closing a File 6-22
 - 6.4.14 Program Example 6-22
- 6.5 Using More Data Stream Functions 6-24
 - 6.5.1 Using Buffered Input and Output 6-24
 - 6.5.2 Reopening a File 6-25
 - 6.5.3 Setting the Buffer 6-26
 - 6.5.4 Putting a Character Back into a Buffer 6-26

- 6.5.5 Flushing a File Buffer 6-27
- 6.6 Using the Low-Level Functions 6-28
- 6.7 Using File Descriptors 6-28
 - 6.7.1 Opening a File 6-29
 - 6.7.2 Reading Bytes From a File 6-30
 - 6.7.3 Writing Bytes to a File 6-30
 - 6.7.4 Closing a File 6-31
 - 6.7.5 Program Examples 6-31
 - 6.7.6 Using Random Access I/O 6-34
 - 6.7.7 Moving the Character Pointer 6-34
 - 6.7.8 Moving the Character Pointer in a Data Stream 6-35
 - 6.7.9 Rewinding a File 6-36
 - 6.7.10 Getting the Current Character Position 6-36
- 6.8 Controlling Terminal Lines Using `termio` and `ioctl()` 6-37
 - 6.8.1 Setting Serial Communications Parameters 6-41
 - 6.8.2 Parity Handling 6-41
 - 6.8.3 Maintaining `tty` Parameters 6-42

6.1 Introduction

Nearly all programs use some form of input and output. Some programs read from or write to files stored on disk. Others write to devices such as line printers. Many programs read from and write to the user's terminal. For this reason, the standard C library provides several predefined input and output functions that a programmer can use in programs.

This chapter explains how to use the I/O functions in the standard C library. In particular, it describes:

- Standard I/O Routines
- Command line arguments
- Standard input and output files
- Data “stream” functions for ordinary files
- Low-level functions for ordinary files
- Random access functions

6.1.1 Preparing for the I/O Functions

To use the standard I/O functions, a program must include the file *stdio.h*, which defines the needed macros and variables. To include this file, place the following line at the beginning of the program:

```
#include <stdio.h>
```

The actual functions are contained in the library file *libc.a*. This file is automatically read whenever you compile a program, so no special argument is needed when you invoke the compiler.

6.1.2 Special Names

The standard I/O library uses many names for special purposes. In general, these names can be used in any program that has included the *stdio.h* file. The following is a list of the special names:

<code>stdin</code>	The name of the standard input file.
<code>stdout</code>	The name of the standard output file.

C Library Guide

<code>stderr</code>	The name of the standard error file.
<code>EOF</code>	The value returned by the read routines on an end-of-file or an error.
<code>NULL</code>	The null pointer, returned by pointer-valued functions, to indicate an error.
<code>FILE</code>	The name of the file type used to declare pointers into data structures (data streams.)
<code>BUFSIZ</code>	The size in bytes (default is 1024) suitable for an I/O buffer supplied by the user.

The **stdin**, **stdout**, and **stderr** files are created automatically when a program is executed. Since the bulk of input and output of most programs is through the user's terminal, the system normally assigns **stdin** to the user's terminal keyboard and **stdout** and **stderr** to the user's terminal screen.

Every file opened for access by the I/O functions has a unique pointer associated with it called a file pointer. This pointer, defined with the predefined type **FILE** found in the *stdio.h* file, points to a structure that contains information about the file, such as the location of the buffer, the current character position in the buffer, and the status of the file (whether the file is being read or written to). The pointer can be given a valid pointer value with the **fopen**, **fdopen**, or **freopen** function, as described later in this chapter. Thereafter, the file pointer can be used to refer to that file until the file is explicitly closed with the **fclose** function.

Typically, a file pointer is defined with the following statement:

```
FILE *infile;
```

The standard input, output, and error files, like other opened files, have corresponding file pointers. Unlike other file pointers, the standard file pointers are predefined in the *stdio.h* file. This means a program can use these pointers to read and write from the standard files without first using the **fopen** function to open them.

The predefined file pointers are typically used when a program needs to alternate between the standard input or output file and an ordinary file. Although the predefined file pointers have type **FILE**, they are constants, not variables. They must not be assigned values.

Redirecting and Piping Input and Output

The XENIX system lets you redirect the standard input and output using the shell's redirection symbols. This allows a program to use other devices and files as sources of input and output instead of the terminal's keyboard and screen.

Use the XENIX redirection symbols (<, >) and the pipe symbol (|) to redefine the standard input and standard output for a particular program. The left angle bracket, <, instructs the shell to treat the named file as **stdin**, while the right angle bracket, >, instructs the shell to treat the named file as **stdout**. For example, the following command line opens the file *phonelist* as the standard input to the program *dial*:

```
dial <phonelist
```

If the file does not exist, the system displays an error message and stops the program. Similarly, the following command line directs the output of the program *dial* to the file *savephone*:

```
dial > savephone
```

If *savephone* does not already exist, the shell automatically creates it.

Use the pipe symbol, |, to direct the output of one program to the input of another. For example, the following command line connects the standard output of *dial* to the standard input of *wc*:

```
dial | wc
```

Note that the standard input of *dial* and the standard output of *wc* are not affected. When the program on the output side of a pipe terminates, the system automatically places the constant value EOF in the standard input of the program on the input side.

6.1.3 Special Macros

The functions **getc**, **getchar**, **putc**, **putchar**, **feof**, **ferror**, and **fileno** are actually macros, not functions. This means that you cannot redeclare them or use them as targets for a breakpoint when debugging.

C Library Guide

6.2 Using Command Line Arguments

The XENIX system lets you pass information to a program at the same time you invoke it for execution. You can do this with command line arguments.

A XENIX command line is the line you type to invoke a program. A command line argument is anything you type in a XENIX command line. A command line argument can be a filename, an option, or a number. The first argument in any command line must be the filename of the program you wish to execute.

When you enter a command line, the system reads the first argument and loads the corresponding program. It also counts the other arguments, stores them in memory in the same order in which they appear on the line, and passes the count and the locations to the main function of the program. The function can then access the arguments by accessing the memory in which they are stored.

To access the arguments, the main function must have two parameters: *argc*, an integer variable containing the argument count, and *argv*, an array of pointers to the argument values. You can define the parameters by using the lines:

```
main (argc, argv)
int argc;
char *argv[];
```

at the beginning of the main program function. When a program begins execution, *argc* contains the count, and each element in *argv* contains a pointer to one argument.

An argument is stored as a null-terminated string (i.e., a string ending with a null character, `\0`). The first string (at “*argv*[0]”) is the program name. The argument count is never less than 1, since the program name is always considered the first argument.

In the following example, the command line arguments are read and then echoed on the terminal screen. This program is similar to the XENIX **echo** command.

```
main(argc, argv)    /* echo arguments */
int argc;
char *argv[];
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%c", argv[i],
              (i<argc-1) ? ' ' : '\n');
}
```

In the example above, an extra space character is added at the end of each argument to separate it from the next argument. This is required, since the system automatically removes leading and trailing whitespace characters (i.e., spaces and tabs) when it reads the arguments from the command line. Adding a newline character to the last argument is for convenience only; it causes the shell prompt to appear on the next line after the program terminates.

When entering arguments on a command line, make sure each argument is separated from the others by one or more whitespace characters. If an argument must contain whitespace characters, enclose that argument in double quotation marks. For example, in the command line

```
display 3 4 "echo hello"
```

the string “echo hello” is treated as a single argument. Also, enclose in double quotation marks any argument that contains characters recognized by the shell (e.g., <, >, |, and ^).

You should not change the values of the *argc* and *argv* variables. If necessary, assign the argument value to another variable and change that variable instead. You can give other functions in the program access to the arguments by assigning their values to external variables.

6.3 Using the Standard Files

Whenever you invoke a program for execution, the XENIX system automatically creates a standard input, a standard output, and a standard error file to handle a program’s input and output needs. Since the bulk of input and output of most programs is through the user’s own terminal, the



C Library Guide

system normally assigns the user's terminal keyboard and screen as the standard input and output, respectively. The standard error file, which receives any error messages generated by the program, is also assigned to the terminal screen.

A program can read and write to the standard input and output files with the **getchar**, **gets**, **scanf**, **putchar**, **puts**, and **printf** functions. The standard error file can be accessed using the data streaming functions described in the section "Using Data Stream I/O" later in this chapter.

The XENIX system lets you redirect the standard input and output using the shell's redirection symbols. This allows a program to use other devices and files as its chief source of input and output in place of the terminal's keyboard and screen.

The following sections explain how to read from and write to the standard input and output. They also explain how to redirect the standard input and output.

6.3.1 Reading From the Standard Input

You can read from the standard input with the **getchar**, **gets**, and **scanf** functions.

The **getchar** function reads one character at a time from the standard input. The function call has the form:

```
c = getchar()
```

where *c* is the variable to receive the character. It must have **int** type. The function normally returns the character read, but will return the end-of-file value, EOF, if the end of a file or an error is encountered.

The **getchar** function is typically used in a conditional loop to read a string of characters from the standard input. For example, the following function reads *cnt* number of characters from the keyboard:

```
readn (p, cnt)
char p[ ];
int cnt;
{
    int i,c;

    i = 0;
    while ( i<cnt )
        if (( p[i++] = getchar() ) == EOF ) {
            p[i] = 0;
            return(EOF);
        }
    return(0);
}
```

Note that if **getchar** is reading from the keyboard, it waits for characters to be entered before returning.

The **gets** function reads a string of characters from the standard input and copies the string to a given memory location. The function call has the form:

```
gets (s)
```

where *s* is a pointer to the location to receive the string. The function reads characters until it finds a newline character, then replaces the newline character with a null character (N0) and copies the resulting string to memory. The function returns the null pointer value NULL if the end of the file or an error is encountered. Otherwise, it returns the value of *s*.

The function **gets** is typically used to read a full line from the standard input. For example, the following program fragment reads a line from the standard input, stores it in the character array *cmdln* and calls a function (called *parse*), if no error occurs:

```
char cmdln[SIZE];

if ( gets(cmdln) != NULL )
    parse();
```

In this case, the length of the string is assumed to be less than *SIZE*.

Note that **gets** cannot check the length of the string it reads, so overflow can occur.

C Library Guide

The **scanf** function reads one or more values from the standard input where a value may be a character string or a decimal, octal, or hexadecimal number. The function call has the form:

```
scanf (format, argptr . . .)
```

where *format* is a pointer to a string that defines the format of the values to be read and *argptr* is one or more pointers to the variables that will receive the values. There must be one *argptr* for each format given in the *format* string. The format may be *%s* for a string, *%c* for a character, and *%d*, *%o*, or *%x* for a decimal, octal, or hexadecimal number, respectively. (Other formats are described in **scanf(S)**, in the *XENIX Programmer's Reference*.) The function normally returns the number of values it read from the standard input, but it will return the value EOF if the end of the file or an error is encountered.

Unlike the **getchar** and **gets** functions, **scanf** skips all whitespace characters, reading only those characters which make up a value. It then converts the characters, if necessary, into the appropriate string or number.

The **scanf** function is typically used whenever formatted input is required (i.e., input that must be entered in a special way or that has a special meaning). For example, in the following program fragment, **scanf** reads both a name and a number from the same line:

```
char name[20];
int number;

scanf("%s %d", name, &number);
```

In this example, the string *%s %d* defines what values are to be read (a string and a decimal number). The string is copied to the character array *name* and the number to the integer variable *number*. Note that pointers to these variables are used in the call and not the actual variables themselves.

When reading from the keyboard, **scanf** waits for values to be entered before returning. Each value must be separated from the next by one or more whitespace characters (such as spaces, tabs, or even newline characters). For example, for the function

```
scanf("%s %d %c", name, age, sex);
```

an acceptable input is:

```
John 27  
M
```

If the value is a number, it must have the appropriate digits; that is, a decimal number must have decimal digits, octal numbers must have octal digits, and hexadecimal numbers must have hexadecimal digits.

If **scanf** encounters an error, it immediately stops reading the standard input. Before **scanf** can be used again, the illegal character that caused the error must be removed from the input using the **getchar** function.

You may use the **getchar**, **gets**, and **scanf** functions in a single program. Just remember that each function reads the next available character, making that character unavailable to the other functions.

Note that when the standard input is the terminal keyboard, the **getchar**, **gets**, and **scanf** functions usually do not return a value until at least one newline character has been entered. This is true even if only one character is desired. If you wish to have immediate input on a single keystroke, see the **raw** function call described in “Setting a Terminal Mode” in the “Screen Processing” chapter of this Guide.



6.3.2 Writing to the Standard Output

You can write to the standard output with the **putchar**, **puts**, and **printf** functions.

The **putchar** function writes a single character to the output buffer. The function call has the form:

```
putchar (c)
```

where *c* is the character to be written. The function normally returns the same character it wrote, but will return the value EOF if an error is encountered.

C Library Guide

The function is typically used in a conditional loop to write a string of characters to the standard output. For example, the following function writes *cnt* number of characters plus a newline character to the standard output:

```
written (p,cnt)
char p[ ];
int cnt;
{
    int i;

    for (i=0; i<=cnt; i++)
        putchar( (i != cnt) ? p[i] : '\n');
}
```

The **puts** function copies the string found at a given memory location to the standard output. The function call has the form:

```
puts (s)
```

where *s* is a pointer to the location containing the string. The string may be any number of characters, but must end with a null character (N). The function writes each character in the string to the standard output and replaces the null character at the end of the string with a newline character.

Since the function automatically appends a newline character, it is typically used when writing full lines to the standard output. For example, the following program fragment writes one of three strings to the standard output:

```
char c;
switch(c) {
    case '1':
        puts("Continuing...");
        break;
    case '2':
        puts("All done.");
        break;
    default:
        puts("Sorry, there was an error.");
}
```

The string to be written depends on the value of *c*.

Using the Standard I/O Functions

The **printf** function writes one or more values to the standard output where the value is a character string or a decimal, octal, or hexadecimal number. The function automatically converts numbers into the proper display format. The function call has the form:

```
printf(format[,arg] . . .)
```

where *format* is a pointer to a string which describes the format of each value to be written and *arg* is one or more variables containing the values to be written. There must be one *arg* for each format in the *format* string. The formats may be *%s* for a string, *%c* for a character, and *%d*, *%o*, or *%x* for a decimal, octal, or hexadecimal number, respectively. (Other formats are described in **printf(S)**, in the *XENIX Programmer's Reference*.) If a string is requested, the corresponding *arg* must be a pointer. The function normally returns zero, but will return a nonzero value if an error is encountered.

The **printf** function is typically used when formatted output is required (i.e., when the output must be displayed in a certain way). For example, you may use the function to display a name and number on the same line as in the following example.

```
char name [ ];
int number;

printf("%s %d", name, number);
```

In this example, the string *%s %d* defines the type of output to be displayed (a string and a number separated by a space). The output values are copied from the character array *name* and the integer variable *number*.

6

You may use the **putchar**, **puts**, and **printf** functions in a single program. Just remember that the output appears in the same order as it is written to the standard output.

6.3.3 Program Example

This section shows how you can use the standard input and output files to perform useful tasks. The **ccstrip** (for “control character strip”) program defined below strips out all ASCII control characters from its input except for newline and tab. You can use this program to display text or data files that contain characters that may disrupt your terminal screen.

C Library Guide

```
#include <stdio.h>

main() /* ccstrip: strip nth characters */
{
    int c;
    while ((c = getchar()) != EOF)
        if ((c >= ' ' && c < 0177) ||
            c == '\t' || c == '\n')
            putchar(c);
    exit(0);
}
```

You can strip and display the contents of a single file by changing the standard input of the **ccstrip** program to the desired file. The command line:

```
ccstrip < doc.t
```

reads the contents of the file *doc.t*, strips out control characters, then writes the stripped file to the standard output.

If you wish to strip several files at the same time, you can create a pipe between the **cat** command and **ccstrip**.

To read and strip the contents of the files *file1*, *file2*, and *file3*, and then display them on the standard output, enter the command:

```
cat file1 file2 file3 | ccstrip
```

If you wish to save the stripped files, you can redirect the standard output of **ccstrip**. For example, this command line writes the stripped files to the file *clean*:

```
cat file1 file2 file3 | ccstrip >clean
```

Note that the **exit** function is used at the end of the program to ensure that any program which executes the **ccstrip** program will receive a normal termination status (typically 0) from the program when it completes. An explanation of the **exit** function and how to execute one program under control of another is given in the “Writing and Using Pipes” chapter.

6.4 Using the Data Stream Functions

The functions described so far have all read from the standard input and written to the standard output. The next step is to show functions that access files not already connected to the program. One set of standard I/O functions allows a program to open and access ordinary files as if they were a “data stream” of characters. For this reason, the functions are called the data stream functions.

The term “stream” in this section refers to a data structure used in the `stdio` routines described here. It should not be confused with data structures of the same name in any other product or package.

Unlike the standard input and output files, a file to be accessed by a data stream function must be explicitly opened with the **fopen** function. The function can open a file for reading, writing, or appending. A program can read from a previously opened file with the **getc**, **fgetc**, **fgets**, **fgetw**, **fread**, and **fscanf** functions. It can write to a previously opened file with the **putc**, **fputc**, **fputs**, **fputw**, **fwrite**, and **fprintf** functions. A program can test for the end of the file or for an error with the **feof** and **ferror** functions. A program can close a file with the **fclose** function.

6.4.1 Using File Pointers

Every file opened for access by the data stream functions has a unique pointer called a file pointer associated with it. This pointer, defined with the predefined type `FILE`, found in the `stdio.h` file, points to a structure that contains information about the file, such as the location of the buffer (the intermediate storage area between the actual file and the program), the current character position in the buffer, and whether the file is being read or written. The pointer can be given a valid pointer value with the **fopen** function as described in the next section. (The `NULL` value, like `FILE`, is defined in the `stdio.h` file.) Thereafter, the file pointer may be used to refer to that file until the file is explicitly closed with the **fclose** function.

Typically, a file pointer is defined with the statement:

```
FILE *infile;
```

The standard input, output, and error files, like other opened files, have corresponding file pointers. These file pointers are named **stdin** for standard input, **stdout** for standard output, and **stderr** for standard error. Unlike other file pointers, the standard file pointers are predefined in the `stdio.h` file. This means that a program may use these pointers to read and write from the standard files without first using the **fopen** function to open them.

The predefined file pointers are typically used when a program needs to alternate between the standard input or output file and an ordinary file. Although the predefined file pointers have type `FILE`, they are constants, not variables. They must not be assigned values.



C Library Guide

6.4.2 Opening a File

The **fopen** function opens a given file and returns a pointer (called a file pointer) to a structure containing the data necessary to access the file. The pointer may then be used in subsequent data stream functions to read from or write to the file. See **fopen(S)** in the *XENIX Programmer's Reference*.

The function call has the form:

```
fp = fopen(filename, type)
```

where *fp* is the pointer to receive the file pointer, *filename* is a pointer to the name of the file to be opened and *type* is a pointer to a string that defines how the file is to be opened. The *type* string may be *r* for reading, *w* for writing, and *a* for appending, (open for writing at the end of the file).

A file may be opened for different operations at the same time if separate file pointers are used. For example, the following program fragment opens the file named */usr/accounts* for both reading and writing:

```
FILE *rp, *wp, *fopen();  
  
rp = fopen("/usr/accounts", "r");  
wp = fopen("/usr/accounts", "a");
```

Opening an existing file for writing destroys the old contents. Opening an existing file for appending leaves the old contents unchanged and causes any data written to the file to be appended to the end.

Trying to open a nonexistent file for reading causes an error. Trying to open a nonexistent file for writing or appending causes a new file to be created. Trying to open any file for which the program does not have appropriate permission causes an error.

The function normally returns a valid file pointer, but will return the value NULL if an error on opening the file is encountered. It is wise to check for the NULL value after each function call to prevent reading or writing after an error.

6.4.3 Reading a Single Character

The **getc** and **fgetc** functions return a single character read from a given file, and return the value EOF if the end of the file or an error is encountered. The function calls have the form:

```
c = getc (stream)
```

and

```
c = fgetc (stream)
```

where *stream* is the file pointer to the file to be read and *c* is the variable to receive the character. The return value is always an integer.

The functions are typically used in conditional loops to read a string of characters from a file. For example, the following program fragment continues to read characters from the file given to it by *infile* until the end of the file or an error is encountered:

```
int i;
char buf[MAX];
FILE *infile;
:
:
while ((c=getc(infile)) != EOF)
    buf[i++]=c;
```

The only difference between the functions is that **getc** is defined as a macro, and **fgetc** as a true function. This means that, unlike **getc**, **fgetc** may be passed as an argument in another function, used as a target for a breakpoint when debugging, or used to avoid any side effects of macro processing.

6

6.4.4 Reading a String from a File

The **fgets** function reads a string of characters from a file and copies the string to a given memory location. The function call has the form:

```
fgets (s, n, stream)
```

where *s* is a pointer to the location to receive the string, *n* is a count of the maximum number of characters to be in the string, and *stream* is the file pointer of the file to be read. The function reads *n-1* characters or up to the first newline character, whichever occurs first. The function appends a null character (N) to the last character read and then stores the string at the specified location. The function returns the null pointer value NULL if the end of the file or an error is encountered. Otherwise, it returns the pointer *s*.

C Library Guide

The function is typically used to read a full line from a file. For example, the following program fragment reads a string of characters from the file given by *myfile*.

```
char cmdln[MAX];
FILE *myfile;
:
:
if ( fgets( cmdln, MAX, myfile ) != NULL)
    parse( cmdln );
```

In this example, **fgets** copies the string to the character array *cmdln*.

6.4.5 Reading Records from a File

The **fread** function reads one or more records from a file and copies them to a given memory location. The function call has the form:

```
fread(ptr, size, nitems, stream)
```

where *ptr* is a pointer to the location to receive the records, *size* is the size (in bytes) of each record to be read, *nitems* is the number of records to be read, and *stream* is the file pointer of the file to be read. The *ptr* may be a pointer to a variable of any type (from a single character to a structure). The *size*, an integer, should give the numbers of bytes in each item you wish to read. One way to ensure this is to use the **sizeof** function on the pointer *ptr* (see the example below). The function always returns the number of records it read, regardless of whether or not the end of the file or an error is encountered.

The function is typically used to read binary data from a file. For example, the following program fragment reads two records from the file given by *database* and copies the records into the structure *person*.

```
# include <stdio.h>

#define dbname "dbfile"

typedef struct
{
    char name[20];
    int age;
} record;

main()
{
    FILE *database, *fopen();
    record person[2];

    if ((database = fopen(dbname,"w")) == NULL) {
        printf("Cannot open %s\n",dbname);
        exit(1);
    }
    fread(char * person, sizeof(record), 2, database);
    printf("record is %d\n",sizeof(record));
    printf("person is %d\n",sizeof(person));
}
```

Note that since **fread** does not explicitly indicate errors, the **feof** and **ferror** functions should be used to detect end of the file and errors. These functions are described later in this chapter.

6

6.4.6 Reading Formatted Data From a File

The **fscanf** function reads formatted input from a given file and copies it to the memory location given by the respective argument pointers, just as the **scanf** function reads from the standard input. The function call has the form:

```
fscanf (stream, format, argptr ...)
```

where *stream* is the file pointer of the file to be read, *format* is a pointer to the string that defines the format of the input to be read, and *argptr* is one or more pointers to the variables that are to receive the formatted input. There must be one *argptr* for each format given in the *format* string. The format may be *%s* for a string, *%c* for a character, and *%d*, *%o*, or *%x* for a decimal, octal, or hexadecimal number, respectively. (Other formats are

C Library Guide

described in **scanf** (S) in the *XENIX Programmer's Reference*.) The function normally returns the number of arguments it read, but will return the value EOF if the end of the file or an error is encountered.

The function is typically used to read files that contain both numbers and text. For example, this program fragment reads a name and a decimal number from the file given by *file*:

```
FILE *file;
int pay;
char name[20];
:
:
fscanf(file, "%s %d\n", name, &pay);
```

This program fragment copies the name to the character array *name* and the number to the integer variable *pay*.

6.4.7 Writing a Single Character

The **putc** and **fputc** functions write single characters to a given file. The function calls have the form:

```
putc (c, stream)
```

and

```
fputc (c, stream)
```

where *c* is the character to be written and *stream* is the file pointer to the file to receive the character. The function normally returns the character written, but will return the value EOF if an error is encountered.

The **putc** function is defined as a macro and may have undesirable side effects resulting from argument processing. In such cases, the equivalent function **fputc** should be used.

These functions are typically used in conditional loops to write a string of characters to a file. For example, the following program fragment writes characters from the array *name* to the file given by *out*.

```
FILE *out;
char name[MAX];
int i;

for (i=0; i<MAX; i++)
    fputc( name[i], out);
```

The only difference between the **putc** and **fputc** functions is that **putc** is defined as a macro and **fputc** as an actual function. This means that **fputc**, unlike **putc**, may be used as an argument to another function, as the target of a breakpoint when debugging, and to avoid the side effects of macro processing.

6.4.8 Writing a String to a File

The **fputs** function writes a string to a given file. The function call has the form:

```
fputs (s, stream)
```

where *s* is a pointer to the string to be written, and *stream* is the file pointer to the file.

The function is typically used to copy strings from one file to another. For example, in the following program fragment, **gets** and **fputs** are combined to copy strings from the standard input to the file given by *out*.

```
FILE *out;
char cmdln[MAX];

if ( gets( cmdln ) != EOF )
    fputs( cmdln, out);
```

The function normally returns zero, but will return EOF if an error is encountered.

6

6.4.9 Writing Formatted Output

The **fprintf** function writes formatted output to a given file, just as the **printf** function writes to the standard output. The function call has the form:

```
fprintf (stream, format [, arg ]...)
```

where *stream* is the file pointer of the file to be written to, *format* is a pointer to a string which defines the format of the output, and *arg* is one or more arguments to be written. There must be one *arg* for each format in the *format* string. The formats may be *%s* for a string, *%c* for a character, and *%d*, *%o*, or *%x* for a decimal, octal, or hexadecimal number, respectively. (Other formats are described in **printf(S)** in the XENIX Reference.) If a string is requested, the corresponding *arg* must be a pointer, otherwise, the actual variable must be used. The function

C Library Guide

normally returns zero, but will return a nonzero number if an error is encountered.

The function is typically used to write output that contains both numbers and text. For example, to write a name and a decimal number to the file given by *outfile*, use the following program fragment:

```
FILE *outfile;
int pay;
char name[20];
⋮
fprintf(outfile, "%s %d\n", name, pay);
```

The name is copied from the character array *name*, and the number from the integer variable *pay*.

6.4.10 Writing Records to a File

The **fwrite** function writes one or more records to a given file. The function call has the form:

```
fwrite (ptr, size, nitems, stream)
```

where *ptr* is a pointer to the first record to be written, *size* is the size (in bytes) of each record, *nitems* is the number of records to be written, and *stream* is the file pointer of the file. The *ptr* may point to a variable of any type (from a single character to a structure). The *size* should give the number of bytes in each item to be written. One way to ensure this is to use the **sizeof** function (see the example below). The function always returns the number of items actually written to the file whether or not the end of the file or an error is encountered.

The function is typically used to write binary data to a file. For example, the following program fragment writes two records to the file given by *database*.

```
FILE *database;
struct record {
    char name[20];
    int age;
} person[2];
⋮
fwrite((char*)person, sizeof(struct record), 2, database);
```

The records are copied from the structure *person*.

Since the function does not report the end of the file or errors, the **feof** and **ferror** functions should be used to detect these conditions.

6.4.11 Testing for the End of a File

The **feof** function returns the value -1 if a given file has reached its end. The function call has the form:

```
feof (stream)
```

where *stream* is the file pointer of the file. The function returns -1 only if the file has reached its end, otherwise it returns 0. The return value is always an integer.

The **feof** function is typically used after those functions whose return value is not a clear indicator of an end-of-file condition. For example, in the following program fragment the function checks for the end of the file after each character is read. The reading stops as soon as **feof** returns -1.

```
char name[10];
FILE *stream;
:
:
do
    fread( name, sizeof(name), 1, stream );
while(!feof( stream ));
```



6.4.12 Testing For File Errors

The **ferror** function tests a given data stream file for an error. The function call has the form:

```
ferror (stream)
```

where *stream* is the file pointer of the file to be tested. The function returns a nonzero (true) value if an error is detected, otherwise it returns zero (false). The function returns an integer value.

The function is typically used to test for errors before performing a subsequent read or write to the file. For example, in the following program fragment **ferror** tests the file given by *stream*.

C Library Guide

```
char *buf;
char x[5];

while ( !ferror(stream) )
    fread(buf, sizeof(x), 10, stream);
```

If it returns zero, the next item in the file given by *stream* is copied to *buf*. Otherwise, execution passes to the next statement.

Further use of a file after a error is detected may cause undesirable results.

6.4.13 Closing a File

The **fclose** function closes a file by breaking the connection between the file pointer and the structure created by **fopen**. Closing a file empties the contents of the corresponding buffer and frees the file pointer for use by another file. The function call has the form:

```
fclose (stream)
```

where *stream* is the file pointer of the file to close. The function normally returns 0, but will return -1 if an error is encountered.

The **fclose** function is typically used to free file pointers when they are no longer needed. This is important because usually no more than 60 files can be open at the same time. For example, the following program fragment closes the file given by *infile* when the file has reached its end:

```
FILE *infile;
if ( feof(infile) )
    fclose( infile );
```

Note that whenever a program terminates normally, the **fclose** function is automatically called for each open file, so no explicit call is required unless the program must close a file before its end. Also, the function automatically calls **fflush** to ensure that everything written to the file's buffer actually gets to the file.

6.4.14 Program Example

This section shows how you can use the data stream functions you have seen so far to perform useful tasks. The following program, which counts the characters, words, and lines found in one or more files, uses the **fopen**, **fprintf**, **getc**, and **fclose** functions to open, close, read, and write to the given files. The program incorporates a basic design that is common to other XENIX programs, namely it uses the filenames found in the

Using the Standard I/O Functions

command line as the files to open and read, or if no names are present, it uses the standard input. This allows the program to be invoked on its own, or be the receiving end of a pipe.

```
#include <stdio.h>
main(argc, argv) /* wc: count lines, words, chars */
int argc;
char *argv[];
{
    int c, i, inword;
    FILE *fp, *fopen();
    long linect, wordct, charct;
    long tlinect = 0, twordct = 0, tcharct = 0;

    i = 1;
    fp = stdin;
    do
    {
        if (argc > 1 &&
            (fp=fopen(argv[i], "r")) == NULL) {
            fprintf(stderr, "wc: can't open %s\n",
                argv[i]);
            continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
            charct++;
            if (c == '\n')
                linect++;
            if (c == ' ' || c == '\t' || c == '\n')
                inword = 0;
            else if (inword == 0) {
                inword = 1;
                wordct++;
            }
        }
        printf("%7ld %7ld %7ld", linect, wordct,
            charct);
        printf(argc > 1 ? " %s\n" : "\n", argv[i]);
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
    } while (++i < argc);
    if (argc > 2)
        printf("%7ld %7ld %7ld total\n", tlinect,
            twordct, tcharct);
    exit(0);
}
```

The program uses *fp* as the pointer to receive the current file pointer. Initially, this is set to *stdin* in case no filenames are present in the command line. If a filename is present, the program calls **fopen** and assigns the file pointer to *fp*. If the file cannot be opened (in which case **fopen** returns

C Library Guide

NULL), the program writes an error message to the standard error file *stderr* with the **fprintf** function. The function prints the format string “*wc: can't open %s*”, replacing the *%s* with the name pointed to by *argv[i]*.

Once a file is opened, the program uses the **getc** function to read each character from the file. As it reads characters, the program keeps a count of the number of characters, words, and lines. The program continues to read until the end of the file is encountered, that is, when **getc** returns the value EOF.

Once a file has reached its end, the program uses the **printf** function to display the character, word, and line counts on the standard output. The format string in this function causes the counts to be displayed as long decimal numbers with no more than 7 digits. The program then closes the current file with the **fclose** function and examines the command line arguments to see if there is another filename.

When all files have been counted, the program uses the **printf** function to display a grand total at the standard output, then stops execution with the **exit** function.

6.5 Using More Data Stream Functions

The data stream functions allow more control over a file than just opening, reading, writing, and closing. The functions also let a program take an existing file pointer and reassign it to another file (similar to redirecting the standard input and output files) as well as manipulate the buffer that is used for intermediate storage between the file and the program.

6.5.1 Using Buffered Input and Output

Buffered I/O is an input and output technique used by the XENIX system to cut down the time needed to read from and write to files. Buffered I/O lets the system collect the characters to be read or written, and then transfer them all at once rather than one character at a time. This reduces the number of times the system must access the I/O devices and consequently provides more time for running user programs. Not all files have buffers. For example, files associated with terminals, such as the standard input and output, are not buffered. This prevents unwanted delays when transferring the input and output. When a file does have a buffer, the buffer size in bytes is given by the manifest constant BUFSIZ, which is defined in the *stdio.h* file.

When a file has a buffer, the data stream functions read from and write to the buffer instead of the file. The system keeps track of the buffer and when necessary, fills it with new characters (when reading) or flushes (copies) it to the file (when writing). Normally, a buffer is not directly accessible to a program, however a program can define its own buffer for a file with the **setbuf** function. The function also lets a program change a buffered file to be an unbuffered one. The **ungetc** function lets a program put a character it has read back into the buffer, and the **fflush** function lets a program flush the buffer before it is full.

6.5.2 Reopening a File

The **freopen** function closes the file associated with a given file pointer, then opens a new file and gives it the same file pointer as the old file. The function call has the form:

```
freopen (newfile, type, stream)
```

where *newfile* is a pointer to the name of the new file, *type* is a pointer to the string that defines how the file is to be opened (*r* for read, *w* for writing, and *a* for appending), and *stream* is the file pointer of the old file. The function returns the file pointer *stream* if the new file is opened. Otherwise, it returns the null pointer value **NULL**.

The **freopen** function is used chiefly to attach the predefined file pointers *stdin*, *stdout*, and *stderr* to other files. For example, the following program fragment opens the file named by *newfile* as the new standard output file:

```
char *newfile;
FILE *nfile;

nfile = freopen(newfile, "w", stdout);
```

This has the same effect as using the redirection symbols in the command line of the program.



C Library Guide

6.5.3 Setting the Buffer

The **setbuf** function changes the buffer associated with a given file to the program's own buffer. It can also change the access to the file to no buffering. The function call has the form:

```
setbuf (stream, buf)
```

where *stream* is a file pointer and *buf* is a pointer to the new buffer, or is the null pointer value **NULL** if no buffering is desired. If a buffer is given, it must be **BUFSIZ** bytes in length, where **BUFSIZ** is a manifest constant found in *stdio.h*.

The function is typically used to create a buffer for the standard output when it is assigned to the user's terminal, thus, improving execution time by eliminating the need to write one character to the screen at a time. For example, the following program fragment changes the buffer of the standard output to the location pointed to by *p*:

```
char *p;  
  
p=malloc( BUFSIZ );  
setbuf ( stdout, p );
```

The new buffer is **BUFSIZ** bytes long.

The function may also be used to change a file from buffered to unbuffered input or output. Unbuffered input and output generally increase the total time needed to transfer large numbers of characters to or from a file, but give the fastest transfer speed for individual characters.

The **setbuf** function should be called immediately after opening a file and before reading or writing to it. Furthermore, the **fclose** or **fflush** function must be used to flush the buffer before terminating the program. If not used, some data written to the buffer may not be written to the file.

6.5.4 Putting a Character Back into a Buffer

The **ungetc** function puts a character back into the buffer of a given file. The function call has the form:

```
ungetc (c, stream)
```

where *c* is the character to put back and *stream* is the file pointer of the file. The function normally returns the same character it put back, but will return the value EOF if an error is encountered.

The function is typically used when scanning a file for the first character of a string of characters. For example, the following program fragment puts the first character that is not a whitespace character back into the buffer of the file given by *infile*, allowing the subsequent call to **gets** to read that character as the first character in the string:

```
FILE *infile;
char name[20];
:
while( isspace( c=getc(infile) ) )
;
  ungetc( c, stdin );
  gets( name, stdin );
```

Putting a character back into the buffer does not change the corresponding file; it only changes the next character to be read.

The function can only put a character back if one has been previously read. The function cannot put more than one character back at a time. This means that if three characters are read, then only the last character can be put back, never the first two.

The value EOF must never be put back in the buffer.

6.5.5 Flushing a File Buffer

6

The **fflush** function empties the buffer of a given file by immediately writing the buffer contents to the file. The function call has the form:

```
fflush (stream)
```

where **stream** is the file pointer of the file. The function normally returns zero, but will return the value EOF if an error is encountered.

The function is typically used to guarantee that the contents of a partially filled buffer are written to the file. For example, the following program fragment empties the buffer for the file given by *outtty* if the error condition given by *errflag* is 0.

```
FILE *outtty;
int errflag;
:
if (errflag == 0)
    fflush( outtty );
```

C Library Guide

Note that **fflush** is automatically called by the **fclose** function to empty the buffer before closing the file. This means that no explicit call to **fflush** is required if the file is also being closed.

The function ignores any attempt to empty the buffer of a file opened for reading.

6.6 Using the Low-Level Functions

The low-level functions provide direct access to files and peripheral devices. They are actually direct calls to the routines used in the XENIX operating system to read from and write to files and peripheral devices. The low-level functions give a program the same control over a file or device as the system, letting it access the file or device in ways that the data stream functions do not. However, low-level functions, unlike data stream functions, do not provide buffering or any other useful services of the data stream functions. This means that any program that uses the low-level functions has the complete burden of handling input and output.

The low-level functions, like the data stream functions, cannot be used to read from or write to a file until the file has been opened. A program may use the **open** function to open an existing or new file. A file can be opened for reading, writing, or appending.

Once a file is opened for reading, a program can read bytes from it with the **read** function. A program can write to a file opened for writing or appending with the **write** function. A program can close a file with the **close** function.

6.7 Using File Descriptors

Each file that has been opened for access by the low-level functions has a unique integer called a “file descriptor” associated with it. A file descriptor is similar to a file pointer in that it identifies the file. A file descriptor is unlike a file pointer in that it does not point to any specific structure. Instead, the descriptor is used internally by the system to access the necessary information. Since the system maintains all information about a file, the only way to access a file in a program is through the file descriptor.

There are three predefined file descriptors (just as there are three predefined file pointers) for the standard input, output, and error files. The descriptors are 0 for the standard input, 1 for the standard output, and 2 for the standard error file. As with predefined file pointers, a program may use the predefined file descriptors without explicitly opening the associated files.

Note that if the standard input and output files are redirected, the system changes the default assignments for the file descriptors 0 and 1 to the named files. This is also true if the input or output is associated with a pipe. File descriptor 2 normally remains attached to the terminal.

6.7.1 Opening a File

The **open** function opens an existing or new file and returns a file descriptor for that file. The function call has the form:

```
fd = open(name, access [,mode] );
```

where *fd* is the integer variable to receive the file descriptor, *name* is a pointer to a string containing the filename, *access* is an integer expression giving the type of file access, and *mode* is an integer number giving a new file's permissions. The function normally returns a file descriptor (a positive integer), but will return -1 if an error is encountered.

The *access* expression is formed by using one or more of the following manifest constants: **O_RDONLY** for reading, **O_WRONLY** for writing, **O_RDWR** for both reading and writing, **O_APPEND** for appending to the end of an existing file, and **O_CREAT** for creating a new file. (Other constants are described in **open(S)** in the *XENIX Programmer's Reference*.) The logical OR operator (**|**) may be used to combine the constants. The *mode* is used only if **O_CREAT** is given. For example, in the following program fragment, the function is used to open the existing file named */usr/accounts* for reading, and create the new file named */usr/tmp/scratch* for writing:

```
int in, out;  
  
in = open( "/usr/accounts", O_RDONLY );  
out = open( "/usr/tmp/scratch", O_WRONLY | O_CREAT, 0755 );
```

In the XENIX system, each file has 9 bits of protection information which control read, write, and execute permission for the owner of the file, for the owner's group, and for all others. A three-digit octal number is the most convenient way to specify the permissions. In the example above, the octal number *0755* specifies read, write, and execute permission for the owner, read and execute permission for the group, and read and execute permission for everyone else.

Note that if **O_CREAT** is given and the file already exists, the function destroys the file's old contents.

C Library Guide

6.7.2 Reading Bytes From a File

The **read** function reads one or more bytes of data from a given file and copies them to a given memory location. The function call has the form:

```
n_read = read(fd, buf, n);
```

where *n_read* is the variable to receive the count of bytes actually read, *fd* is the file descriptor of the file, *buf* is a pointer to the memory location to receive the bytes read, and *n* is a count of the desired number of bytes to be read. The function normally returns the same number of bytes as requested, but will return fewer if the file does not have that many bytes left to be read. The function returns 0 if the file has reached its end, or -1 if an error is encountered.

When the file is a terminal, **read** normally reads only up to the next newline.

The number of bytes to be read is arbitrary. The two most common values are 1, which means one character at a time, and 512, which corresponds to the physical block size on many peripheral devices.

6.7.3 Writing Bytes to a File

The **write** function writes one or more bytes from a given memory location to a given file. The function call has the form:

```
n_written = write(fd, buf, n);
```

where *n_written* is the variable to receive a count of bytes actually written, *fd* is the file descriptor of the file, *buf* is the name of the buffer containing the bytes to be written, and *n* is the number of bytes to be written.

The function always returns the number of bytes actually written. It is considered an error if the return value is not equal to the number of bytes requested to be written.

The number of bytes to be written is arbitrary. The two most common values are 1, which means one character at a time and 512, which corresponds to the physical block size on many peripheral devices.

6.7.4 Closing a File

The **close** function breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. The function call has the form:

```
close (fd)
```

where *fd* is the file descriptor of the file to close. The function normally returns 0, but will return -1 if an error is encountered.

The function is typically used to close files that are no longer needed. For example, the following program fragment closes the standard input if the argument count is greater than 1.

```
if (argc >1)
    close( 0 );
```

Note that all open files in a program are closed when a program terminates normally or when the **exit** function is called, so no explicit call to **close** is required.

6.7.5 Program Examples

This section shows how to use the low-level functions to perform useful tasks. It presents three examples that incorporate the functions as the sole method of input and output.

6

The first program copies its standard input to its standard output:

```
#define      BUFSIZE      512

main()      /* copy input to output */
{
    char  buf[ BUFSIZE ];
    int   n;
    while ((n = read( 0, buf, BUFSIZE )) > 0)
        write(1, buf, n);
    exit(0);
}
```

The program uses the **read** function to read **BUFSIZE** bytes from the standard input (file descriptor 0). It then uses **write** to write the same number of bytes it read to the standard output (file descriptor 1). If the standard input file size is not a multiple of **BUFSIZE**, the last **read** returns a smaller number of bytes to be written by **write**, and the next call to **read**

C Library Guide

returns zero.

This program can be used like a copy command to copy the content of one file to another. You can do this by redirecting the standard input and output files.

The second example shows how the **read** and **write** functions can be used to construct higher level functions like **getchar** and **putchar**. For example, the following is a version of **getchar** that performs unbuffered input:

```
#define      CMASK 0377
            /* for making chars > 0 */

getchar()
            /* unbuffered single character input */
{
    char c;
    return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

The variable *c* must be declared **char**, because **read** accepts a character pointer. In this case, the character being returned must be masked with octal *0377* to ensure that it is positive; otherwise sign extension may make it negative.

The second version of **getchar** reads input in large blocks, but hands out the characters one at a time:

```
#define      CMASK 0377
            /* for making char's > 0 */
#define      BUFSIZE 512

getchar()  /* buffered version */
{
    static char buf[BUFSIZE];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0) { /* buffer is empty */
        n = read(0, buf, BUFSIZE);
        bufp = buf;
    }
    return((--n >= 0) ?
           *bufp++ & CMASK : EOF);
}
```

Again, each character must be masked with the octal constant *0377*.

Using the Standard I/O Functions

The final example is a simplified version of the XENIX utility, **cp**, a program that copies one file to another. The main simplification is that this version copies only one file, and does not permit the second argument to be a directory.

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner,
                   R for group, others */

main(argc, argv) /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
    int    f1, f2, n;
    char  buf[ BUFSIZE ];

    if (argc != 3)
        error("Usage: cp from to", NULL);
    if ((f1 = open(argv[1], O_RDONLY)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = open(argv[2], O_CREAT | O_WRONLY,
                  PMODE)) == -1)
        error("cp: can't create %s", argv[2]);

    while ((n = read(f1, buf, BUFSIZE)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error", NULL);
    exit(0);
}

error(s1, s2)
/*
 * print error message and die
 */
char *s1, *s2;
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}
```

There is a limit (usually 60) to the number of files that a program may have open simultaneously. Therefore, any program that intends to process many files must be designed to reuse file descriptors by closing unneeded files.

C Library Guide

6.7.6 Using Random Access I/ O

Input and output operations on any file are normally sequential. This means each read or write takes place at the character position immediately after the last character read or written. The standard library, however, provides a number of data stream and low-level functions that allow a program to access a file randomly; that is, to exactly specify the position it wishes to read from or write to next.

The functions that provide random access operate on a file's "character pointer." Every open file has a character pointer that points to the next character to be read from that file, or the next place in the file to receive a character. Normally, the character pointer is maintained and controlled by the system, but the random access functions let a program move the pointer to any position in the file.

6.7.7 Moving the Character Pointer

The `lseek` function, a low-level function, moves the character pointer in a file opened for low-level access to a given position. The function call has the form:

```
lseek(fd, offset, origin);
```

where *fd* is the file descriptor of the file, *offset* is the number of bytes to move the character pointer, and *origin* is the number that gives the starting point for the move. It may be 0 for the beginning of the file, 1 for the current position, and 2 for the end.

For example, the following call forces the current position in the file, whose descriptor is 3, to move to the 512th byte from the beginning of the file:

```
lseek( 3, (long)512, 0 )
```

Subsequent reading or writing will begin at that position. Note that *offset* must be a long integer and *fd* and *origin* must be integers.

Using the Standard I/O Functions

The function may be used to move the character pointer to the end of a file to allow appending, or to the beginning as in a rewind function. For example, the call:

```
lseek (fd, (long)0, 2);
```

prepares the file for appending, and:

```
lseek (fd, (long)0, 0);
```

rewinds the file (moves the character pointer to the beginning). Notice the “(long)0” argument; it could also be written as:

```
0L
```

Using **lseek**, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file:

```
get (fd, pos, buf, n)
    /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
    lseek (fd, pos, 0); /* get to pos */
    return (read (fd, buf, n));
}
```

6.7.8 Moving the Character Pointer in a Data Stream



The **fseek** function, a data stream function, moves the character pointer in a file to a given location. The function call has the form:

```
fseek (stream, offset, ptrname)
```

where *stream* is the file pointer of the file, *offset* is the number of characters to move to the new position (it must be a long integer), and *ptrname* is the starting position in the file of the move (it must be 0 for beginning, 1, for current position, or 2 for end of the file). The function normally returns zero, but will return the value EOF if an error is encountered.

For example, the following program fragment moves the character pointer to the end of the file given by *stream*.

```
FILE *stream;
fseek (stream, (long)0, 2);
```

The function may be used on either buffered or unbuffered files.

C Library Guide

6.7.9 Rewinding a File

The **rewind** function, a data stream function, moves the character pointer to the beginning of a given file. The function call has the form:

```
rewind (stream)
```

where *stream* is the file pointer of the file. The function is equivalent to the following function call:

```
fseek (stream, 0L, 0);
```

It is chiefly used as a more readable version of the call.

6.7.10 Getting the Current Character Position

The **ftell** function, a data stream function, returns the current position of the character pointer in the given file. The returned position is always relative to the beginning of the file. The function call has the form:

```
p = ftell (stream)
```

where *stream* is the file pointer of the file and *p* is the variable to receive the position. The return value is always a long integer. The function returns the value -1 if an error is encountered.

The function is typically used to save the current location in the file so that the program can later return to that position. For example, the following program fragment first saves the current character position in *oldp*, then restores the file to this position if the current character position is greater than 800.

```
FILE *outfile;  
long oldp;  
:  
oldp = ftell( outfile );  
:  
if ((ftell( outfile )) > 800)  
    fseek(outfile, oldp, 0);
```

The **ftell** function is identical to the function call

```
lseek( fd, (long)0, 1)
```

where *fd* is the file descriptor of the given data stream file.

6.8 Controlling Terminal Lines Using `termio` and `ioctl()`

Normal XENIX terminal input is done on a line by line basis. When a process issues a read request to the XENIX tty driver, the driver gathers input characters until a newline or carriage return character is received, then passes the entire line to the calling process. However, applications programs often need to obtain each character as it is sent, without waiting for the newline character. Programs may change the handling of input and output characters using the `ioctl(S)` function. This function allows a program to change the terminal's default input configuration from a line at a time (called "canonical" or "cooked" input), to one character at a time (often called "raw" input), as well as mapping carriage return and linefeed characters. These functions apply to all types of terminal oriented interfaces including serial ports, the XENIX console multi-screens, and network tty connections. If a terminal's port (or "line") is a serial device, the program may also control the serial communication parameters of baud rate, parity, and data bits.

All the tty control parameters are found in the structure called "termio." This structure is defined in the file `/usr/include/termio.h`. The current values of this structure are retrieved using the `ioctl(S)` function call. The first parameter to the `ioctl` call is an open file descriptor that must be open as a tty device. The second parameter is the `ioctl` command to the tty driver (defined below), and the third parameter must be a pointer to a `termio` structure. This structure is used by the tty driver to either store or retrieve the current parameters. See `termio(S)` in the *XENIX Programmer's Reference* for more specific information.

Valid `ioctl` commands to the tty driver are:

TCGETA	Gets the current values of the tty parameters, and store them into the <code>termio</code> structure pointed to by the third argument.
TCSETA	Immediately change the tty parameters to those given by the <code>termio</code> structure. <code>TCSETA</code> can cause data corruption if parameter changes affect output and there are characters waiting to be sent. Using <code>TCSETAW</code> will eliminate this possibility.
TCSETAW	Waits for all pending output characters to be sent, then changes the tty parameters to those given by the <code>termio</code> structure. This command should be used whenever the parameter changes may affect output.
TCSETAF	Waits for all pending output to be sent, then flushes the input queue and sets the new parameters.

C Library Guide

The following program fragment demonstrates how to retrieve the current termio settings from the tty device open through the file descriptor `fd`:

```
#include <termio.h>
struct termio buf
int fd;
:
:
ioctl(fd, TCGETA, &buf );
```

Note that `fd` must be opened with `open()` before the `ioctl` statement. Also, note that since raw i/o is most often used on stdin, the most common usage of this `ioctl` is with the value 0 in the place of `fd`. Each of the configuration parameters is represented by bits in one of the four termio integer fields. Parameters which have only on or off settings are represented by a single bit. Those that have more than two possible values, such as baud rate, are represented as a binary number in a selected set of bits. The “`c_iflag`” field contains input related parameters. “`c_oflag`” contains parameters that affect output. “`c_cflag`” contains serial communication parameters, and “`c_lflag`” contains parameters that control the behavior of the tty driver itself. In addition to these four fields, the termio structure also contains the “`c_line`” field, which selects the current line discipline, and “`c_cc,`” a character array containing the characters for end of file, end of line, interrupt, etc. A complete description of control modes may be found in `termio(M)` in the *XENIX User's Reference*.

In order to receive individual character input, the program must turn off the ICANON bit in the `c_lflag` field. When this bit is off, the meanings of the characters in `c_cc[4]` and `c_cc[5]` change. With ICANON on, these characters denote the end of file and end of line characters. Their default values are Ctrl-D and NULL, respectively. However, with ICANON off, these characters change to represent VMIN and VTIME, respectively. VMIN denotes the minimum number of characters that must be input before a read function will return. If VTIME is greater than 0, then a read call returns if VTIME tenths of seconds elapse and at least one character has been read. Since the value of VMIN is initially 4, (the value of the end of file character Ctrl-D, left over from canonical mode) the program must change it to 1 if single character reads are desired.

While the ICANON bit is turned off, the tty driver still checks each incoming character for the interrupt, quit, stop, and start characters, and acts on them appropriately. If the process does not want these checks performed, the ISIG bit in `c_lflag` should be set to 0. If you do not want each character echoed, the ECHO bit of `c_lflag` should be set to 0. The IXOFF bit should be set to 0 if you do not want the STOP character to halt tty output.

Using the Standard I/O Functions

In addition to the ICANON parameter, the following parameters are also commonly modified by applications programs:

- ISIG If this bit is set, the tty driver checks each incoming character against the special characters in `c_cc`, such as the interrupt and quit characters. If a match is found, the appropriate action is taken, and the special character is removed from the input data stream. For instance, if the interrupt character is found, a SIGINT signal is sent to all processes associated with that tty device. If a program wants to keep the user from interrupting the program, or the program is transferring binary data, then ISIG should be set to 0.
- ECHO If the ECHO bit is set, all characters read in are echoed back to the terminal. Turn this bit off if the process desires input without echoing.
- IXOFF If this bit is set (its default setting), then output from the program can be paused when the Ctrl-S character is received. Output is then stopped until the Ctrl-Q character is received. If the IXANY parameter is set to 1, then any character may restart paused output.
- ICRNL Carriage return characters (ASCII 13) in the input data stream are changed to newline characters (ASCII 11) if this bit is set to 1, its default setting. Applications that need to distinguish between the newline and carriage return characters should turn off this bit.
- ONLCR This parameter is in the `c_oflag` field. If set to 1, ONLCR causes a carriage return and newline character pair to be output for each newline character sent by the program.

C Library Guide

The following program demonstrates how to set up the tty driver as discussed above:

```
#include <stdio.h>
#include <termio.h>
#include <fcntl.h>

main()
{
    char ch;
    int fd;
    struct termio tio, old_tio;

    /* get file descriptor to controlling terminal */
    fd = open( "/dev/tty", O_RDWR );

    /* get original tty settings to restore later */
    ioctl( fd, TCGETA, &old_tio );

    /* get current modes again to alter */
    ioctl( fd, TCGETA, &tio );

    /* turn off line by line mode */
    tio.c_iflag &= ~ICANON;

    /* turn off character echo, special */
    /* character checking, and flow control */
    tio.c_lflag &= ~(ISIG|ECHO|IXOFF);

    /* set minimum characters to receive */
    /* to 1, and minimum timeout to 0 */
    tio.c_cc[VMIN] = 1;
    tio.c_cc[VTIME] = 0;

    /* send new settings to tty driver */
    ioctl( fd, TCSETA, &tio );

    while ((ch = getchar()) != 'q' )
        printf( "Received character %d\n", ch );
}

/* restore original tty parameters */
/* Use TCSETAW instead of TCSETA to prevent */
/* corruption of any pending output */
ioctl( fd, TCSETAW, &old_tio );

exit(0);
}
```

6.8.1 Setting Serial Communications Parameters

In addition to controlling character input, output, and translation, the `termio` structure also controls the serial communications parameters of baud rate, number of data bits, and parity. The baud rate is controlled by four bits of the `c_cflag` field of the `termio` structure. These four bits allow for 16 different baud rates. Each baud rate bit pattern is represented by the constants `B50` (50 baud) through `B9600` (9600 baud). Two external baud rate constants, `EXTA`, and `EXTB`, are also defined. The meanings of these settings are specific to the serial driver. All of the baud rate constants are defined in the system include file `termio.h`. The constant `CBAUD` represents all the baud rate bits, and can be used to mask off the current baud rate setting while leaving all other bits in the field unchanged. The following program fragment demonstrates how to set the baud rate of a serial line to 2400 baud using `CBAUD` when the current baud rate is unknown:

```
#include <termio.h>

struct termio buf
int fd;
:
/* fd must be a valid file descriptor open to a tty device */
ioctl( fd, TCGETA, &buf );
buf.c_cflag = (buf.c_cflag & ~CBAUD) | B2400;
ioctl( fd, TCSETA, &buf);
```

The constants `CS5`, `CS6`, `CS7`, and `CS8` select five through eight data bits, respectively. The constant `CSIZE` is a mask for the data bit values, and can be used in the same way as `CBAUD` in the example above. The `CSTOPB` bit controls the number of stop bits in each byte. If set to 1, two stop bits are used, otherwise, one stop bit is used.

6.8.2 Parity Handling

Parity checking and generation is handled by the `PARENB` and `PARODD` bits of the `c_lflag` field, and the `INPCK`, `PARMRK`, and `IGNPAR` bits of the `c_iflag` field. If the `PARENB` bit is set to 0, both parity generation on output characters and parity checking on input characters are disabled. If `PARENB` is 1, parity generation on output is enabled, and input parity checking is controlled by the other four parameters. `PARODD` selects odd parity if set to 1, and even parity if set to 0. If `INPCK` is 0, no parity checking is done on the input data stream. This allows parity generation on output without affecting input. If `INPCK` and `IGNPAR` are both set to 1, then characters with parity errors are completely removed from the

C Library Guide

input data stream. If INPCK is 1 but IGNPAR is 0, the results of a parity error is controlled by PARMRK. If this parameter is 1, a parity error on input is indicated by the three character sequence 0377, 0, and X, where X is the erroneous character. If PARMRK is 0, parity errors are indicated by the receipt of a null (ASCII 0) character.

6.8.3 Maintaining tty Parameters

When a tty device is first opened, it has all of the parameters set to default values. The process may then change these values using the *ioctl()* call. However, these changes will stay in effect only as long as the device is open. When all processes have closed the device, all parameters are restored to their defaults.

Chapter 7

Screen Processing

- 7.1 Introduction 7-1
 - 7.1.1 Terminal Capability Descriptions 7-1
 - 7.1.2 Screen-Processing Overview 7-2
 - 7.1.3 Using the Library 7-3
 - 7.1.4 termcap curses Using /etc/termcap 7-6
 - 7.1.5 termcap curses Using terminfo 7-6
 - 7.1.6 terminfo curses Using terminfo 7-6
 - 7.1.7 Some Additional Notes 7-7

- 7.2 Preparing the Screen 7-7
 - 7.2.1 Initializing the Screen 7-7
 - 7.2.2 Using Terminal Capability and Type 7-8
 - 7.2.3 Using Default Terminal Modes 7-9
 - 7.2.4 Using Default Window Flags 7-9
 - 7.2.5 Using the Default Terminal Size 7-10
 - 7.2.6 Terminating Screen Processing 7-10

- 7.3 Using the Standard Screen 7-11
 - 7.3.1 Adding a Character 7-11
 - 7.3.2 Adding a String 7-12
 - 7.3.3 Printing Strings, Characters, and Numbers 7-12
 - 7.3.4 Reading a Character from the Keyboard 7-13
 - 7.3.5 Reading a String from the Keyboard 7-14
 - 7.3.6 Reading Strings, Characters, and Numbers 7-15
 - 7.3.7 Moving the Current Position 7-16
 - 7.3.8 Inserting a Character 7-16
 - 7.3.9 Inserting a Line 7-17
 - 7.3.10 Deleting a Character 7-17
 - 7.3.11 Deleting a Line 7-18
 - 7.3.12 Clearing the Screen 7-18
 - 7.3.13 Clearing a Part of the Screen 7-19
 - 7.3.14 Refreshing from the Standard Screen 7-19

- 7.4 Creating and Using Windows 7-20
 - 7.4.1 Creating a Window 7-20
 - 7.4.2 Creating a Subwindow 7-21

- 7.4.3 Accessing Window Structure 7-22
- 7.4.4 Adding and Printing to a Window 7-23
- 7.4.5 Reading and Scanning for Input 7-24
- 7.4.6 Moving the Current Position in a Window 7-26
- 7.4.7 Inserting Characters and Lines 7-27
- 7.4.8 Deleting Characters and Lines 7-28
- 7.4.9 Clearing the Window Screen 7-28
- 7.4.10 Saving from a Window 7-30
- 7.4.11 Refreshing from a Window 7-30
- 7.4.12 Overlaying Windows 7-31
- 7.4.13 Overwriting a Screen 7-32
- 7.4.14 Moving a Window 7-32
- 7.4.15 Reading a Character from a Window 7-33
- 7.4.16 Touching a Window 7-34
- 7.4.17 Deleting a Window 7-34

- 7.5 Using Other Window Functions 7-34
 - 7.5.1 Drawing a Box 7-35
 - 7.5.2 Displaying Bold Characters 7-35
 - 7.5.3 Restoring Normal Characters 7-36
 - 7.5.4 Getting the Current Position 7-37
 - 7.5.5 Setting Window Flags 7-38
 - 7.5.6 Scrolling a Window 7-39

- 7.6 Combining Movement with Action 7-40

- 7.7 Controlling the Terminal 7-40
 - 7.7.1 Setting a Terminal Mode 7-40
 - 7.7.2 Clearing a Terminal Mode 7-41
 - 7.7.3 Moving the Terminal's Cursor 7-42
 - 7.7.4 Getting the Terminal Mode 7-43
 - 7.7.5 Saving and Restoring the Terminal Flags 7-43
 - 7.7.6 Setting a Terminal Type 7-43
 - 7.7.7 Reading the Terminal Name 7-44

- 7.8 Advanced Topics 7-44
 - 7.8.1 Multiple Attributes 7-44
 - 7.8.2 Saving and Restoring tty Settings 7-45
 - 7.8.3 Output Mapping Features 7-45

7.1 Introduction

This chapter explains how to use the **curses** and **terminfo** screen processing libraries. These libraries provide functions to create and update screen windows, get input from the terminal in a screen-oriented way, and optimize the motion of the cursor on the screen.

The **curses** library is the standard screen processing library provided with this and previous versions of XENIX. The **terminfo(S)** library is a new library that provides the same functions as the **curses** library plus the following additional screen processing capabilities:

- soft-label control
- termcap conversion
- character attribute control
- function key return values
- line-graphic drawing

In addition, the **terminfo(M)** terminal capabilities database file can be redefined by the terminal capabilities compiler, **tic**. For more information, see **tic (M)** in the *XENIX User's Reference*.

7.1.1 Terminal Capability Descriptions

There are two different versions of the **curses** library distributed with the XENIX system. The principal difference between the two versions is that each draws its terminal descriptions from a different terminal capability database.

The **termcap curses** is the original XENIX version of **curses**. It is designed to use the *etc/termcap* database of terminal descriptions. **termcap** is described in the **termcap(M)** manual page. You may, however, use the **terminfo** terminal capability database instead of **termcap**.

The **terminfo curses** library is a recently developed compatible version of **curses** with extended functionality. It is designed to use the **terminfo** database of terminal descriptions. This database is described in the **terminfo(M)** manual page. It is not possible to use *etc/termcap* with **terminfo curses**.

This chapter primarily discusses **termcap curses**. Since **terminfo curses** is an extended, yet compatible, version of **curses**, this chapter also describes the basic **terminfo curses** routines.

C Library Guide

The **termcap** **courses** routines are summarized in the **courses(S)** manual page. The **terminfo** **courses** routines are completely summarized in the **terminfo(S)** manual page. The extensions provided by **terminfo** **courses** over **termcap** **courses** are not described in this chapter.

The **terminfo** **courses** package available with XENIX includes a number of extensions over other versions of **courses**. These extensions provide for superior handling of typeahead and function keys. These extensions require that programs using **terminfo** **courses** be compiled with the XENIX extensions library (**-lx**).

7.1.2 Screen-Processing Overview

Screen processing gives a program a simple and efficient way to use the capabilities of the terminal attached to the program's standard input files and output files. Screen processing does not rely on the terminal's type. Instead, the screen-processing functions use the following XENIX terminal capability files to tailor their actions for any given terminal.

This makes a screen-processing program terminal independent. The program can be run with any terminal as long as that terminal is described in the appropriate terminal capability file. Programs using terminal functions must use the proper terminal capability file.

For **courses** programs, use

/etc/termcap

For **terminfo** programs, use

/usr/lib/terminfo

The screen-processing functions access a terminal screen by working through intermediate screens and windows in memory.

- A screen represents what the entire terminal screen should look like. A screen can be composed of one or more windows.
- A window represents what some portion of the terminal screen should look like. A window can be as small as a single character or as large as an entire screen.

Before a screen or window can be used, it must be created using the **newwin** or **subwin** function. These functions define the size of the screen or window in terms of lines and columns.

Each position in a screen or window represents a place for a single character and corresponds to a similar place on the terminal screen. Positions are numbered according to line and column. For example, the position in the upper-left corner of a screen or window is numbered (0,0) and the position immediately to its right is (0,1).

A typical screen has 24 lines and 80 columns. Its upper-left corner corresponds to the upper-left corner of the terminal screen. A window, on the other hand, can be any size (within the limits of the actual screen). Its upper-left corner can correspond to any position on the terminal screen. For convenience, the **initscr** function, which initializes a program for screen processing, also creates a default screen, **stdscr** (standard screen). The **stdscr** can be used without first creating it explicitly with the **newwin** or **subwin** function. The function also creates **curscr** (current screen), which contains a copy of what is currently on the terminal screen.

To display characters on the terminal screen, a program must write these characters to a screen or window using screen-processing functions, such as **addch** and **waddch**. If necessary, a program can move to the desired position in the screen or window by using the **move** and **wmove** functions. Once characters are added to a screen or window, the program can copy the characters to the terminal screen by using the **refresh** or **wrefresh** function. These functions update the terminal screen according to what has changed in the given screen or window. Since the terminal screen is not changed until a program calls **refresh** or **wrefresh**, a program can maintain several different windows, each containing different characters for the same portion of the terminal screen. The program can choose which window should actually be displayed before updating.

A program can continue to add new characters to a screen or window as needed and edit these characters by using functions such as **insertln**, **deleteln**, and **clear**. A program can also combine windows to make a composite screen using the **overlay** and **overwrite** functions. In each case, the **refresh** or **wrefresh** function is used to copy the changes to the terminal screen.

7.1.3 Using the Library

To execute library functions in a program, you must declare the library's include file in your program, and then link the library's object code to your compiled program. The include file contains data types and variables used in the library functions.

C Library Guide

Declaring the include File

To declare the library's include file, add the appropriate include statement to the beginning of your program.

For **curses** programs, use

```
#define M_TERMCAP
#include <curses.h>
```

For **terminfo** programs, use

```
#define M_TERMINFO
#include <curses.h>
```

Specifying the Screenprocessing Library

To link your compiled program with the library functions, specify the screen-processing library file on the C compiler command line when you start the compiler.

For **curses** programs, type

```
cc files -lcurses -ltermcap
```

For **terminfo** programs, type

```
cc files -ltinfo -lx
```

where

- *files* are the names of the files compiled, and linked with the library functions.
- **-l** specifies the library filename linked with the compiled program. Note that **-l** requires you to omit the *lib* portion of the library filename.

The screen-processing functions are contained in following libraries:

For **curses** library functions, use

```
libcurses.a
libtermcap.a
```

For **terminfo** library functions, use

libtinfo.a

For example, following the command line compiles the files *main.c* and *intf.c* and copies the executable program to the file *sample* after linking the **curses** screen-processing-library files to the program:

```
cc main.c intf.c -lcurses -ltermcap -o sample
```

The **curses** and **terminfo** screen-processing libraries contain a variety of predefined names. These names refer to variables, manifest constants, and types that can be used with the library functions. The following table lists these names:

Table 7.1
Screen Processing Special Names

Type	Name	Description
WINDOW*	<i>curscr</i>	A pointer to the current version of the terminal screen
WINDOW*	<i>stdscr</i>	A pointer to the default screen used for updating when no explicit screen is defined
char*	<i>Def_term</i>	A pointer to the default terminal type if the type cannot be determined
bool	<i>My_term</i>	The terminal type flag; if set, it causes the terminal specification in <i>Def_term</i> to be used, regardless of the real terminal type
char	<i>ttytype</i>	A pointer to the full name of the current terminal
int	<i>LINES</i>	The number of lines on the terminal
int	<i>COLS</i>	The number of columns on the terminal
int	<i>ERR</i>	The error flag; returned by functions on an error
int	<i>OK</i>	The okay flag; returned by functions on successful operation

7

C Library Guide

7.1.4 termcap curses Using /etc/termcap

termcap curses is used with the */etc/termcap* terminal description database. You must place the lines:

```
#define M_TERMCAP
#include <curses.h>
```

in your code and link the program with the command:

```
cc files -ltcap -ltermcap
```

instead of:

```
cc files -lcurses -ltermcap
```

This functionality is specific to and not portable outside of XENIX.

7.1.5 termcap curses Using terminfo

You can use **termcap curses** with the **terminfo** database. To do this, you must place the lines:

```
#define M_TERMCAP
#include <curses.h>
```

and link the program with the command:

```
cc files -ltcap -ltinfo
```

This functionality is specific to and is not portable outside of XENIX. It is useful primarily as a transitional step in converting from **termcap** to **terminfo**.

7.1.6 terminfo curses Using terminfo

In this method you use **terminfo curses** with the **terminfo** database. To do this you must place the lines:

```
#define M_TERMINFO
#include <curses.h>
```

and link your program with the command:

```
cc files -ltinfo
```

7.1.7 Some Additional Notes

Since **curses.h**, **tcap.h**, and **tinfo.h** all include **stdio.h** and **termio.h**, you should not include either of those files in your program.

Note that:

```
#include <tcap.h>
```

is equivalent to:

```
#define M_TERMCAP
#include <curses.h>
```

and that:

```
#include <tinfo.h>
```

is equivalent to:

```
#define M_TERMINFO
#include <curses.h>
```

Types and Constants

Name	Description
reg	A storage class. It is the same as register storage class.
bool	A type. It is the same as char type.
TRUE	The Boolean true value (1).
FALSE	The Boolean false value (0).

7.2 Preparing the Screen

The **initscr** and **endwin** functions initialize and terminate programs that use the screen-processing functions. The following sections describe these functions and how they affect the terminal.

7.2.1 Initializing the Screen

The **initscr** function initializes screen processing for programs by allocating the required memory space for the screen-processing functions and variables, and setting the screen to the proper mode.



C Library Guide

The function call has the following syntax:

initscr()

No arguments are required. For example, in the following program fragment, **initscr** initializes the screen-processing functions.

```
main ()
{

initscr ();
if (!(strcmp(ttytype,"unknown")))
    fprintf(stderr, "Terminal type can't display screen.");
```

In the above example, the *strcmp* function checks to see if the predefined variable *ttytype* set to the specified terminal type *unknown*.

7.2.2 Using Terminal Capability and Type

The **initscr** function examines the terminal-capability descriptions specified in the terminal capabilities database to prepare the screen-processing functions for creating and updating terminal screens.

For **curses** programs, **initscr** examines

/etc/termcap

For **terminfo** programs, **initscr** examines

/usr/lib/terminfo

The descriptions define the character sequences required to perform a given operation on a given terminal. These sequences are used by the screen-processing functions to add, insert, delete, and move characters on the screen. The descriptions are automatically read from the file when screen processing is initialized, so direct access by a program is not required.

The **initscr** function use the shell's *TERM* variable to determine which terminal-capability description to use. The *TERM* variable is usually assigned an identifier when a user logs in. This identifier defines the terminal type and is associated with a terminal-capability description in the */etc/termcap* file (**curses**) or */usr/lib/terminfo/src/ti.st.src* file (**terminfo**).

If the *TERM* variable has no value, the functions use the default terminal type in the library's predefined variable *Def_term*. This variable initially

has the value *unknown* (unknown terminal), but the user can change it to any desired value. This must be done before calling the **initscr** function.

In some cases, it is desirable to force the screen-processing functions to use the default terminal type. This can be done by setting the library's predefined variable *My_term* to the value 1. The full name of the current terminal is stored in the predefined variable *ttytype*.

For more information on terminal capabilities, types, and identifiers, see **termcap** (F) and **terminfo** (M) in the *XENIX User's Reference*.

7.2.3 Using Default Terminal Modes

The **initscr** function automatically sets a terminal to default operation modes. These modes define how the terminal displays characters sent to the screen and how it responds to characters typed at the keyboard. The **initscr** function sets the terminal to ECHO mode, which causes characters typed at the keyboard to be displayed on the screen, and to RAW mode, which causes characters to be used as direct input (no editing or signal processing is done).

The default terminal modes can be changed by using the appropriate functions described in "Setting a Terminal Mode." If the modes are changed, they must be changed immediately after calling **initscr**. For more information on terminal modes, see **tty** (M) in the *XENIX User's Reference*.

Note

The terminal-mode functions should be used only in conjunction with other screen-processing functions. They should not be used alone.

7

7.2.4 Using Default Window Flags

The **initscr** function automatically clears the *cursor*, *scroll*, and *clear* flags of the standard screen to their default values. These flags, called the *window flags*, define how the **refresh** function affects the terminal screen when updating the standard screen. When clear, the *cursor* flag prevents the terminal's cursor from moving back to its original location after the screen is updated. The *scroll* flag prevents scrolling on the screen. The

C Library Guide

clear flag prevents the characters on the screen from being cleared before being updated. These flags can be changed by using the functions described in “Setting Window Flags.”

7.2.5 Using the Default Terminal Size

The **initscr** function sets the terminal screen size to a default number of lines and columns. The default values are given in the predefined variables *LINES* and *COLS*. You can change the default size of a terminal by setting the variables to new values. This should be done before the first call to **initscr**. If it is done after the first call, a second call to **initscr** must be made to delete the existing standard screen and create a new one.

7.2.6 Terminating Screen Processing

The **endwin** function terminates the screen processing in a program by freeing all memory resources allocated by the screen-processing functions and restoring the terminal to the state before screen processing began. No arguments are required. The function call has the following form:

endwin()

To restore the terminal to its previous state, the **endwin** function must be used before leaving a program that has called the **initscr** function. The function is generally the last function call in the program. For example, in the following program fragment, **initscr** and **endwin** form the beginning and end of the program:

```
#include < curses.h> /* use <terminfo.h> for terminfo functions */
                /* use <curses.h> for curses functions */
main ()
{
    initscr();
    .
    . /* Program body */
    .
    endwin();
}
```

endwin must not be called if **initscr** has not been called. Also, **endwin** should be called before any call to the **exit** function. The **endwin** function must also be called if the **gettermode** and **setterm** functions have been called, even if **initscr** has not.

7.3 Using the Standard Screen

The following sections explain how to use the standard screen, `stdscr`, to display and edit characters on the terminal screen.

7.3.1 Adding a Character

The **addch** function adds a given character to the standard screen and moves the character pointer one position to the right. The function call has the following form:

```
addch(ch)
```

where *ch* gives the character to be added. It must have **char** type. For example, in the following program fragment, if the current cursor position is (0, 0), the **addch** function call places the letter *A* at (0, 0) and moves the cursor pointer to (0, 1) to the standard screen:

```
addch ('A');
```

Other characters used by the **addch** function are as follows:

Character	Description
<code>\n</code>	New-line. Deletes all characters from the current position to the end of the line and moves the pointer one line down. If the <i>newline</i> flag is set, the addch function deletes the characters and moves the pointer to the beginning of the next line.
<code>\r</code>	Return. Moves the pointer to the beginning of the current line.
<code>\t</code>	Tab. Moves the pointer to the next tab stop, adding enough spaces to fill the gap between the current position and the stop. Tab stops are placed at every eight character positions.

The **addch** function returns `ERR` if it encounters an error, such as illegal scrolling.

C Library Guide

7.3.2 Adding a String

The **addstr** function adds a string of characters to the standard screen, placing the first character of the string at the current position and moving the pointer one position to the right for each character in the string. The function call has the following form:

```
addstr(str)
```

where *str* is a character pointer to the given string. For example, in the following program fragment, if the current cursor position is (0,0), the **addstr** function call places the beginning of the string *line* at (0,0) and moves the cursor pointer to (0,4), while adding each character of the string:

```
addstr("line");
```

If the string contains new-line, return, or tab characters, the **addstr** function performs the same actions as described for the **addch** function. If the string does not fit on the current line, the string is truncated.

The **addstr** function returns ERR if it encounters an error, such as illegal scrolling.

7.3.3 Printing Strings, Characters, and Numbers

The **printw** function prints one or more values to the standard screen, where a value can be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the following form:

```
printw(fmt [,arg1, arg2])
```

where

- *fmt* is a pointer to a string that defines the format of the values. A format may be *%s* for a string, *%c* for a character, or *%d*, *%o*, or *%x* for a decimal, octal, or hexadecimal number, respectively. (Other formats are described in **printf**(S) in the *XENIX Programmer's Reference*.) If *%s* is given, the corresponding *arg* must be a character pointer. For other formats, the actual value, or a variable containing the value, can be given.
- *arg* is a value to be printed. If more than one *arg* is given, each must be separated from the preceding argument with a comma (.). For each *arg* given, there must be a corresponding format given in *fmt*.

This function is typically used to copy both numbers and strings to the standard screen at the same time. For example, in the following program fragment, if the current cursor position is (0,0), the **printw** function call prints the string given by the variable *name* starting at cursor position (0,0). The **printw** function then prints the number 15 one space after the *name* string.

```
printw("%s %d", name, 15);
```

The function returns ERR if it encounters an error, such as illegal scrolling.

7.3.4 Reading a Character from the Keyboard

The **getch** function reads a single character from the terminal keyboard and returns the character as a value. The function call has the following form:

```
c = getch()
```

where *c* is the variable to receive the character.

This function is typically used to read a series of individual characters. For example, in the following program fragment characters are read and stored until a new-line character or the end-of-file character is encountered, or until the buffer size has been reached:

```
char c, p[MAX];
int i;

i = 0;
while ((c=getch()) != '\n' && c != EOF && i < MAX)
    p[i++] = c;
```

If the terminal is set to ECHO mode, **getch** updates the character to the standard screen; otherwise, the screen remains unchanged. If the terminal is not set to RAW or NOECHO mode, **getch** automatically sets the terminal to CBREAK mode, then restores the previous mode after reading the character. Terminal modes are described in “Controlling the Terminal.”

The function returns ERR if it encounters an error, such as illegal scrolling.



C Library Guide

Keypad Mode

Keypad mode allows an application to handle function key sequences such that certain sequences of characters are recognized as a single token. Each token is identified by a unique integer to the application. These integer values are all greater than 255 so as not to conflict with any valid ASCII characters, and are #defined in the file *tcap.h*.

To add additional function key sequences to those already defined in *tcap.h*, the application can call:

```
addkey(string, value)
char *str;
int val;
```

where *string* is the key sequence to add and *val* is the unused integer greater than 255 that is to identify the function key.

In order to use this functionality, the application must invoke the macro

```
keypad (win, TRUE)
```

soon after calling *initscr()*. In the above example, *win* is the window from which reads will be taken.

7.3.5 Reading a String from the Keyboard

The **getstr** function reads a string of characters from the terminal keyboard and updates the string to a given location. The function call has the following form:

```
getstr(str)
```

where *str* is a character pointer to the variable or location to receive the string. When typed at the keyboard, the string must end with a new-line character or with the end-of-file character. The extra character is replaced by a null character when the string is stored. You must ensure that *str* has adequate space to store the typed string.

This function typically reads names and other text from the keyboard. For example, in the following program fragment, **getstr** reads a filename from the keyboard and stores it in the array *name*:

```
char name[20];
getstr(name);
```

If the terminal is set to ECHO mode, **getstr** updates the string to the standard screen. If the terminal is not set to RAW or NOECHO mode, the function automatically sets the terminal to CBREAK mode, then restores

the previous mode after reading the character. Terminal modes are described in “Controlling the Terminal.”

The function returns ERR if it encounters an error, such as illegal scrolling.

7.3.6 Reading Strings, Characters, and Numbers

The `scanw` function reads one or more values from the terminal keyboard and copies the values to given locations. A value can be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the following form:

```
scanw(fmt, [argptr1, arg2])
```

where

- *fmt* is a pointer to a string defining the format of the values to be read.
- *argptr* is a pointer to the variable to receive a value. If more than one *argptr* is given, each must be separated from the preceding item with a comma (.). For each *argptr* given, there must be a corresponding format given in *fmt*. A format can be *%s* for a string, *%c* for a character, or *%d*, *%o*, or *%x* for a decimal, octal, or hexadecimal number, respectively. (Other formats are described in `printf(S)` in the *XENIX Programmer's Reference*.)

This function typically reads a combination of strings and numbers from the keyboard. For example, in the following program fragment, `scanw` reads a name and a number from the keyboard:

```
char name[20];
int id;

scanw("%s %d", name, &id);
```

In this example, the input values are stored in the character array *name* and the integer variable *id*.

If the terminal is set to ECHO mode, the function copies the string to the standard screen. If the terminal is not set to RAW or NOECHO mode, the function automatically sets the terminal to CBREAK mode, then restores the previous mode after reading the character. The function returns ERR if it encounters an error, such as illegal scrolling.

C Library Guide

7.3.7 Moving the Current Position

The **move** function moves the pointer to the given position. The function call has the following form:

```
move(y, x)
```

where

- *y* is an integer value giving the new row position.
- *x* is an integer value giving the new column position.

For example, if the current position is (0,0), the following function call moves the pointer to line 5, column 4:

```
move(5, 4);
```

The function returns ERR if it encounters an error, such as illegal scrolling.

7.3.8 Inserting a Character

The **insch** function inserts a character at the current position and shifts the character previously at that position (and all characters to its right) one position to the right. The function call has the following form:

```
insch(c)
```

where *c* is the character to be inserted.

This function is typically used to insert a series of characters into an existing line. For example, in the following program fragment, **insch** is used to insert the number of characters given by *cnt* into the standard screen at the current position:

```
int cnt;
char *string;

while(cnt != 0) {
    insch(string[cnt]);
    cnt--;
}
```

The function returns ERR if it encounters an error, such as illegal scrolling.

7.3.9 Inserting a Line

The **insertln** function inserts a blank line at the current position and moves the line previously at that position (and all lines below it) down one line, causing the last line to move off the bottom of the screen. The function call has the following form:

insertln()

No arguments are required.

This function is used to insert additional lines of text in the standard screen. For example, in the following program fragment, **insertln** is used to insert a blank line when the count in *cnt* is equal to 79:

```
int cnt;

if (cnt == 79) {
    insertln();
}
```

The function returns ERR if it encounters an error, such as illegal scrolling.

7.3.10 Deleting a Character

The **delch** function deletes the character at the current position and shifts the character to the right of the deleted character (and all characters to its right) one position to the left. The last character on the line is replaced by a space. The function call has the following form:

delch()

No arguments are required.

This function is typically used to delete a series of characters from the standard screen. For example, in the following program fragment, **delch** deletes the character at the current position as long as the count in *cnt* is not zero:

C Library Guide

```
int cnt;

while (cnt != 0) {
    delch();
    cnt--;
}
```

The function returns ERR if it encounters an error, such as illegal scrolling.

7.3.11 Deleting a Line

The **deleteln** function deletes the current line and shifts the line below the deleted line (and all lines below it) one line up, leaving the last line on the screen blank. The function call has the following form:

deleteln()

No arguments are required.

The **deleteln** function is used to delete existing lines from the standard screen. For example, in the following program fragment, **deleteln** is used to delete a line from the standard screen if the count in *cnt* is 79:

```
int cnt;

if (cnt == 79) {
    deleteln();
}
```

The function returns ERR if it encounters an error, such as illegal scrolling.

7.3.12 Clearing the Screen

The **clear** and **erase** functions clear all characters from the standard screen by replacing them with spaces. The functions are typically used to prepare the screen for new text.

The **clear** function clears all characters from the standard screen, moves the pointer to (0,0), and sets the standard screen's *clear* flag. The flag causes the next call to the **refresh** function to clear all characters from the terminal screen.

The **erase** function clears the standard screen, but does not set the *clear* flag. For example, in the following program fragment, **clear** clears the

screen if the input value is 12:

```
char c;

if ((c=getch()) == 12) {
    clear();
    refresh();
}
```

7.3.13 Clearing a Part of the Screen

The **clrrobot** and **clrtoeol** functions clear one or more characters from the standard screen by replacing the characters with spaces. The functions are typically used to prepare a part of the standard screen for new characters.

The **clrrobot** function clears the screen from the current position to the bottom of the screen. For example, if the current position is (10,0), the following **clrrobot** function call clears all characters from line 10 and all lines below line 10:

```
clrrobot();
refresh();
```

The **clrtoeol** function clears the standard screen from the current position to the end of the current line. For example, if the current position is (10,10), the following **clrtoeol** function call clears all characters from (10,10) to (10,79); the characters at the beginning of the line remain unchanged:

```
clrtoeol();
```

Note that both the **clrrobot** and **clrtoeol** functions do not change the current cursor position. Each **refresh** function call updates the modified standard screen to the terminal screen.

7

7.3.14 Refreshing from the Standard Screen

The **refresh** function updates the terminal screen by copying one or more characters from the standard screen to the terminal. The function effectively changes the terminal screen to reflect the new contents of the standard screen. The function call has the following form:

```
refresh()
```

C Library Guide

No arguments are required.

This function is used solely to display changes to the standard screen. The function copies only those characters that have changed since the last call to **refresh** and leaves any existing text on the terminal screen. For example, in the following program fragment, **refresh** is called twice:

```
addstr("The first time.\n");
refresh();
addstr("The second time.\n");
refresh();
```

In this example, the first call to **refresh** copies the string “The first time.” to the terminal screen. The second call copies only the string “The second time.” to the terminal, since the original string has not changed.

The function returns ERR if it encounters an error, such as illegal scrolling. If an error is encountered, the function attempts to update as much of the screen as possible without causing the scroll.

7.4 Creating and Using Windows

The following sections explain how to create and use windows to display and edit text on the terminal screen.

7.4.1 Creating a Window

The **newwin** function creates a window and returns a pointer that can be used in subsequent screen-processing functions. The function call has the following form:

```
win = newwin(lines, cols, begin_y, begin_x)
```

where

- *win* is the pointer variable to receive the return value.
- *lines* and *cols* are integer values that give the total number of lines and columns in the window, respectively.
- *begin_y* and *begin_x* are integer values that give the line and column positions, respectively, of the upper-left corner of the window when displayed on the terminal screen. The *win* variable must be a pointer of type **WINDOW**.

The **newwin** function is typically used in programs that maintain a set of windows, displaying different windows at different times or alternating between windows as needed. For example, in the following program fragment, **newwin** creates a new window on the standard screen and assigns a pointer to this window to the variable *midscreen*:

```
WINDOW *midscreen;

midscreen = newwin(5, 10, 9, 35);
```

The window has 5 lines and 10 columns. The upper-left corner of the window is placed at the position (9,35) on the terminal screen.

If either *lines* or *cols* is zero, the function automatically creates a window that has *LINES* lines or *COLS* columns, where *LINES* and *COLS* are the predefined constants that give the total number of lines and columns on the terminal screen. For example, the following function call creates a new window whose upper-left corner is at position (0,0) and that has *LINES* lines and *COLS* columns:

```
newwin(0, 0, 0, 0)
```

You must not create windows that exceed the dimensions of the actual screen.

The **newwin** function returns the value (WINDOW*) ERR on an error, such as “insufficient memory for the new window.”

7.4.2 Creating a Subwindow

The **subwin** function creates a subwindow and returns a pointer to the new window. A subwindow is a window that shares all or part of the character space of another window and provides an alternate way to access the characters in that space. The function call has the following form:

```
swin = subwin(win, lines, cols, begin_y, begin_x)
```

where

- *swin* is the pointer variable to receive the return value.
- *win* is the pointer to the window to contain the new subwindow.
- *lines* and *cols* are integer values that give the total number of lines and columns in the subwindow, respectively.



C Library Guide

- *begin_y* and *begin_x* are integer values that give the line and column positions, respectively, of the upper-left corner of the subwindow when displayed on the terminal screen. The *swin* variable must be a pointer of type **WINDOW**.

The **subwin** function is typically used to divide a large window into separate regions. For example, in the following program fragment, **subwin** creates the subwindow named *cmdmenu* in the lower part of the standard screen:

```
WINDOW *cmdmenu;  
  
cmdmenu = subwin(stdscr, 5, 80, 19, 0);
```

In this example, changes to *cmdmenu* affect the standard screen as well.

The **subwin** function returns the value (WINDOW*) ERR on an error, such as insufficient memory for the new window.

7.4.3 Accessing Window Structure

The following **termcap** macros are provided to allow you to get useful information on the structure of windows you have created.

getdim(win, y, x)	Gets dimensions of window <i>win</i> specified as coordinates <i>y</i> and <i>x</i> .
getorg(win, y, x)	Locates origin (upper left point) of <i>win</i> expressed as <i>y</i> and <i>x</i> .
is_standout(win)	Determines if window <i>win</i> is in standout mode.
iscuroff()	Determines if cursor is in invisible mode.
tscroll(win, bf)	Enables or disables scrolling in window <i>win</i> based on the value of boolean flag <i>bf</i> (1 enables, 0 disables).
autoflush(bf)	Suspends explicit flushing of output buffers in a wrefresh() command when the boolean flag <i>bf</i> is 0. The output buffer will still be refreshed when it is full but not on each call to wrefresh() . The default mode is to flush the buffer after each call to wrefresh() .

7.4.4 Adding and Printing to a Window

The **waddch**, **waddstr**, and **wprintw** functions add and print characters, strings, and numbers to a given window.

The **waddch** function adds a given character to the given window and moves the character pointer one position to the right. The function call has the following form:

```
waddch(win, c)
```

where

- *win* is a pointer to the window to receive the character.
- *c* is the character to be added, which must have **char** type.

For example, if the current position in the window *midscreen* is (0,0), the function call below places the letter *A* at this position and moves the pointer to (0,1):

```
waddch(midscreen, 'A')
```

The **waddstr** function adds a string of characters to the given window, placing the first character of the string at the current position and moving the pointer one position to the right for each character in the string. The function call has the following form:

```
waddstr(win, str)
```

where

- *win* is a pointer to the window to receive the string.
- *str* is a character pointer to the given string.

For example, if the current position is (0,0), the following function call places the beginning of the string line at this position and moves the pointer to (0,4):

```
waddstr(midscreen, "line");
```

The **wprintw** function prints one or more values in the given window, where a value may be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the following form:

```
wprintw(win, fmt[, arg1, arg2])
```

where

- *win* is a pointer to the window to receive the values.



C Library Guide

- *fmt* is a pointer to a string that defines the format of the values. A format may be *%s* for a string, *%c* for a character, or *%d*, *%o*, or *%x* for a decimal, octal, or hexadecimal number, respectively. (Other formats are described in **printf(S)** in the *XENIX Programmer's Reference*.)
- *arg* is a value to be printed. If more than one *arg* is given, each must be separated from the preceding one with a comma (.). For each *arg* given, there must be a corresponding format given in *fmt*. If *%s* is given, the corresponding *arg* must be a character pointer. For other formats, the actual value or a variable containing the value can be given.

This function is typically used to copy both numbers and strings to the standard screen at the same time. For example, in the following program fragment, **wprintw** prints a name and then the number 15 at the current position in the window *midscreen*:

```
char *name;

wprintw(midscreen, "%s %d", name, 15);
```

Note that when a new-line, return, or tab character is given to a **waddch**, **waddstr**, or **wprintw** function, the functions perform the same actions as described for the **addch** function. The functions return ERR if they encounter an error, such as illegal scrolling.

7.4.5 Reading and Scanning for Input

The **wgetch**, **wgetstr**, and **wscanw** functions read characters, strings, and numbers from the standard input file and usually echo the values by copying them to the given window.

The **wgetch** function reads a single character from the standard input file and returns the character as a value. The function call has the following form:

```
c = wgetch(win)
```

where

- *win* is a pointer to a window.
- *c* is the character variable to receive the character.

This function is typically used to read a series of characters from the keyboard. For example, in the following program fragment, **wgetch** reads characters until a colon (:) is found:

```
char c, dir[MAX];
int i;

i = 0;
while ((c=wgetch(cmdmenu)) != ':' && i <MAX)
    dir[i++] = c;
```

The **wgetstr** function reads a string of characters from the terminal keyboard and copies the string to a given location. The function call has the following form:

```
wgetstr(win, str)
```

where

- *win* is a pointer to a window.
- *str* is a character pointer to the variable or location to receive the string. When typed at the keyboard, the string must end with a new-line character or with the end-of-file character. The extra character is replaced by a null character when the string is stored. You must ensure that *str* has adequate space for storing the typed string.

This function typically reads names and other text from the keyboard. For example, in the following program fragment, **wgetstr** reads a string from the keyboard and stores it in the array *filename*:

```
char filename[20];

wgetstr(cmdmenu, filename);
```

The **wscanw** function reads one or more values from the standard input file and copies the values to given locations. A value can be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the following form:

```
wscanw(win, fmt [, argptr1, arg2])
```

where

- *win* is a pointer to a window.
- *fmt* is a pointer to a string defining the format of the values to be read.

C Library Guide

- *argptr* is a pointer to the variable to receive a value. If more than one *argptr* is given, each must be separated from the preceding by a comma (,). For each *argptr* given, there must be a corresponding format given in *fmt*. A format can be *%s* for a string, *%c* for a character, or *%d*, *%o*, or *%x* for a decimal, octal, or hexadecimal number, respectively. (Other formats are described in `scanf(S)` in the *XENIX Programmer's Reference*.)

This function typically reads a combination of strings and numbers from the keyboard. For example, in the following program fragment `wscanw` reads a name and a number from the keyboard:

```
char name[20];
int id;

wscanw("%s %d", name, &id);
```

In this example, the input values are stored in the character array *name* and the integer variable *id*.

If you set the terminal to ECHO mode, the function copies the string to the standard screen. If you do not set the terminal to RAW or NOECHO mode, the function automatically sets the terminal to CBREAK mode, then restores the previous mode after reading the character.

The function returns ERR if it encounters an error, such as illegal scrolling.

7.4.6 Moving the Current Position in a Window

The `wmove` function moves the current position in a given window. The function call has the following form:

```
wmove(win, y, x)
```

where

- *win* is a pointer to a window.
- *y* is an integer value that gives the new line position.
- *x* is an integer value that gives the new column position.

For example, the following function call moves the current position in the window *midscreen* to (4,4):

```
wmove(midscreen, 4, 4)
```

The function returns ERR if it encounters an error, such as illegal scrolling.

7.4.7 Inserting Characters and Lines

The **winsch** and **winsertln** functions insert characters and lines into a given window.

The **winsch** function inserts a character at the current position and shifts the character previously at that position (and all characters to its right) one position to the right. The function call has the following form:

```
winsch(win, c)
```

where

- *win* is a pointer to a window.
- *c* is the character to be inserted.

This function is typically used to edit the contents of the given window. For example, the following program fragment inserts the character *X* at the current position in the window *midscreen*:

```
winsch(midscreen, 'X');
```

The **winsertln** function inserts a blank line at the current position and moves the existing line and all lines below it down one line, causing the last line to move off the bottom of the screen. The function call has the following form:

```
winsertln(win)
```

where *win* is a pointer to the window to receive the blank line.

The **winsertln** function is used to insert lines into a window. For example, in the following program fragment, **winsertln** inserts a blank line at the top of the window *cmdmenu*, preparing it for a new line:

```
char line[80];

wmove(cmdmenu, 3, 0);
winsertln(cmdmenu);
waddstr(cmdmenu, line);
```

Both functions return ERR if they encounter errors, such as illegal scrolling.



C Library Guide

7.4.8 Deleting Characters and Lines

The **wdelch** and **wdeleteln** functions delete characters and lines from the given window.

The **wdelch** function deletes the character at the current position and shifts all characters to the right of the deleted character one position to the left. The last character on the line is replaced with a space. The function call has the following form:

```
wdelch(win)
```

where *win* is a pointer to a window.

This function is typically used to edit the contents of the standard screen. For example, the following program fragment call deletes the character at the current position in the window *midscreen*:

```
wdelch(midscreen);
```

The **wdeleteln** function deletes the current line and shifts all lines below the deleted line one line up, leaving the last line in the screen blank. The function call has the following form:

```
wdeleteln(win)
```

where *win* is a pointer to a window.

This function is typically used to delete existing lines from a given window. For example, in the following program fragment, **wdeleteln** deletes the lines in *midscreen* until *cnt* is equal to zero:

```
int cnt;

while (cnt != 0) {
    wdeleteln(midscreen);
    cnt--;
}
```

7.4.9 Clearing the Window Screen

The **wclear**, **werase**, **wclrto bot**, and **wclrtoeol** functions clear all or part of the characters from the given window by replacing them with spaces. The functions are typically used to prepare the window for new text.

The **wclear** function clears all characters from the window, moves the pointer to (0,0), and sets the standard screen's *clear* flag. The flag causes the next **refresh** function call to clear all characters from the terminal screen. The function call has the following form:

wclear (*win*)

where *win* is the window to be cleared.

The **werase** function clears the given window and moves the pointer to (0,0), but does not set the *clear* flag. It is used whenever the contents of the terminal screen must be preserved. The function call has the following form:

werase (*win*)

where *win* is a pointer to the window to be cleared.

The **wclrtoBOT** function clears the window from the current position to the bottom of the screen. The function call has the following form:

wclrtoBOT (*win*)

where *win* is a pointer to the window to be cleared. For example, if the current position in the window *midscreen* is (10,0), the following program fragment clears all characters from line 10 and all lines below line 10:

```
wclrtoBOT (midscreen) ;
```

The **wclrtoeOL** function clears the standard screen from the current position to the end of the current line. The function call has the following form:

wclrtoeOL (*win*)

where *win* is a pointer to the window to be cleared. For example, if the current position in *midscreen* is (10,10), the following program fragment clears all characters from (10,10) to the end of the line; the characters at the beginning of the line remain unchanged:

```
wclrtoeOL (midscreen) ;
```

Note that the **wclrtoBOT** and **wclrtoeOL** functions do not change the current position.

C Library Guide

7.4.10 Saving from a Window

The **dmpwin()** function saves the contents of a window to a specified file. The information in the window is saved without attributes. A call to **dmpwin()** takes the following form:

```
dmpwin(win, fp, margin)
WINDOW *win;
FILE *fp;
int margin;
```

where *win* is the window to be dumped, *fp* is a pointer to the file where the information must go, and *margin* is an integer value indicating the left margin justification of the saved information in the file.

This function allows the use of a *printscreen* function key. For example, you can implement a *printscreen* key by programming the key to signal your application to call **dmpwin** to save to a file and then print the file.

7.4.11 Refreshing from a Window

The **wrefresh** function updates the terminal screen by copying one or more characters from the given window to the terminal. The function effectively changes the terminal screen to reflect the new contents of the window. The function call has the following form:

wrefresh(*win*)

where *win* is a pointer to a window. This function solely displays changes to the window. The function copies only those characters that have changed since the last call to **wrefresh** and leaves any existing text on the terminal screen. For example, in the following program fragment, **wrefresh** is called twice:

```
waddstr(cmdmenu, "Type a command name\n");
wrefresh(cmdmenu);
waddstr(cmdmenu, "Command: ");
wrefresh(cmdmenu);
```

In this example, the first call to **wrefresh** copies the string “Type a command name” to the terminal screen. The second call copies only the string “Command:” to the terminal, since the original string has not changed.

Note

If *curscr* is given with **wrefresh**, the function restores the actual screen to its most recent contents. This is useful for implementing a *redraw* feature for screens that become cluttered with unwanted output.

The function returns ERR if it encounters an error, such as illegal scrolling. If an error is encountered, the function attempts to update as much of the screen as possible without causing the scroll.

7.4.12 Overlaying Windows

The **overlay** function copies all characters, except spaces, from one window to another, moving characters from their original positions in the first window to identical positions in the second. The function effectively lays the first window over the second, letting characters in the second window that would otherwise be covered by spaces remain unchanged. The function call has the following form:

```
overlay(win1, win2)
```

where

- *win1* is a pointer to the window to be copied.
- *win2* is a pointer to the window to receive the copied text.

The starting positions of *win1* and *win2* must match, otherwise an error occurs. If *win1* is larger than *win2*, the function copies only those lines and columns in *win1* that fit in *win2*.

This function is typically used to build a composite screen from overlapping windows. For example, in the following program fragment **overlay** is used to build the standard screen from two different windows:

```
WINDOW *info, *cmdmenu;  
  
overlay(info, stdscr);  
overlay(cmdmenu, stdscr);  
refresh();
```

C Library Guide

Note that this function cannot be used with disjoint windows. (Windows that do not overlap.)

7.4.13 Overwriting a Screen

The **overwrite** function copies all characters, including spaces, from one window to another, moving characters from their positions in the first window to identical positions in the second. The function effectively writes the contents of the first window over the second, destroying the previous contents of the second window. Note that the windows must be overlapping in order to use this function. **overwrite** is typically used to display the contents of a temporary window in the middle of a larger window. The function call has the following form:

```
overwrite(win1, win2)
```

where

- *win1* is a pointer to the window to be copied.
- *win2* is a pointer to the window to receive the copied text.

If *win1* is larger than *win2*, the function copies only those lines and columns in *win1* that fit in *win2*.

For example, in the following program fragment, **overwrite** is used to copy the contents of a window to the standard screen:

```
WINDOW *work;  
  
overwrite(work, stdscr);  
refresh();
```

7.4.14 Moving a Window

The **mvwin** function moves a given window to a new position on the terminal screen, causing the upper-left corner of the window to occupy a given line and column position. The function call has the following form:

```
mvwin(win, y, x)
```

where

- *win* is a pointer to the window to be moved.
- *y* is an integer value giving the line to which the corner is to be moved.

- x is an integer value giving the column the corner is to be moved to.

This function is typically used to move a temporary window when an existing window under it contains information to be viewed. For example, in the following program fragment, `mvwin` moves the window named *work* to the upper-left corner of the terminal screen:

```
WINDOW *work;

mvwin(work, 0, 0);
```

The function returns `ERR` if it encounters an error, such as an attempt to move part of a window off the edge of the screen.

7.4.15 Reading a Character from a Window

The `inch` and `winch` functions read a single character from the current pointer position in a window or screen.

The `inch` function reads a character from the standard screen. The function call has the following form:

```
 $c = \text{inch}()$ 
```

where c is the character variable to receive the character read.

The `winch` function reads a character from a given window or screen. The function call has the following form:

```
 $c = \text{winch}(win)$ 
```

where win is the pointer to the window containing the character to be read.

These functions are typically used to compare the actual contents of a window with what is assumed to be there. For example, in the following program fragment, `inch` and `winch` are used to compare the characters at position (0,0) in the standard screen and in the window named *altscreen*:

```
char c1, c2;

c1 = inch();
c2 = winch(altscreen);
if(c1 != c2)
    error();
```

Note that reading a character from a window does not alter the contents of the window.

C Library Guide

7.4.16 Touching a Window

The **touchwin** function makes the entire contents of a given window appear to be modified, causing a subsequent **refresh** call to copy all characters in the window to the terminal screen. The function call has the following form:

```
touchwin(win)
```

where *win* is a pointer to the window to be touched.

This function is typically used when two or more overlapping windows comprise the terminal screen. For example, the following function call is used to touch the window named *leftscreen*:

```
touchwin(leftscreen);
```

A subsequent **refresh** copies all characters in *leftscreen* to the terminal screen.

7.4.17 Deleting a Window

The **delwin** function deletes a given window from memory, freeing the space previously occupied by the window for other windows or for dynamically allocated variables.

The function call has the following form:

```
delwin(win)
```

where *win* is a pointer to the window to be deleted.

This function is typically used to remove temporary windows from a program or to free memory space for other uses. For example, the following function call removes the window named *midscreen*:

```
delwin(midscreen);
```

7.5 Using Other Window Functions

The following sections explain how to perform a variety of operations on existing windows, such as setting window flags and drawing boxes around the windows.

7.5.1 Drawing a Box

The **box** function draws a box around a window using the given characters to form the horizontal and vertical sides. The function call has the following form:

```
box(win, vert, hor)
```

where

- *win* is a pointer to the desired window.
- *vert* is the vertical character.
- *hor* is the horizontal character.

Both *vert* and *hor* must have **char** type.

This function is typically used to distinguish one window from another when combining windows on a single screen. For example, in the following program fragment, **box** creates a box around the window in the lower half of the screen:

```
WINDOW *cmdmenu;

cmdmenu = subwin(stdscr, 5, 80, 19, 0);
box(cmdmenu, '|', '-');
```

If necessary, the function will leave the corners of the box blank to prevent illegal scrolling.

7.5.2 Displaying Bold Characters

The **standout** and **wstandout** functions set the standout character attribute, causing characters subsequently added to the given window or screen to be displayed as bold characters.

The **standout** function sets the standout attribute for characters added to the standard screen. The function call has the following form:

```
standout()
```

No arguments are required.

C Library Guide

The **wstandout** function sets the standout attribute of characters added to the given window or screen. The function call has the following form:

```
wstandout(win)
```

where *win* is a pointer to a window.

These functions are typically used to make error messages or instructions clearly visible when displayed at the terminal screen. For example, in the following program fragment, **standout** sets the standout character attribute before adding an error message to the standard screen:

```
if (code = 5) {  
    standout();  
    addstr("Illegal character.\n");  
}
```

Note that the actual appearance of characters with the standout attribute depends on the given terminal. This attribute is defined by the SO and SE (or US and UE) sequences given in the terminal's **termcap** or **terminfo** terminal-capability database entry. For more information, see **termcap** (M) or **terminfo** (M) in the *XENIX User's Reference*.

7.5.3 Restoring Normal Characters

The **standend** and **wstandend** functions restore the normal character attribute, causing characters subsequently added to a given window or screen to be displayed as normal characters.

The **standend** function restores the normal attribute for the standard screen. The function call has the following form:

```
standend()
```

No arguments are required.

This **wstandend** function restores the normal attribute for a given window or screen. The function call has the following form:

```
wstandend(win)
```

where *win* is a pointer to a window.

These functions are typically used after an error message or instructions have been added to a screen using the standout attribute. For example, in

the following program fragment, **standend** restores the normal attribute after an error message has been added to the standard screen:

```
if (code = 5) {
    standout();
    addstr("Illegal character.\n");
    standend();
}
```

Multiple Attributes

It is possible to specify more than one attribute at a time for a given character. Each character in a window is represented by two bytes of information. The low order byte represents the character itself, and the high order byte represents the attributes, if any, that the character requires.

In the attribute byte the low three bits are predefined. The lowest bit represents `A_STANDOUT`, the second bit represents `A_BLINK`, and the third bit represents `A_UNDERLINE`. The last 5 bits are user definable. To turn the attributes on, use the function

wattron(*win, attribute*)

and to turn the attribute off, use the function:

wattroff(*win, attribute*)

For more information on defining your own character attributes, refer to the section on “Advanced Topics” later in this chapter.

7.5.4 Getting the Current Position

The **getyx** function copies the current line and column positions of a given window pointer to a corresponding pair of variables. The function call has the following form:

getyx(*win, y, x*)

where

- *win* is a pointer to the window containing the pointer to be examined.
- *y* is the integer variable to receive the line position.
- *x* is the integer variable to receive the column position.



C Library Guide

This function is typically used to save the current position so that the program can return to the position at a later time. For example, in the following program fragment, **getyx** saves the current line and column positions in the variables *line* and *column*:

```
int line, column;

getyx(stdscr, line, column);
```

7.5.5 Setting Window Flags

The **leaveok**, **scrollok**, and **clearok** functions set or clear the *cursor*, *scroll*, and *clear-screen* flags. The flags control the action of the **refresh** function when it is called for the given window.

The **leaveok** function sets or clears the *cursor* flag, which defines how the **refresh** function places the terminal cursor and the window pointer after updating the screen. If the flag is set, **refresh** leaves the cursor after the last character to be copied and moves the pointer to the corresponding position in the window. If the flag is cleared, **refresh** moves the cursor to the same position on the screen as the current pointer position in the window. The function call has the following form:

```
leaveok(win, state)
```

where

- *win* is a pointer to the window containing the flag to be set.
- *state* is a Boolean value defining the state of the flag.

If *state* is **TRUE**, the flag is set; if it is **FALSE**, the flag is cleared. For example, the following function call sets the *cursor* flag:

```
leaveok(stdscr, TRUE);
```

The **scrollok** function sets or clears the *scroll* flag for the given window. If the flag is set, scrolling through the window is allowed. If the flag is clear, no scrolling is allowed. The function call has the following form:

```
scrollok(win, state)
```

where

- *win* is a pointer to a window.
- *state* is a Boolean value defining how the flag is to be set.

If *state* is TRUE, the flag is set; if it is FALSE, the flag is cleared. The flag is initially clear, making scrolling illegal.

The **clearok** function sets and clears the *clear* flag for a given screen. The function call has the following form:

```
clearok(win, state)
```

where

- *win* is a pointer to the desired screen.
- *state* is a Boolean value.

The function sets the flag if *state* is TRUE, and clears the flag if it is FALSE. For example, the following function call sets the *clear* flag for the standard screen:

```
clearok(stdscr, TRUE)
```

When the *clear* flag is set, each **refresh** call to the given screen automatically clears the screen by passing a clear-screen sequence to the terminal. This sequence affects the terminal only; it does not change the contents of the screen.

If **clearok** is used to set the *clear* flag for the current screen *curscr*, each call to **refresh** automatically clears the screen, regardless of which window is given in the call.

7.5.6 Scrolling a Window

The **scroll** function scrolls the contents of a given window upward one line. The function call has the following form:

```
scroll(win)
```

where *win* is a pointer to the window to be scrolled. This function should be used only in special cases.



C Library Guide

7.6 Combining Movement with Action

Many screen operations move the current position of a given window before performing an action on the window. For convenience, you can combine a number of functions with the movement prefix. This combination has the following form:

```
mvfunc([win,]y, x[,arg1, arg2])
```

where

- *func* is the name of a function.
- *win* is a pointer to the window to be operated on (*stdscr* is used if no window is given).
- *y* is an integer value that states the line to move to.
- *x* is an integer value that states the column to move to.
- *arg* is a required argument for the given function.

If more than one argument is required, they must be separated with commas (,). For example, the following function call moves the position to (10,5) and adds the character *X*:

```
mvaddch(10, 5, 'X');
```

The operation is the same as moving the position with the **move** function, then adding a character with **addch**.

For a complete list of the functions that can be used with the movement prefix, see **curses** (S) and **terminfo** (S) in the *XENIX Programmer's Reference*.

7.7 Controlling the Terminal

The following sections explain how to set the terminal modes, how to move the cursor, and how to access other aspects of the terminal. These functions should be used only when using other screen-processing functions.

7.7.1 Setting a Terminal Mode

The **crmode**, **echo**, **nl**, and **raw** functions set the terminal mode, causing subsequent input from the terminal's keyboard to process accordingly.

The **crmode** function sets the CBREAK mode for the terminal. This mode preserves the function of the signal keys, allowing signals to be sent to a program from the keyboard, but disables the function of the editing keys. The function call has the following form:

crmode()

No arguments are required.

The **echo** function sets the ECHO mode for the terminal, causing each character typed at the keyboard to be displayed at the terminal screen. The function call has the following form:

echo()

No arguments are required.

The **nl** function sets a terminal to NEWLINE mode, causing all new-line characters to be mapped to a corresponding carriage-return-new-line character combination. The function call has the following form:

nl()

No arguments are required.

The **raw** function sets the RAW mode for the terminal, causing each character typed at the keyboard to be sent as direct input. The RAW mode disables the function of the editing and signal keys and disables the mapping of new-line characters into carriage-return-new-line combinations. The function call has the following form:

raw()

No arguments are required.



7.7.2 Clearing a Terminal Mode

The **nocrmode**, **noecho**, **nonl**, and **noraw** functions clear the current terminal mode, allowing input to be processed according to a previous mode.

The **nocrmode** function clears a terminal from the CBREAK mode. The function call has the following form:

nocrmode()

No arguments are required.

C Library Guide

The **noecho** function clears a terminal from the ECHO mode. This mode prevents characters typed at the keyboard from being displayed on the terminal screen. The function call has the following form:

noecho()

No arguments are required.

The **nonl** function clears a terminal from NEWLINE mode, causing new-line characters to be mapped into themselves, rather than be acted on by the screen. This lets the screen-processing functions optimize performance. The function call has the following form:

nonl()

No arguments are required.

The **noraw** function clears a terminal from RAW mode, restoring normal editing and signal-generating functions to the keyboard. The function call has the following form:

noraw()

No arguments are required.

7.7.3 Moving the Terminal's Cursor

The **mvcur** function moves the terminal's cursor from one position to another in an optimal fashion. The function call has the following form:

mvcur(*last_y*, *last_x*, *new_y*, *new_x*)

where

- *last_y* and *last_x* are integer values that give the last line and column positions of the cursor.
- *new_y* and *new_x* are integer values that give the new line and column positions of the cursor.

For example, the following program fragment call moves the cursor from (10,5) to (3,0) on the terminal screen:

```
mvcur(10, 5, 3, 0)
```

Note

The **mvcur** function should be used only in programs that do not use other screen-processing functions. This means the function can be used to perform optimal cursor motion without the aid of the other functions. For programs that do use other functions, the **move**, **wmove**, **refresh**, and **wrefresh** functions must be used to move the cursor.

7.7.4 Getting the Terminal Mode

The **getmode** function returns the current *tty* mode. The function call has the following form:

```
getmode()
```

This function is normally called by the **initscr** function.

7.7.5 Saving and Restoring the Terminal Flags

The **savetty** function saves the current terminal flags and the **resetty** function restores the flags previously saved by the **savetty** function. These functions are performed automatically by the **initscr** and **endwin** functions. They are not required when performing ordinary screen processing.

7.7.6 Setting a Terminal Type

The **setterm** function sets the terminal type to a given type. The function call has the following form:

```
setterm (name)
```

where *name* is a pointer to a string containing the terminal type identifier. The function is normally called by the **initscr** function, but can be called separately in special cases.

C Library Guide

7.7.7 Reading the Terminal Name

The **longname** function returns the full name of the terminal corresponding to a given **termcap** or **terminfo** identifier. The function call has the following form:

```
longname(termbuf, name)
```

where

- *termbuf* is a pointer to the string containing the terminal type identifier.
- *name* is a default string to return if no terminal name is found in *termbuf*.

The terminal type identifier must exist in the */etc/termcap* file (**courses**) or */usr/lib/terminfo* file (**terminfo**).

This function is typically used to get the full name of the terminal currently being used. Note that the current terminal's identifier is stored in the variable *ttytype*, which may be used to receive a new name.

7.8 Advanced Topics

This section covers more advanced issues in screen processing topics. They provide added control for more extensive screen applications.

7.8.1 Multiple Attributes

It is possible to specify more than one attribute at a time for a given character. Each character in a window is represented by two bytes of information. The low order byte represents the character itself, and the high order byte represents the attributes, if any, that the character requires.

In the attribute byte the low three bits are predefined. The lowest bit represents **A_STANDOUT**, the second bit represents **A_BLINK**, and the third bit represents **A_UNDERLINE**. The last 5 bits are user definable.

To establish your attributes at start-up time, initialize your variables in the following way:

```
char *A4S; /* Attribute Start Sequence */
char *A4E; /* Attribute End Sequence */
int *A4G; /* Number of Magic Cookie Glitch Characters
           associated with attribute*/
```

You can identify the attributes using `A_ATTR4` through `A_ATTR8`.

7.8.2 Saving and Restoring tty Settings

There are two functions to allow you to restore the default terminal settings and then return to your terminal setup for **curses**. These functions are `reset_tty()` and `save_tty()`. They can also be used to implement shell escapes. For example.

```
{
    reset_tty();
    system("sh");
    save_tty();
}
```

7.8.3 Output Mapping Features

Four functions are provided for users to define their own output mapping. These are blank functions that the programmer can fill in to provide any sort of custom handling desired. They are described below:

tputch

```
tputch(c) /* default version of tputch */

unsigned char c;
{
    putchar(c);
}
```

This function is available for users to define their own handling of terminal sequences. If you wish to implement special handling of outputting terminal sequences, you can write your own `tputch()` and **curses** will call the function whenever it is outputting sequences to the terminal.



C Library Guide

mputc

```
mputc(c) /* default version of mputc */  
  
unsigned char c;  
{  
    putchar(c);  
}
```

This function is available for output mapping. If you wish to do your own output mapping you can write your own version of `mputc()` and it will be called by `curses` during output.

obufflush

```
obufflush() /* default output buffer flushing */  
  
{  
    fflush(stdout)  
}
```

You may define your own mechanism for flushing the output buffer. `curses` will call this routine when flushing output.

init_cmap

```
init_cmap()  
{  
    return;  
}
```

This function is provided to allow the user to set up any special handling they might require for their customized routines. This function is called during `initscr()`.

Chapter 8

Character and String Processing

- 8.1 Introduction 8-1
- 8.2 Using the Character Functions 8-1
 - 8.2.1 Testing for an ASCII Character 8-1
 - 8.2.2 Converting to ASCII Characters 8-2
 - 8.2.3 Testing for Alphanumerics 8-3
 - 8.2.4 Testing for a Letter 8-3
 - 8.2.5 Testing for Control Characters 8-4
 - 8.2.6 Testing for a Decimal Digit 8-4
 - 8.2.7 Testing for a Hexadecimal Digit 8-5
 - 8.2.8 Testing for Printable Characters 8-5
- 8.3 Testing for Punctuation 8-5
 - 8.3.1 Testing for Whitespace 8-6
 - 8.3.2 Testing for Case in Letters 8-6
 - 8.3.3 Converting the Case of a Letter 8-7
- 8.4 Using the String Functions 8-7
 - 8.4.1 Concatenating Strings 8-8
 - 8.4.2 Comparing Strings 8-8
 - 8.4.3 Copying a String 8-9
 - 8.4.4 Getting a String's Length 8-10
 - 8.4.5 Concatenating Characters to a String 8-10
 - 8.4.6 Comparing Characters in Strings 8-11
 - 8.4.7 Copying Characters to a String 8-11
 - 8.4.8 Reading Values from a String 8-12
 - 8.4.9 Writing Values to a String 8-13

8.1 Introduction

Character and string processing is an important part of many programs. Programs regularly assign, manipulate, and compare characters and strings in order to complete their tasks. For this reason, the standard library provides a variety of character and string processing functions. These functions give a convenient way to test, translate, assign, and compare characters and strings.

To use the character functions in a program, the file *ctype.h*, which provides the definitions for special character macros, must be included in the program. The line:

```
#include <ctype.h>
```

must appear at the beginning of the program.

To use the string functions, no special action is required. These functions are defined in the standard C library and are read whenever you compile a C program.

8.2 Using the Character Functions

The character functions test and convert characters. Many character functions are defined as macros, and as such cannot be redefined or used as a target for a breakpoint when debugging.

8.2.1 Testing for an ASCII Character

The `isascii` function tests for characters in the ASCII character set; i.e., characters whose values range from 0 to 127. The function call has the form:

```
isascii (c)
```

where *c* is the character to be tested. The function returns a nonzero (true) value if the character is ASCII, otherwise it returns zero (false). For example, in the following program fragment, `isascii` determines whether



C Library Guide

or not the value in *c*, read from the file given by *data*, is in the acceptable ASCII range:

```
FILE *data;
int c;
:
:
c = fgetc(data);
if (!isascii(c))
    notext();
```

In this example, a function named *notext* is called if the character is not in range.

8.2.2 Converting to ASCII Characters

The *toascii* function converts non-ASCII characters to ASCII. The function call has the form:

```
c = toascii(i)
```

where *c* is the variable to receive the character, and *i* is the value to be changed. The function creates an ASCII character by truncating all but the low order 7 bits of the non-ASCII value. If the *i* value is already an ASCII character, no change takes place. For example, the function call:

```
ascii = toascii(160);
```

converts value 160 to 32, the ASCII value of the space character.

The function is typically used to prepare non-ASCII characters for display on the standard output. For example, in the following program fragment, *toascii* converts each character read from the file given by *oddstrm*:

```
FILE *oddstrm;
int c;
:
:
c = toascii( getc( oddstrm ) );
if ( isprint(c) || isspace(c) )
    putchar(c);
```

If the resulting character is printable or is whitespace, it is written to the standard output.

8.2.3 Testing for Alphanumeric

The **isalnum** function tests for letters and decimal digits; i.e., the alphanumeric characters. The function call has the form:

isalnum (*c*)

where *c* is the character to test. The function returns a nonzero (true) value if the character is an alphanumeric, otherwise it returns zero (false). For example, the function call:

```
isalnum('1')
```

returns a nonzero value, but the call:

```
isalnum('>')
```

returns zero.

8.2.4 Testing for a Letter

The **isalpha** function tests for uppercase or lowercase letters; i.e., alphabetic characters. The function call has the form:

isalpha (*c*)

where *c* is the character to be tested. The function returns a nonzero (true) value if the character is a letter, otherwise it returns zero (false). For example, the function call:

```
isalpha('a')
```

returns a nonzero value, but the call:

```
isalpha('1')
```

returns zero.

C Library Guide

8.2.5 Testing for Control Characters

The **isctrl** function tests for control characters; i.e., characters whose ASCII values are in the range 0 to 31 or is 127. The function call has the form:

isctrl (*c*)

where *c* is the character to be tested. The function returns a nonzero (true) value if the character is a control character, otherwise it returns zero (false). For example, in the program following fragment, **isctrl** determines whether or not the character in *c* read from the file given by *infile* is a control character:

```
FILE *infile, *outfile;
int c;
:
:
c = fgetc(infile);
if ( !isctrl(c) )
    fputc( c, outfile );
```

The **fputc** function is ignored if the character is a control character.

8.2.6 Testing for a Decimal Digit

The **isdigit** function tests for decimal digits. The function call has the form:

isdigit (*c*)

where *c* is the character to be tested. The function returns a nonzero value if the character is a digit, otherwise it returns zero. For example, in the following program fragment, each new character in *c* is added to the running total if the character is a digit:

```
FILE *infile;
int c, num;
:
:
while ( isdigit( c=getc(infile) ) )
    num = num*10 + c-48;
```

8.2.7 Testing for a Hexadecimal Digit

The **isxdigit** function tests for a hexadecimal digit; that is, a character that is either a decimal digit or an uppercase or lowercase letter in the range A to F. The function call has the form:

isxdigit (c)

where *c* is the character to be tested. The function returns a nonzero value if the character is a digit, otherwise it returns zero. For example, in the following program fragment, **isxdigit** tests whether a hexadecimal digit is read from the standard input:

```
int c;

c = getchar();
if ( isxdigit(c) )
    hexmode();
```

In this example, a function named *hexmode* is called if a hexadecimal digit is read.

8.2.8 Testing for Printable Characters

The **isprint** function tests for printable characters; i.e., characters whose ASCII values range from 32 to 126. The function call has the form:

isprint (c)

where *c* is the character to be tested. The function returns a nonzero (true) value if the character is printable, otherwise it returns zero (false).

8.3 Testing for Punctuation

The **ispunct** function tests for punctuation characters; i.e., characters that are neither control characters nor alphanumeric characters. The function call has the form:

ispunct (c)

where *c* is the character to be tested. The function returns a nonzero (true) function if the character is a punctuation character, otherwise it returns zero (false).



C Library Guide

8.3.1 Testing for Whitespace

The **isspace** function tests for whitespace characters; i.e, the space, horizontal tab, vertical tab, carriage return, formfeed, and newline characters. The function call has the form:

isspace (*c*)

where *c* is the character to be tested. The function returns a nonzero (true) value if the character is a whitespace character, otherwise it returns zero (false).

8.3.2 Testing for Case in Letters

The **isupper** and **islower** functions test for uppercase and lowercase letters, respectively. The function calls have the form:

isupper (*c*)

and

islower (*c*)

where *c* is the character to be tested. The function returns a nonzero (true) value if the character is the proper case, otherwise it returns zero (false).

For example, the function call:

isupper('b')

returns zero (false), but the call:

islower('b')

returns a nonzero (true) value.

8.3.3 Converting the Case of a Letter

The **tolower** and **toupper** functions convert the case of a given letter. The function calls have the form:

```
c = tolower (i)
```

and

```
c = toupper (i)
```

where *c* is the variable to receive the converted letter, and *i* is the letter to be converted. For example, the function call:

```
lower = tolower('B')
```

converts *B* to *b* and assigns it to the variable *lower*, and the call:

```
upper = toupper('b')
```

converts *b* to *B* and assigns it to the variable *upper*.

The **tolower** function returns the character unchanged if it is not an uppercase letter. Similarly, the **toupper** function returns the character unchanged if it is not a lowercase letter.

These functions are typically used to make the case of the characters read from a file or the standard input consistent. For example, in the following statement, **tolower** changes the character read from the standard input to lowercase before it is compared:

```
if ( tolower( getchar() ) != 'y' )
    exit(0);
```

This conversion allows the user to enter either *Y* or *y* to prevent the statement from executing the **exit** function.

8.4 Using the String Functions

The string functions concatenate, compare, copy, and keep track of the number of characters in a string. Two special string functions, **sscanf** and **sprintf**, let a program read from and write to a string in the same way the standard input and output can be read and written. These functions are convenient when reading or writing whole lines containing values of several different formats.

C Library Guide

Many string functions have two forms: a form that manipulates all characters in the string and one that manipulates a given number of characters. This gives programs very fine control over all or part of a string.

8.4.1 Concatenating Strings

The **strcat** function concatenates two strings by appending the characters of one string to the end of another. The function call has the form:

```
strcat (dst, src)
```

where *dst* is a pointer to the string to receive the new characters, and *src* is a pointer to the string containing the new characters. The function appends the new characters in the same order as they appear in *src*, then appends a null character ($\backslash 0$) to the last character in the new string. The function always returns the pointer *dst*.

The function is typically used to build a string such as a full pathname from two smaller strings. For example, in the following program fragment, **strcat** concatenates the string *temp* to the contents of the character array *dir*:

```
char dir[MAX] = "/usr/";  
strcat (dir, "temp");
```

8.4.2 Comparing Strings

The **strcmp** function compares the characters in one string to those in another and returns an integer value showing the result of the comparison. The **strcmp** function call has the form:

```
strcmp (s1, s2)
```

where *s1* and *s2* are the pointers to the strings to be compared. The function returns zero if the strings are equal (i.e., if they have the same characters in the same order). If the strings are not equal, the function returns the difference between the ASCII values of the first unequal pair of characters. The value of the second string character is always subtracted from the first. For example, the function call:

```
strcmp("Character A", "Character A");
```

returns zero, since the strings are identical in every way, but the function call :

```
strcmp("Character A", "Character B");
```

returns -1, since the ASCII value of *B* is one greater than *A*.

Note that the **strcmp** function continues to compare characters until a mismatch is found. If one string is shorter than the other, the function usually stops at the end of the shorter string. For example, the function call:

```
strcmp("Character A", "Character ")
```

returns 65, that is, the difference between the null character at the end of the second string and the *A* in the first string.

8.4.3 Copying a String

The **strcpy** function copies a given string to a given location. The function call has the form:

strcpy (dst, src)

where *src* is a pointer to the string to be copied, and *dst* is a pointer to the location to receive the string. The function copies all characters in the source string *src* to the *dst* and appends a null character ($\backslash 0$) to the end of the new string. If *dst* contained a string before the copy, that string is destroyed. The function always returns the pointer to the new string.

For example, in the following program fragment, **strcpy** copies the string “not available” to the location given by *name*:

```
char na[] = "not available";  
char name[20];  
  
strcpy( name, na );
```

Note that the location to receive a string must be large enough to contain the string. The function cannot detect overflow.

C Library Guide

8.4.4 Getting a String's Length

The **strlen** function returns the number of character contained in a given string. The function call has the form:

strlen (*s*)

where *s* is a pointer to a string. The count includes all characters up to, but not including, the first null character. The return value is always an integer.

In the following program fragment, **strlen** is used to determine whether or not the contents of *iname* are short enough to be stored in *name*:

```
char *iname;
char name[MAX];
:
:
if ( strlen(iname) < MAX )
    strcpy( name, iname);
```

8.4.5 Concatenating Characters to a String

The **strncat** function appends one or more characters to the end of a given string. The function call has the form:

strncat (*dst*, *src*, *n*)

where *dst* is a pointer to the string to receive the new characters, *src* is a pointer to the string containing the new characters, and *n* is an integer value giving the number of characters to be concatenated. The function appends the given number of characters to the end of the *dst* string, then returns the pointer *dst*.

In the following program fragment, **strncat** copies the first three characters in *letter* to the end of *cover*.

```
char cover[] = "cover";
char letter[] = "letter";

strncat( cover, letter, 3);
```

This example creates the new string *coverlet* in *cover*.

8.4.6 Comparing Characters in Strings

The **strncmp** function compares one or more pairs of characters in two given strings and returns an integer value which gives the result of the comparison. The function call has the form:

strncmp (s1, s2, n)

where *s1* and *s2* are pointers to the strings to be compared, and *n* is an integer value giving the number of characters to compare. The function returns zero if the first *n* characters are identical. Otherwise, the function returns the difference between the ASCII values of the first unequal pair of characters. The function generates the difference by subtracting the second string character from the first.

For example, the function call:

```
strncmp("Character A", "Character B", 5)
```

returns zero because the first five characters are identical, but the function call:

```
strncmp("Character A", "Character B", 11)
```

returns -1 because the value of *B* is one greater than *A*.

Note that the function continues to compare characters until a mismatch or the end of a string is found.

8.4.7 Copying Characters to a String

The **strncpy** function copies a given number of characters to a given string. The function call has the form:

strncpy (dst, src, n)

where *dst* is a pointer to the string to receive the characters, *src* is a pointer to the string containing the characters, and *n* is an integer value giving the number of characters to be copied. The function copies either the first *n* characters in *src* to *dst*, or if *src* has fewer than *n* characters, copies all characters up to the first null character. The function always returns the pointer *dst*.

C Library Guide

In the following program fragment, **strncpy** copies the first three characters in *date* to *day*.

```
char buf [MAX];
char date [29] = {"Fri Dec 29 09:35:44 EDT 1985"};
char *day = buf;

strncpy( day, date, 3);
```

In this example, *day* receives the string *Fri*.

8.4.8 Reading Values from a String

The **sscanf** function reads one or more values from a given character string and stores the values at a given memory location. The function is similar to the **scanf** function that reads values from the standard input. The function call has the form:

sscanf (s, format, argptr ...)

where *s* is a pointer to the string to be read, *format* is a pointer to the string defining the format of the values to be read, and *argptr* is a pointer to the variable that is to receive the values read. If more than one *argptr* is given, they must be separated with commas. The *format* string may contain the same formats as given for **scanf** (see **scanf(S)** in the *XENIX Programmer's Reference*). The function always returns the number of values read.

The function is typically used to read values from a string containing several values of different formats, or to read values from a program's own input buffer. For example, in the following program fragment, **sscanf** reads two values from the string pointed to by *datestr*:

```
char datestr[] =
    {"THU MAR 29 11:04:40 EST 1985"};
char month[4];
char year[5];

sscanf (datestr, "%*3s%3s%*2s%*8s%*3s%4s ",
        month, year);
printf("%s, %s\n", month, year);
```

The first value (a three-character string) is stored at the location pointed to by *month*, the second value (a four-character string) is stored at the location pointed to by *year*.

8.4.9 Writing Values to a String

The **sprintf** function writes one or more values to a given string. The function call has the form:

```
sprintf (s, format [, arg] ...)
```

where *s* is a pointer to the string to receive the value, *format* is a pointer to a string which defines the format of the values to be written, and *arg* is the variable or value to be written. If more than one *arg* is given, they must be separated by commas (.). The *format* string may contain the same formats as given for **printf** (see **printf(S)** in the *XENIX Programmer's Reference*). After all values are written to the string, the function adds a null character (\0) to the end of the string. The function normally returns zero, but will return a nonzero value if an error is encountered.

The function is typically used to build a large string from several values of different format. For example, in the following program fragment, **sprintf** writes three values to the string pointed to by *cmd*:

```
char cmd[100];
char *doc = "/usr/src/cmd/cp.c";
int width = 50;
int length = 60;

sprintf(cmd, "pr -w%d -l%d %s\n",
        width, length, doc);
system(cmd);
```

In this example, the string created by **sprintf** is used in a call to the **system** function. The first two values are the decimal numbers given by *width* and *length*. The last value is a string (a filename) and is pointed to by *doc*. The final string has the form:

```
pr -w50 -l60 /usr/src/cmd/cp.c
```

Note that the string to receive the values must have sufficient length to store those values. The function cannot check for overflow.



Chapter 9

Using Process Control

- 9.1 Introduction 9-1
- 9.2 Using Processes 9-1
- 9.3 Calling a Program 9-2
- 9.4 Stopping a Program 9-3
- 9.5 Starting a New Program 9-3
- 9.6 Executing a Program Through a Shell 9-7
- 9.7 Duplicating a Process 9-7
- 9.8 Waiting for a Process 9-8
- 9.9 Inheriting Open Files 9-9
- 9.10 Program Example 9-10

9.1 Introduction

This chapter describes the process control functions of the standard C library. The functions let a program call other programs, using a method similar to calling functions.

There are a variety of process control functions. The **system** and **exit** functions provide the highest level of execution control and are used by most programs that need a straightforward way to call another program or terminate the current one. The **execl**, **execv**, **fork**, and **wait** functions provide low-level control of execution and are for those programs which must have very fine control over their own execution and the execution of other programs. Other process control functions such as **abort** and **exec** are described in detail in section (S) of the *XENIX Programmer's Reference*.

The process control functions are a part of the standard C library. Since this library is automatically read when compiling a C program, no special library argument is required when invoking the compiler.

9.2 Using Processes

“Process” is the term used to describe a program executed by the XENIX system. A process consists of instructions and data, and a table of information about the program, such as its allocated memory, open files, and current execution status.

You create a process whenever you invoke a program through a shell. The system assigns a unique process ID to a program when it is invoked, and uses this ID to control and manage the program. The unique IDs are needed in a system running several processes at the same time.

You can also create a process by directing a program to call another program. This causes the system to perform the same functions as when it invokes a program through a shell. In fact, these two methods are actually the same method; invoking a program through a shell is nothing more than directing a program (the shell) to call another program.

The XENIX system handles all processes in essentially the same way, so the sections that follow should give you valuable information for writing your own programs and an insight into the XENIX system itself.

C Library Guide

9.3 Calling a Program

The **system** function calls the given program, executes it, and then returns control to the original program. The function call has the form:

```
system (command-line)
```

where *command-line* is a pointer to a string containing a shell command line. The command line must be exactly as it would be entered at the terminal; that is, it must begin with the program name followed by any required or optional arguments. For example, the call:

```
system("date");
```

causes the system to execute the **date** command, which displays the current time and date on the standard output. The call:

```
system("cat >response");
```

causes the system to execute the **cat** command. In this case, the standard output is redirected to the file *response*, so the command reads from the standard input and copies this input to the file *response*.

The **system** function is typically used in the same way as a function call; to execute a program and return to the original program. For example, in the following program fragment, **system** calls a program whose name is given in the string *cmd*:

```
char *name, *cmd;

printf("Enter filename: ");
scanf("%s", name);
sprintf(cmd, "cat %s ", name);
system(cmd);
```

Note that the string in *cmd* is built using the **sprintf** function and contains the program name **cat** and an argument (the filename read by **scanf**). The effect is to execute the **cat** command with the given filename.

When using the **system** function, it is important to remember that buffered input and output functions, such as **getc** and **putc**, do not change the contents of the buffer until it is ready to be read or flushed. If a program uses one of these functions, then executes a command with the **system** function, that command may read or write data not intended for its use. To avoid this problem, the program should clear all buffered input and output before making a call to the **system** function. You can do this for output with the **fflush** function, and for input with the **setbuf** function described in the section "Using More Stream Functions" in "Using the Standard I/O Functions."

9.4 Stopping a Program

The **exit** function stops program execution by returning control to the system. The function call has the form:

```
exit (status)
```

where *status* is the integer value to be sent to the system as the termination status.

The function is typically used to terminate a program before its normal end, such as after a serious error. For example, in the following program fragment, **exit** stops the program and sends the integer value "2" to the system if the **fopen** function returns the null pointer value NULL.

```
FILE *ttyout;
  ⋮
if ( fopen(ttyout, "r") == NULL )
    exit(2);
```

Note that the **exit** function automatically closes each open file in the program before returning to the system. This means no explicit calls to the **fclose** or **close** functions are required before an **exit**.

9.5 Starting a New Program

The **exec** family of functions cause the system to overlay the calling program with the given one, allowing the calling program to terminate while the new program continues execution. There are several different forms of **exec**; the differences are summarized in Table 4.1. The function names are given in the first column. The second column specifies whether the current **PATH** setting (see **environ** (M) in the *XENIX User's Reference*) is used to locate the file to be executed as the child process.

The third column describes the method for passing arguments to the child process. Arguments may be passed as a list or as an array: in an argument list, the arguments to the child process are listed as separate arguments in the **exec** call; in an argument array, the arguments are stored in an array, and a pointer to the array is passed to the child process. The argument-list method is typically used when the number of arguments is constant or is known at compile time, while the argument-array method is useful when the number of arguments must be determined at run time.

C Library Guide

The last column specifies whether the child process inherits the environment settings of its parent or whether a table of environment settings can be passed to set up a different environment for the child process.

Table 9.1
Forms of the exec Routine

Routine	Use of PATH Setting	Argument-Passing Convention	Environment
execl	Does not use PATH	Argument list	Inherited from parent
execle	Does not use PATH	Argument list	Pointer to environment table for child process passed as last argument
execlp	Uses PATH	Argument list	Inherited from parent
execv	Does not use PATH	Argument array	Inherited from parent
execve	Does not use PATH	Argument array	Pointer to environment table for child process passed as last argument
execvp	Uses PATH	Argument array	Inherited from parent

The **exec** functions return control to the original program only if there is an error in finding the given program, such as a misspelled pathname or lack of execute permission. This allows the original program to check for errors and display an error message, if necessary.

Note that the **exec** functions will not expand metacharacters (<, >, ?, and []) given in the argument list. If a program needs these features, simply use an **exec** function to call a shell and let the shell execute the command you want.

Examples: `execl` and `execv`

The `execl` function call has the form:

```
execl (pathname, command-name, argptr ...)
```

pathname is a pointer to a string containing the full pathname of the command you want to execute, *command-name* is a pointer to a string containing the name of the program you want to execute, and *argptr* is one or more pointers to strings which contain the program arguments. Each *argptr* must be separated from any other argument by a comma. The last *argptr* in the list must be the null pointer value NULL. For example, in the call:

```
execl("/bin/date", "date", NULL);
```

the **date** command, whose full pathname is “/bin/date”, takes no arguments, and in the call:

```
execl("/bin/cat", "cat", file1, file2, NULL);
```

the **cat** command, whose full pathname is “/bin/cat”, takes the pointers “file1” and “file2” as arguments.

The `execv` function call has the form:

```
execv (pathname, ptr);
```

where *pathname* is the full pathname of the program you want to execute, and *ptr* is a pointer to an array of pointers. Each element in the array must point to a string. The array may have any number of elements, but the first element must point to a string containing the program name, and the last must be the null pointer, NULL.

The `execl` and `execv` functions are typically used in programs that execute in two or more phases and communicate through temporary files (for example a two-pass compiler). The first part of such a program can call the second part by giving the name of the second part and the appropriate

C Library Guide

arguments. For example, the following program fragment checks the status of "errflag," then either overlays the current program with the program *pass2*, or displays an error message and quits:

```
char *tmpfile;
int errflag;
.
.
.
if (errflag == 0)
    execl("/usr/bin/pass2", "pass2", tmpfile,
        NULL);
else {
    fprintf(stderr, "Error %d: Quitting",
        errflag);
    exit(2);
}
```

The `execv` function is typically used to pass arguments to a program when the precise number of arguments is not known beforehand. For example, the following program fragment reads arguments from the command line (beginning with the third one), copies the pointer of each to an element in *cmd*, sets the last element in *cmd* to NULL, and executes the `cat` command.

```
char *cmd[ ];
int i;

cmd[0] = "cat";
for (i=3; i<argc; i++)
    cmd[i] = argv[i];
cmd[argc] = NULL;

execv("/bin/cat", cmd);
```

The `execl` and `execv` functions return control to the original program only if there is an error in finding the given program (e.g., a misspelled path-name or no execute permission). This allows the original program to check for errors and display an error message if necessary. For example, the following program fragment searches for the program *display* in the */usr/bin* directory:

```
execl("/usr/bin/display", "display", NULL);
fprintf(stderr, "Can't execute 'display' \n");
```

If the program *display* is not found or lacks the necessary permissions, the original program resumes control and displays an error message.

9.6 Executing a Program Through a Shell

A drawback of the `execl` and `execv` functions is that they do not provide the metacharacter features of a shell. One way to overcome this problem is to use `execl` to execute a shell and let the shell execute the command you want.

The function call has the form:

```
execl ("/bin/sh", "sh", "-c", command-line, NULL);
```

where *command-line* is a pointer to the string containing the command line needed to execute the program. The string must be exactly as it would appear if it were entered at the terminal.

For example, a program can execute the command:

```
cat *.c
```

(that contains the metacharacter `*`) with the call:

```
execl ("/bin/sh", "sh", "-c", "cat *.c", NULL);
```

In this example, the full pathname `/bin/sh` and command name `sh` start the shell. The argument `-c` causes the shell to treat the argument `cat *.c` as a whole command line. The shell expands the metacharacter and displays all files which end with something that the `cat` command cannot do by itself.

9.7 Duplicating a Process

The `fork` function splits an executing program into two independent and fully-functioning processes. The function call has the form:

```
fork ()
```

No arguments are required.

The function is typically used to make multiple copies of any program that must take divergent actions as a part of its normal operation; e.g., a program that must use the `execl` function, yet still continue to execute. The original program, called the “parent” process, continues to execute normally, just as it would after any other function call. The new process, called the “child” process, starts its execution at the same point, that is, just after the `fork` call. (The child never goes back to the beginning of the program to start execution.) The two processes are in effect synchronized, and continue to execute as independent programs.

C Library Guide

The **fork** function returns a different value to each process. To the parent process, the function returns the process ID of the child. The process ID is always a positive integer and is always different than the parent's ID. To the child, the function returns 0. All other variables and values remain exactly as they were in the parent.

The return value is typically used to determine which steps the child and parent should take next. For example, in the following program segment:

```
char *cmd;
  ⋮
if (fork() == 0)
    execl("/bin/sh", "sh", "-c", cmd, NULL);
```

the child's return value, 0, causes the expression "fork() == 0" to be true, and therefore the **execl** function is called. The parent's return value, on the other hand, causes the expression to be false, and the function call is skipped. Executing the **execl** function causes the child to be overlaid by the program given by *command*. This does not affect the parent.

If **fork** encounters an error and cannot create a child, it will return the value -1. It is a good idea to check for this value after each call.

9.8 Waiting for a Process

The **wait** function causes a parent process to wait until its child processes have completed their execution before continuing its own execution. The function call has the form:

wait (*ptr*)

where *ptr* is a pointer to an integer variable. It receives the termination status of the child from both the system and the child itself. The function normally returns the process ID of the terminated child, so the parent may check it against the value returned by **fork**.

The function is typically used to synchronize the execution of a parent and its child, and is especially useful if the parent and child processes access the same files. For example, the following program fragment

causes the parent to wait while the program named by *pathname* (which has overlaid the child process) finishes its execution:

```
int status;
char *pathname;
char *cmd[ ];
  ⋮
if (fork() == 0)
    execv(pathname, cmd);
wait(&status);
```

The **wait** function always copies a status value to its argument. The status value is actually two 8-bit values combined into one. The low-order 8 bits contains the termination status of the child as defined by the system. This status is zero for normal termination and nonzero for other kinds of termination, such as termination by an interrupt, quit, or hangup signal (see **signal(S)** in the *XENIX Programmer's Reference* for a description of the various kinds of termination). The next 8 bits contains the termination status of the child as defined by its own call to **exit**. If the child did not explicitly call the function, the status is zero.

9.9 Inheriting Open Files

Any program called by another program or created as a child process to a program automatically inherits the original program's open files and standard input, output, and error files. This means that if the file was open in the original program, it will be open in the new program or process.

A new program also inherits the contents of the input and output buffers used by the open files of the original program. To prevent a new program or process from reading or writing data that is not intended for its use, these buffers should be flushed before calling the program or creating the new process. A program can flush an output buffer with the **fflush** function, and an input buffer with **setbuf**.

C Library Guide

9.10 Program Example

This section shows how to use the process control functions to control a simple process. The following program starts a shell on the terminal given in the command line. The terminal is assumed to be connected to the system through a line that has not been enabled for multi-user operation.

```
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[ ];
{
    int status;

    if (argc < 2) {
        fprintf(stderr, "No tty given.\n");
        exit(1);
    }
    if (fork() == 0) {
        if (freopen(argv[1], "r", stdin) == NULL)
            exit(2);
        if (freopen(argv[1], "w", stdout) == NULL)
            exit(2);
        if (freopen(argv[1], "w", stderr) == NULL)
            exit(2);
        execl("/bin/sh", "sh", NULL);
    }
    wait(&status);
    if (status == 512)
        fprintf("Bad tty name: %s\n", argv[1]);
}
```

In this example, the **fork** function creates a duplicate copy of the program. The child changes the standard input, output, and error files to the new terminal by closing and reopening them with the **freopen** function. The terminal name pointed to by *argv* must be the name of the device special file associated with the terminal, e.g., `"/dev/tty03"`. The **execl** function then calls the shell which uses the new terminal as its standard input, output, and error files.

The parent process waits for the child to terminate. The **exit** function terminates the process if an error occurs when reopening the standard files. Otherwise, the process continues until the Ctrl-D key is pressed on the terminal keyboard.

Chapter 10

Using the Event Manager

- 10.1 Introduction 10-1
- 10.2 Using the Event Manager 10-1
 - 10.2.1 Operation and Semantics 10-1
 - 10.2.2 Event Generating Hardware 10-2
- 10.3 Events 10-3
 - 10.3.1 Using the Event Queue 10-3
 - 10.3.2 Structure of an Event 10-4
 - 10.3.3 Event Field Macros 10-4
 - 10.3.4 Keyboard Events 10-5
- 10.4 Event Manager Calls 10-5
 - 10.4.1 Debugging 10-9
- 10.5 Configuration Files 10-10
 - 10.5.1 Event-Devices Configuration File 10-10
 - 10.5.2 Event-Terminals Configuration File 10-12
- 10.6 Event Manager C Language Definitions and Syntax 10-13
- 10.7 Summary of Event Manager C Syntax 10-14
 - 10.7.1 Event API function prototypes 10-14
 - 10.7.2 Function Invocation Syntax 10-14
- 10.8 A Sample Program 10-18

10.1 Introduction

XENIX supports development of applications using mice and other graphics input devices. Data from graphics input devices like mice is received by the application in the form of “events.” An event represents a movement or action on the part of the graphics input device. For example, moving a mouse generates a series of signals from the mouse that are interpreted as discrete events. An “up” motion of the mouse is interpreted as an “up” event. Events are accumulated in, and read from, an “event queue.” Mouse support is provided by a series of utilities and system services collectively known as the event manager. The event manager consists of kernel functionality to read and manage graphic input device data, an event driver, and an Advanced Programmatic Interface (API).

The programmatic interface allows applications software to be portable to systems with different event devices, such as a different brand of mouse or a mouse attached to a different port. The interface provides routines for opening and accessing an event queue. A variety of graphics input devices, such as mice, bitpads or the keyboard, may be used with the event driver.

Since speed is of top priority in graphics applications, the event driver and the system routines are optimized for speed of access.

10.2 Using the Event Manager

This section describes four topics: operation and semantics, devices, API routines, and the event configuration files. This covers the basics of program development with mouse support.

10.2.1 Operation and Semantics

To open an event queue, a process makes a library call. The process must indicate the classes (types) of devices it wants to use. Common classes for devices include “relative locator” devices (mice) or “absolute locator” devices (bitpads). The event manager interface then checks the configuration files to determine what physical devices are allowed to give input to the process’s terminal. The devices listed in the configuration file are “associated” with that terminal. Associated devices of the proper class are opened for events with the initial call from the process. When a device is opened for events, data from the device is translated to events, and enqueued into the process’s event queue.

C Library Guide

Every event device is associated with some terminals or console multiscreens. To use an event device with the console, it should be associated with each multiscreen. In order for an application to use event devices, the devices must be associated with the terminal that invoked the application. It is possible for programs that are not running on any terminal to use the mouse, but they cannot use the API.

An event device is said to be “attached” to an event queue if the device has been opened and input from the device is being received by the event queue. Note that an application can choose whether or not to attach any associated event device to its event queue while it is running. For example if both a bitpad and a mouse are associated with a terminal, it is possible for the application to open only the mouse for input, and then close the mouse and open the bitpad without exiting.

10.2.2 Event Generating Hardware

Event devices work in one of three ways. These are relative locator, absolute locator, and string. Most mice and trackballs are examples of relative locator devices. They are termed “relative” devices because the mouse generates events when it is moved relative to its previous position. The motion of the mouse generates the events, not the final position of the mouse. Relative locations are reported as signed 32 bit quantities. Bit pads and light pens are absolute locator devices. The pens are used on a sensitive pad, and the events are generated by the absolute position of the pen on the pad, not the relative motion of the pen. Absolute locations are reported as unsigned 32 bit quantities. The keyboard is a string graphics input device. This means that the events will be generated by pressing a key on the keyboard, such as an arrow key. Some devices can operate in multiple modes. For example, some mice can function in bitpad mode with the appropriate bitpad software.

Many event devices have buttons; others do not. Three-button devices are viewed as having a left-button, a middle-button, and a right-button. Two button devices do not have a middle button. One button devices have only a middle button. Bits representing buttons are set to 1 when the button is depressed. Up to eight buttons can be read at a time.

Under XENIX, devices are categorized like this:

1. This is a relative locator device such as a mouse.
2. This is an absolute locator device such as a bitpad.
3. This is a device which generates a character stream.

4. This is any event device not within the previous categories.

10.3 Events

10.3.1 Using the Event Queue

This interface allows an application to obtain sequential input from any number of event devices. Devices like mice, trackballs, bitpads, and the keyboard can be assigned to one or more terminals or multiscreens. When a process opens an event queue with more than one device attached, input from all the devices is intermixed in the event queue. This allows a process to “sleep on multiple devices” without recourse to the `select(S)` system call.

An application makes a call to a library routine to open an event queue. That call attaches the event devices to the event queue. In general, an application may be using multiple devices. For example, an application would typically want to use a mouse and the keyboard. In cases where there are several devices attached to the terminal, for example multiple mice, calls are provided for listing the attached devices and including them or excluding them on an individual basis.

The event driver maintains a circular queue of events. “Circular” means that if the total number of events in the queue exceeds the maximum number of events storable, that the oldest event is removed from the queue and an error condition is returned.

The event queue is ‘fed’ by all the devices attached to the queue. An input mask may also be set which prevents certain classes of events from being inserted into the queue. For example, you may wish to exclude button events or movement events during some portions of your program.

The application program reads the event queue by getting a pointer to the next event. When the application has copied or finished examining the event, it pops the event queue using `ev_pop`. The next `ev_read` call will return a pointer to the next event in the queue, or `NULL` if the queue is empty. Multiple reads which are not separated by a call to pop the queue, return the same pointer.

C Library Guide

10.3.2 Structure of an Event

Events are timestamped as they enter the queue. The timestamp is the time of day in milliseconds when the event was enqueued. This timestamp is accurate to the extent of the interrupt timer rate which, for AT class machines, is 50Hz.

Events have a tag field indicating the type of the event. There are four kinds of events: relative locator, absolute locator, string, and other.

These four types of events have three possible event formats. string type events and other type events have the same structure. An event structure is a union of the possible types of events. Absolute and relative locator events have a byte for button state (up or down), four bytes each for X and Y coordinates, and space for additional information (currently unused). String events and other events use a character array to store the information, along with a byte for the count.

10.3.3 Event Field Macros

These macros allow the application to take apart an event without knowing its internal structure. The parameter to these macros is a pointer to an event. Here are the macro definitions:

```
/* Values for event tag */
#define T_OTHER
#define T_BUTTON
#define T_STRING
#define T_ABS_LOCATOR
#define T_REL_LOCATOR

/* Bit definitions for the buttons */
#define BUTTON1 0x01
#define BUTTON2 0x02
#define BUTTON3 0x08
#define RT_BUTTON BUTTON1
#define MD_BUTTON BUTTON2
#define LT_BUTTON BUTTON3

/* Extracting fields */
#define EV_TIME(x) ((x).timestamp)
#define EV_TAG(x) ((x).tag) /* device making event*/
#define EV_BUFCNT(x) ((x).ul.bufcnt) /* num bytes in buffr */
```

```
#define EV_BUTTONS(x)    ((x).ul.buttons)
#define EV_BUF(x)        ((x).un.buf)          /* pointer to buffer */
#define EV_DX(v)         ((v).un.loc.rel.dx)  /* Relative X coordinate */
#define EV_DY(v)         ((v).un.loc.rel.dy)  /* Relative Y coordinate */
#define EV_X(v)          ((v).un.loc.abs.x)   /* Absolute X coordinate */
#define EV_Y(v)          ((v).un.loc.abs.y)   /* Absolute Y coordinate */
```

10.3.4 Keyboard Events

When the keyboard is being used to generate events, it is at least in raw mode. It may, in fact, be in scan-code pass through mode. It is up to the application to put it into scan code pass-through mode if that is desired. The default mode is raw mode.

An application reading keyboard events should not assume that keyboard events are always one character. The application should check the *bufcnt* field in the event to see how many bytes of data the event contains.

10.4 Event Manager Calls

The event manager consists of routines for managing and accessing an event queue and the attached devices. This section lists and describes the routines. All of the event manager routines which return an integer return a negative number if they fail. All of these routines with the exception of *ev_init()* and *ev_open()* fail if the calling program does not have an open event queue. *ev_init* opens the event queue for the calling program.

The constants and types used in the event manager are defined in */usr/include/mouse.h*.

Event Manager Library Calls

<i>ev_init</i>	Mandatory initialization (first routine invoked)
<i>ev_open</i>	Open the event queue, attach event devices
<i>ev_close</i>	Close the event queue and all attached devices

C Library Guide

<code>ev_count</code>	Return the number of events in the queue
<code>ev_read</code>	Get a pointer to the 'top' event
<code>ev_pop</code>	Pop the 'top' event off the queue
<code>ev_block</code>	Sleep until the queue is nonempty
<code>ev_flush</code>	Pop all the events
<code>ev_getdev</code>	Get a list of devices feeding the queue
<code>ev_gindev</code>	Exclude or later re-include a event device
<code>ev_setemask</code>	Mask out certain kinds of events
<code>ev_getemask</code>	Get the current event mask
<code>ev_suspend</code>	Suspend the active event queue (make it inactive)
<code>ev_resume</code>	Resume the suspended event queue

`ev_init()` This must be the first event routine invoked. It initializes the event queue. It reads information from the configuration files into memory. It reports an error if there is a syntax error or inconsistency in the configuration files. If there is an error, it returns a small negative integer, otherwise it returns 0, indicating successful initialization.

`ev_open()` This routine opens an event queue for the process. All of the event devices which are associated with the terminal, and whose class is masked in, are opened and added to the queue.

The *ev_open* routine takes an argument which is a pointer to a device mask. The mask is a bitwise ORing of any or all of the four values: **D_STRING**, **D_REL**, **D_ABS**, and **D_OTHER**. The function tries to attach devices of the indicated type(s) to the event queue. Upon exit it sets the mask to indicate what kinds of devices it has found. If *ev_open()* cannot find any devices to attach to the queue, it closes the queue and returns a negative number indicating an error.

This routine returns a file descriptor if it succeeds. The file descriptor is used with the **select(S)** system call. It should not be used for reading or writing.

`ev_close()` Close the event queue and all associated devices. This routine takes no arguments.

`ev_count()` returns the number of events in the queue. This routine takes no arguments.

ev_read() This routine returns a pointer to the next element in the event queue or NULL if the queue is empty. Multiple calls to this routine return the same pointer until a call to *ev_pop* is made. The routine takes no arguments.

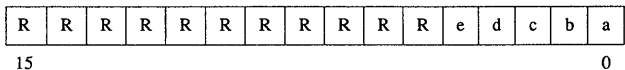
ev_pop() This routine pops the next element off the queue. As soon as this routine is called the pointer returned by the most recent call to *ev_read* must be considered invalid as the event driver may overwrite the pointed area with new information.

The routine does not take any arguments. It fails if the queue is empty. Otherwise, it returns a number indicating how many events have been lost to queue overrun since the last *ev_pop()*. For explanation, the queue is of fixed size and events may be lost if the application does not read the queue fast enough. That condition is called “queue overrun” because the event driver would overrun the tail of the queue if it wrote new events. The event driver provides a counter indicating the number of events it has lost. That counter is cleared when it succeeds in writing an event. A *pop()* always causes a new slot to be available, which clears the overrun counter. The *pop()* routine returns the old overrun counter so that an application is aware if events are being lost.

ev_flush() This routine discards all the events currently in the queue. It does not take any arguments.

ev_setemask(event_mask)

This routine sets the current event mask. Events that are of a type that is currently masked in are allowed on the queue. The 16 bits of the mask have the following definitions:



Where:

C Library Guide

R indicates Reserved
a indicates "Other" Device Events
b indicates Button Events
c indicates String events
d indicates Relative Locator movement events
e indicates Absolute Locator movement events

`ev_getemask(emask_t *)`

This routine takes an argument which is a pointer to an event mask. It fills in the mask pointed to with the event queue's current event mask.

`ev_getdev(dmask_t, struct devinfo *)`

This routine is used to get information about the devices attached to the event queue. The argument to this routine is a device mask made from ORing together any or all of `D_ABS`, `D_REL`, `D_STRING`, and `D_OTHER`, and a pointer to a `struct devinfo`. The function returns a pointer to the *next* device whose type is masked in. Consecutive calls will iterate through all the devices attached to the queue. The function should be called initially with `(struct devinfo *) NULL` and thereafter with the pointer it returns until it returns `NULL`. `getdev()` returns `NULL` if there are no devices of the requested class(es) present.

The routine returns a pointer to a `struct devinfo` which is defined as follows:

```
struct devinfo {  
    short handle;      /* not used by application */  
    short class;      /* REL, ABS, STRING or OTHER */  
    short type;       /* Number encoding the type of hardware */  
    char *name;       /* Device name, from configuration files */  
    char *key;        /* Device key, from configuration files */  
    short buttons;    /* number of buttons on the mouse */  
}
```

Using the pointer returned by this routine and the `ev_gindev()` routine, an application can exclude a device from feeding a queue, or later re-include it.

If a terminal's (or multiscreen's) keyboard is attached to an event queue, then keyboard input will not be

available from the normal keyboard device. The keyboard is a class `STRING` device always implicitly assigned to a `tty`, and thus, is attached whenever `D_STRING` devices are requested, even if it is not listed in the configuration files.

`ev_gindev(struct devinfo *, char)`

This routine is used to exclude or re-include a event device feeding the event queue. The pointer is obtained from a `getdev` call. The second argument has one of two values, **INCLUDE** or **EXCLUDE**, which are defined in `/usr/include/mouse.h`.

`ev_suspend()` This call makes the open event queue inactive. If a process with an event queue forks a shell and wait for the subshell to complete, it should first suspend its event queue. That way an event queue can successfully be opened from within the forked shell. When the shell returns, the suspended event queue should be resumed. This routine does not take any arguments.

`ev_resume()` This call complements `ev_suspend`. It restarts a suspended queue. This routine does not take any arguments.

The `ev_suspend` and `ev_resume` routines are present because one terminal (or multiscreen) cannot have two active event queues. Having more than one open queue leads to a situation where the input data destination is indeterminate. Whenever multiple applications are running concurrently on one terminal (or screen), one of the event queues must be suspended.

10.4.1 Debugging

The event manager maintains an error-level variable which may be set by the application. The variable is an integer named `ev_errlev` and it defaults to zero. When it is nonzero, the event manager prints out more verbose diagnostics. This may be helpful during debugging.

C Library Guide

10.5 Configuration Files

The two configuration files used by the Event Manager are */usr/lib/event/devices* and */usr/lib/event/ttys*. The former file indicates information about the event devices attached to the system. The latter file associates event devices with particular terminals or multiscreens.

In both files, blank lines and lines beginning with a “#” character in the first column are comments. Entries may be continued onto subsequent lines by placing a backslash (\) at the end of the line.

10.5.1 Event-Devices Configuration File

Entries in the devices file have the following format:

```
key device class type [ parm=value ... ]
```

Where:

1. “*key*” is a unique identifier used to refer to this entry. The use of this key is analogous to the single-letter identifier in **gettydefs(F)**. Keys are at most 20 characters.

A device can be present in this file in different modes by giving it multiple entries with unique identifiers. The second configuration file uses these identifiers to select devices.

2. “*device*” is the absolute pathname of the device file. For a serial device it may be the tty device or the equivalent entry in */dev/mouse*; for a busmouse it is */dev/mouse/busmouse0* or */dev/mouse/busmouse1*.
3. “*class*” contains one of **ABS**, **REL**, **STRING**, or **OTHER**. It may have a “**b**” appended to indicate that the device can generate button events. This field is used by the library routines to determine what kind of device is present. This information is checked when a program opens an event queue.
4. “*type*” indicates exactly what kind of device this is. This information is passed into the driver so that it knows how to build the events. The keyword in this field *must* be present in the event-devices table in **master(F)**. Supported keywords in this field include:

Keyword	Device
keyboard	the keyboard
mousems	Microsoft serial mouse
busmouse	Any busmouse
mousepc	Mouse Systems serial mouse
mouse0	serial mouse mode 0
mouse1	serial mouse mode 1
mouse2	serial mouse mode 2
mouse3	serial mouse mode 3
mouse4	serial mouse mode 4
mouse5	serial mouse mode 5
mouse6	serial mouse mode 6

5. “*parm*” is a set of *string=value* pairs which provide hardware dependent information. The following keyword strings may be recognized:

- “**STTY=**” Valid only for serial devices, provides information about the serial characteristics of the line. If there are multiple items specified, they should be quoted by double quotes and separated by spaces as in **STTY=“CS7 1200”**.
- “**INIT=**” Valid only for serial devices, defines an initialization string which is sent to the device after the serial characteristics of the line are established. The user must be able to open the device for writing.

If the initialization string contains spaces or tabs it must be quoted. Nonprinting characters may be embedded using escaped octal notation, such as “\033” for the Escape character.

Serial devices having multiple modes must have an initialization string present to put them into the desired mode. No error will be reported if there is no initialization string, but there is no guarantee that the device will operate in the desired mode.

- “**SENSITIVITY=**” This provides a mechanism whereby locator devices with varying sensitivities and scales can be made to behave uniformly. The *value* is a hexadecimal number. Locations are multiplied by this *value* as they enter the event queue. Then they are divided (by shifting) by 0x2000. Using this mechanism, a device can be made to appear twice as sensitive by setting **SENSITIVITY=4000**

C Library Guide

or three-fourths as sensitive by setting **SENSITIVITY=1800**.

Possible ratios include:

Sample Values for **SENSITIVITY**

SENSITIVITY (hex)	800	1000	1800	2000	2800	3000	3800	4000	6000	8000
effective ratioing achieved	1/4	1/2	3/4	1	5/4	3/2	7/4	2	3	4

- “**NAME=**” This gives the device a name which is available to the user or applications software. It may be anything the system administrator chooses and has no intrinsic definition. It is not passed into the driver. For example: **NAME=“Microsoft serial mouse”** is an acceptable entry.

10.5.2 Event-Terminals Configuration File

This file lists the terminal to event device associations of the system.

Entries in this file are of the form:

device key [key ...]

where

“*device*” is the filename of a terminal device or console multiscreen

“*key*” is the key identifier of a event device described in the *devices* file.

10.6 Event Manager C Language Definitions and Syntax

```

typedef short dmask_t; /* device mask type */
typedef short emask_t; /* event mask type */

/* Bit Values for dmask_ts */

#define D_OTHER
#define D_REL
#define D_ABS
#define D_STR

/* Values for event mask */

#define T_OTHER
#define T_BUTTON
#define T_STRING
#define T_ABS_LOCATOR
#define T_REL_LOCATOR
#define T_LOCATOR

/* Values for action in ev_gindex() */

#define EXCLUDE 0
#define INCLUDE 1

int ev_open(dmask_t *);
int ev_close();
int ev_count();
EVENT* ev_read();
int ev_pop();
int ev_flush();
int ev_block();
struct gindex * ev_getdev(dmask_t, struct gindex *);
int ev_gindex(struct gindex *, char);
int ev_setemask(emask_t);
emask ev_getemask(emask_t *);
int ev_suspend();
int ev_resume();

```


C Library Guide

10.7 Summary of Event Manager C Syntax

The following section gives the correct C language syntax for calling any of the Event Manager functions.

10.7.1 Event API function prototypes

The following declarations indicate the correct syntax for declaring the Event Manager functions.

```
int ev_init(void);
int ev_open(unsigned short *);
int ev_close(void);
int ev_count();
int ev_block(void);
EVENT *ev_read(void);
int ev_pop(void);
int ev_flush(void);
int ev_setemask(unsigned short);
int ev_getemask(unsigned short *);
struct devinfo *ev_getdev(unsigned short, struct devinfo *);
int ev_gindev(struct devinfo *, char);
int ev_suspend(void);
int ev_resume(void);
```

10.7.2 Function Invocation Syntax

The following code fragments indicate the correct syntax for using any of the Event Manager system calls.

To initialize the event library:

```
{
    ...
    /* Read the system event configuration files */
    ev_init();
    ...
}
```

To open an event queue with the keyboard and a mouse:

```
{
    /* Declare the device mask, open queue
       with keyboard and mouse */
    dmask_t dmask = D_REL | D_STRING;
    /* Try to open the event queue */
    ret = ev_open(&dmask);
    if ( ret < 0 || dmask != D_REL | D_STRING )
        fail();
}
```

To close the event queue:

```
{
    ...
    ev_close();
    ...
}
```

To block until there's an event in the queue:

```
{
    ...
    ret = ev_block();
    ...
}
```

To read an event:

```
{
    EVENT *evp, *ev_read();
    ...
    evp = ev_read();
    if (evp == (EVENT*) NULL)
        /* no events available */
    ...
}
```

To remove the last read event from the queue:

```
{
    ...
    ev_pop();
    ...
}
```

The recommended methodology is for `ev_block()`, `ev_read()` and `ev_pop()` to be used together, as in the following example.

C Library Guide

Block until an event is available, and then read it:

```
{
    EVENT      *evp;
    ...
    ev_block();
    evp = ev_read();
    /* Examine, copy, or process event */
    ev_pop();
    ...
}
```

To count the number of events in the event queue:

```
{
    ...
    ev_count();
    ...
}
```

To flush the event queue:

```
{
    ...
    ev_flush();
    ...
}
```

To get the current event mask:

```
{
    emask_t    emask;
    ...
    ev_getemask(&emask);
    ...
}
```

To set the event mask to only permit keyboard events:

```
{
    ...
    ev_setemask(T_STRING);
    ...
}
```

To cycle through the devices attached to the event queue:

```
{
    struct devinfo * devp = NULL, *ev_getdev();
    dmask_t dmask = D_REL | D_STRING;
    ...
    while ( (devp = ev_getdev(dmask, devp)) != NULL )
        /* examine the structure returned */
    ...
}
```

To exclude a device from feeding the queue. A pointer to the device was previously obtained with `ev_getdev()`:

```
{
    struct devinfo *devp;
    ...
    ev_gindex(devp, EXCLUDE);
}
```

To suspend our event queue:

```
{
    ev_suspend();
}
```

And to resume it:

```
{
    ev_resume();
}
```

C Library Guide

10.8 A Sample Program

The following sample program opens an event queue with a mouse and the keyboard. Note that depending on your system, you may have to configure system event parameters. The program prints events as they enter the queue:

```
/*
 * This program opens an event queue and with a relative device (mouse)
 * and a string device (keyboard). It prints out information about
 * events as they enter the queue.
 *
 * Since the keyboard is put into the event line discipline, we
 * lose line discipline zero functionality like signals. Therefore
 * the DEL key loses its special meaning. So we test each
 * keyboard character against DEL and finish when we see one.
 */
#include <fcntl.h>
#include <sys/termio.h>
#include <stdio.h>
#include <signal.h>
#include <sys/machdep.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

#define DEL 0x7f

char  progname[80];
int   kfd=-1,mfd=-1;
struct termio ksave;
extern int errno;

main(argc,argv)
int  argc;
char *argv;
{
    int   qfd, finish(), report();
    long  time=-1;
    EVENT *evp;
    char  c;
    int   ret;
    cmask_t cmask = D_STRING | D_REL;
    extern int ev_errlev;

    ev_errlev = 1; /* Turn on event manager diagnostics */
    strcpy(progname,argv[0]);
    signal(SIGINT,finish);
    signal(SIGSEGV,report);
    signal(SIGSYS,report);
}
```

```

printf("\n");
ret = ev_init();
printf('\ninit == %d\n', ret);
if ( ret < 0 )
    finish();
qfd = ev_open(&dmask);
printf("open == %d\n", qfd);
if ( qfd < 0 )
    fail("could not open event queue");
if ( dmask != D_STRING | D_REL )
    fail("could not attach mouse and keyboard");
/* Events should be enqueued now */
while ( ev_block() )
    if ( (evp = ev_read()) != (EVENT *) NULL ) {
        evprint(evp);
        ev_pop();
    }
return 0;
}

fail(s)
char *s;
{
    extern int errno;

    printf("%s(%d): %s\n",programe,errno,s);
    finish();
    return 0;
}

evprint(evp)
EVENT *evp;
{
    static long time=-1;

    if (time == -1)
        time = evp->timestamp;
    if (EV_TAG(*evp) & T_STRING) {
        printf("time(%d) tag(%d) bufsize(%d) buf(%x) \n",
            EV_TIME(*evp)-time,
            EV_TAG(*evp),
            EV_BUFCNT(*evp),
            EV_BUF(*evp)[0]);
        if (EV_BUF(*evp) == DEL)
            finish();
    }
}

```

C Library Guide

```
    else {
        printf("time(%ld) tag(%d) butts(%d) x(%ld) y(%ld)\n",
            EV_TIME(*evp) - time,
            EV_TAG(*evp),
            EV_BUTTONS(*evp),
            EV_DX(*evp),
            EV_DY(*evp));
        if (EV_BUTTONS(*evp) == 7)
            finish();
    }
    return 0;
}

finish()
{
    ev_close();
    printf("done\n");
    exit(0);
}

report(s)
int s;
{
    printf("got signal %d\n", s);
    finish(0);
}
```

Chapter 11

Writing and Using Pipes

- 11.1 Introduction 11-1
- 11.2 Opening a Pipe to a New Process 11-1
- 11.3 Reading and Writing to a Process 11-2
- 11.4 Closing a Pipe 11-3
- 11.5 Opening a Low-Level Pipe 11-3
 - 11.5.1 Reading and Writing to a Low-Level Pipe 11-4
 - 11.5.2 Closing a Low-Level Pipe 11-5
- 11.6 Program Examples 11-5
- 11.7 Named Pipes 11-8

11.1 Introduction

A pipe is an artificial file that a program may create and use to pass information to other programs. A pipe is similar to a file in that it has a file pointer and/or a file descriptor and can be read from or written to using the input and output functions of the standard library. Unlike a file, a pipe does not represent a specific file or device. Instead, a pipe represents temporary storage in memory that is independent of the program's own memory and is controlled entirely by the system.

Pipes are chiefly used to pass information between programs, just as the shell pipe symbol (|), is used to pass the output of one program to the input of another. This eliminates the need to create temporary files to pass information to other programs. A pipe can also be used as a temporary storage place for a single program. A program can write to the pipe, then read that information back at a later time.

The standard library provides several pipe functions. The **popen** and **pclose** functions control both a pipe and a process. The **popen** function opens a pipe and creates a new process at the same time, making the new pipe the standard input or output of the new process. The **pclose** function closes the pipe and waits for termination of the corresponding process. The **pipe** function, on the other hand, gives low-level access to a pipe. The function is similar to the **open** function, but opens the pipe for both reading and writing, returning two file descriptors instead of one. The program can either use both sides of the pipe or close the one it does not need. The low-level input and output functions **read** and **write**, can be used to read from and write to a pipe. Pipe file descriptors are used in the same way as other file descriptors.

11.2 Opening a Pipe to a New Process

The **popen** function creates a new process and then opens a pipe to the standard input or output file of that new process. The function call has the form:

popen (*command*, *type*)

where *command* is a pointer to a string that contains a shell command line, and *type* is a pointer to the string which defines whether the pipe is to be opened for reading or writing by the original process. It may be *r* for reading or *w* for writing. The function normally returns the file pointer to the open pipe, but will return the null pointer value **NULL**, if an error is encountered.

C Library Guide

The function is typically used in programs that need to call another program and pass substantial amounts of data to that program. For example, in the following program fragment, **popen** creates a new process for the **cat** command and opens a pipe for writing:

```
FILE *pstrm;

pstrm = popen("cat >response", "w");
```

The new pipe given by *pstrm* links the standard input of the command with the program. Data written to the pipe will be used as input by the **cat** command.

11.3 Reading and Writing to a Process

The **fscanf**, **fprintf**, and other stream functions may be used to read from or write to a pipe opened by the **popen** function. These functions have the same form as described in “Run-Time Routines by Category.”

The **fscanf** function can be used to read from a pipe opened for reading. For example, in the following program fragment, **fscanf** reads from the pipe given by *pstrm*.

```
FILE *pstrm;
char name[20];
int number;

pstrm = popen("cat", "r");
fscanf(pstrm, "%s %d", name, &number);
```

This pipe is connected to the standard output of the **cat** command, so **fscanf** reads the first name and number written by **cat** to its standard output.

The **fprintf** function can be used to read from a pipe opened for writing. For example, in the following program fragment, **fprintf** writes the string pointed to by *buf* to the pipe given by *pstrm*:

```
FILE *pstrm;
char buf[MAX];

pstrm = popen("wc", "w");
fprintf(pstrm, "%s", buf)
```

This pipe is connected to the standard input of the **wc** command, so the command reads and counts the contents of *buf*.

11.4 Closing a Pipe

The **pclose** function closes the pipe opened by the **popen** function. The function call has the form:

pclose (*stream*)

where *stream* is the file pointer of the pipe to be closed. The function normally returns the exit status of the command that was issued as the first argument of its corresponding **popen**, but will return the value -1, if the pipe was not opened by **popen**.

For example, in the following program fragment, **pclose** closes the pipe given by *pstrm* if the end-of-file value, EOF, has been found in the pipe:

```
FILE *pstrm;
  ⋮
if (feof(pstrm) )
    pclose (pstrm);
```

11.5 Opening a Low-Level Pipe

The **pipe** function opens a pipe for both reading and writing. The function call has the form:

pipe (*fd*)

where *fd* is a pointer to a two-element array. It must have **int** type. Each element receives one file descriptor. The first element receives the file descriptor for the reading side of the pipe, and the other element receives the file descriptor for the writing side. The function normally returns 0, but will return the value -1, if an error is encountered. For example, in the following program fragment, **pipe** creates two file descriptors if no error is encountered:

```
int chan[2];

if (pipe(chan) == -1)
    exit(2);
```

The array element *chan[0]* receives the file descriptor for the reading side of the pipe, and *chan[1]* receives it for the writing side.

C Library Guide

The function is typically used to open a pipe in preparation for linking it to a child process. For example, in the following program fragment, **pipe** causes the program to create a child process if it successfully creates a pipe:

```
int  fd[2];

if ( pipe(fd) != -1 )
    if ( fork() == 0 )
        close(fd[1]);
```

Note that the child process closes the writing side of the pipe. The parent can now pass data to the child by writing to the pipe and the child can retrieve the data by reading the pipe.

11.5.1 Reading and Writing to a Low-Level Pipe

The **read** and **write** input and output functions can be used to read and write characters to a low-level pipe. These functions have the same form and operation described in “Run-Time Routines by Category.”

The **read** function can be used to read from the read side of an open pipe. For example, in the following program fragment, **read** reads MAX characters from the read side of the pipe given by *chan*:

```
int  chan[2];
char buf[MAX];
int  number;

number = read(chan[0], buf, MAX);
```

In this example, **read** stores the characters in the array *buf*.

Note that unless the end-of-file character is encountered, a **read** call waits for the given number of characters to be read before returning.

The **write** function can be used to write to the write side of a pipe. For example, in the following program fragment, **write** writes MAX characters from the character array *buf* to the writing side of the pipe given by *chan*:

```
int chan[2];
char buf[MAX];
int number;

pipe(chan);
number = write(chan[1], input, 512);
```

If the **write** function finds that a pipe is too full, it waits until some characters have been read before completing its operation.

11.5.2 Closing a Low-Level Pipe

The **close** function can be used to close the reading or the writing side of a pipe. The function has the same form and operation as described in “Run-Time Routines by Category.” For example, the function call:

```
close(chan[0])
```

closes the reading side of the pipe given by *chan*, and the call:

```
close(chan[1])
```

closes the writing side.

The system copies the end-of-file value, EOF, to a pipe when the process that made the original pipe and every process created or called by that process has closed the writing side of the pipe. This means, for example, that if a parent process is sending data to a child process through a pipe and closes the pipe to signal the end of the file, the child process will not receive the end-of-file value unless it has already closed its own write side of the pipe.

11.6 Program Examples

This section shows how to use the process control functions with the low-level **pipe** function to create functions similar to the **popen** and **pclose** functions.

C Library Guide

The first example is a modified version of the **popen** function. The modified function identifies the new pipe with a file descriptor rather than a file pointer. It also requires a “mode” argument rather than a “type” argument, where the mode is 0 for reading or 1 for writing:

```
#include <stdio.h>

#define READ 0
#define WRITE 1
#define tst(a, b) (mode == READ ? (b) : (a))
static int popen_pid;

popen(cmd, mode)
char *cmd;
int mode;
{
    int p[2];

    if (pipe(p) < 0)
        return(NULL);

    if ((popen_pid = fork()) == 0) {
        close(tst(p[WRITE], p[READ]));
        close(tst(0, 1));
        dup(tst(p[READ], p[WRITE]));
        close(tst(p[READ], p[WRITE]));
        execl("/bin/sh", "sh", "-c", cmd, 0);
        exit(1); /* sh cannot be found */
    }
    if (popen_pid == -1)
        return(NULL);

    close(tst(p[READ], p[WRITE]));
    return(tst(p[WRITE], p[READ]));
}
```

The function creates a pipe with the **pipe** function first. It then uses the **fork** function to create two copies of the original process. Each process has its own copy of the pipe. The child process decides whether it is supposed to read or write through the pipe, then closes the other side of the pipe and uses **execl** to create the new process and execute the desired program. The parent, on the other hand, closes the side of the pipe it does not use.

The sequence of **close** functions in the child process is a trick used to link the standard input or output of the child process to the pipe. The first **close** determines which side of the pipe should be closed and closes it. If “mode” is WRITE, the writing side is closed; if READ, the reading side is closed. The second **close** closes the standard input or output depending on the mode. If the mode is WRITE, the input is closed; if READ, the output is closed. The **dup** function creates a duplicate of the side of the pipe

that is still open. Since the standard input or output was closed immediately before this call, this duplicate receives the same file descriptor as the standard file. The system always chooses the lowest available file descriptor for a newly opened file. Since the duplicate pipe has the same file descriptor as the standard file, it becomes the standard input or output file for the process. Finally, the last **close** closes the original pipe, leaving only the duplicate.

The following example is a modified version of the **pclose** function. The modified version requires a file descriptor as an argument rather than a file pointer.

```
#include <signal.h>

pclose(fd)    /* close pipe fd */
int fd;
{
    int r, status;
    int (*hstat)(), (*istat)(), (*qstat)();
    extern int popen_pid;

    close(fd);

    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    hstat = signal(SIGHUP, SIG_IGN);

    while ((r = wait(&status)) != popen_pid && r != -1)
        ;
    if (r == -1)
        status = -1;

    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    signal(SIGHUP, hstat);

    return(status);
}
```

The function closes the pipe first. It then uses a **while** statement to wait for the child process given by *popen_pid*. If other child processes terminate while it waits, it ignores them and continues to wait for the given process. It stops waiting as soon as the given process terminates or if no child process exists. The function returns the termination status of the child, or the value -1, if there was an error.

The **signal** function calls used in this example ensure that no interrupts interfere with the waiting process. The first set of functions causes the process to ignore the interrupt, quit, and hang up signals. The last set restores the signals to their original status. The **signal** function is

C Library Guide

described in detail in the “Using Signals” chapter of the *XENIX Programmer’s Guide*.

Note that both example functions use the external variable `popen_pid` to store the process ID of the child process. If more than one pipe is to be opened, the `popen_pid` value must be saved in another variable before each call to `popen`, and this value must be restored before calling `pclose` to close the pipe. The functions can be modified to support more than one pipe by changing the `popen_pid` variable to an array indexed by a file descriptor.

11.7 Named Pipes

Named pipes are supported under XENIX System V. A named pipe is identical to a normal pipe, except that it has a name in the filesystem and it, therefore, stays around even when not being used. Named pipes are created by `mknod(S)`, not `pipe(S)`.

Typically, named pipes are used as “dump locations.” A daemon or server program creates and reads from a named pipe, while programs associated with the daemon or server program `open(S)` the pipe by name and write to it.

A named pipe is used like a normal file (`open(S)`, `read(S)`, and `write(S)`). Data is read and removed from the pipe in a FIFO (“First in First out”) manner. Data is written to the pipe in an atomic manner; that is, all data is written in a single `write`. It is not intermixed with other process’s written data. The `writes` appear consecutively in the pipe when they are read. Thus, one process can open the pipe for writing, and another process can open the pipe for reading.

The following routine creates a named pipe:

```
#include <sys/stat.h>
extern int errno;

/* make a named pipe, mode 666 */
if (mknod("/u/eric/pipe", S_IFIFO | 0666, 0) == -1) {
    perror("/u/eric/pipe"); /* An error occurred. */
    exit (errno);
}
```

Use **unlink(S)** to remove a named pipe.

```
if (unlink("/u/eric/pipe") == -1) {  
    perror("/u/eric/pipe");  
    exit(errno);  
}
```



(

(

Chapter 12

Using System Resources

- 12.1 Introduction 12-1
- 12.2 Allocating Memory 12-1
 - 12.2.1 Allocating Space for a Variable 12-2
 - 12.2.2 Allocating Memory for an Array 12-3
 - 12.2.3 Reallocating Memory 12-4
 - 12.2.4 Freeing Unused Memory 12-4
 - 12.2.5 Tuning Memory Allocation 12-5
 - 12.2.6 Optimizing Memory Allocation 12-6
 - 12.2.7 Gathering Memory Allocation Information 12-7
 - 12.2.8 Accessing Additional Memory Segments 12-7
- 12.3 Overview of File Locking 12-8
- 12.4 Locking Files Under XENIX 12-9
 - 12.4.1 Preparing a File for Locking 12-10
 - 12.4.2 Locking a File 12-10
 - 12.4.3 Program Example Using Locking 12-11
- 12.5 Locking Files Under UNIX System V 12-11
 - 12.5.1 Terminology 12-12
 - 12.5.2 File Protection 12-13
 - 12.5.3 Opening a File for Record Locking 12-14
 - 12.5.4 Setting a File Lock 12-14
 - 12.5.5 Setting and Removing Record Locks 12-17
 - 12.5.6 Getting Lock Information 12-20
 - 12.5.7 Handling Deadlocks 12-23
- 12.6 Message Operations 12-23
 - 12.6.1 Getting Message Queues 12-27
 - 12.6.2 Example Program Using msgget 12-31
 - 12.6.3 Controlling Message Queues 12-34
 - 12.6.4 Example Program Using msgctl 12-35
 - 12.6.5 Operations for Messages 12-40
 - 12.6.6 Sending Messages 12-40
 - 12.6.7 Receiving Messages 12-41

- 12.6.8 Example Program Using msgop 12-42
- 12.7 Overview of Semaphores 12-50
- 12.8 Using Semaphores Under XENIX 12-50
 - 12.8.1 Creating a Semaphore 12-51
 - 12.8.2 Opening a Semaphore 12-52
 - 12.8.3 Requesting Control of a Semaphore 12-53
 - 12.8.4 Checking the Status of a Semaphore 12-54
 - 12.8.5 Relinquishing Control of a Semaphore 12-54
 - 12.8.6 Program Example 12-55
- 12.9 Using Semaphores Under UNIX System V 12-57
 - 12.9.1 Semaphore Data Structures and Arrays 12-59
- 12.10 Getting Semaphores 12-63
 - 12.10.1 Example Program Using semget 12-67
 - 12.10.2 Controlling Semaphores 12-70
 - 12.10.3 Example Program Using semctl 12-71
 - 12.10.4 Example Program Using semop 12-82
- 12.11 Overview of Shared Memory 12-86
- 12.12 Using Shared Memory 12-87
 - 12.12.1 Creating a Shared Memory Segment 12-88
 - 12.12.2 Attaching a Shared Memory Segment 12-89
 - 12.12.3 Entering a Shared Memory Segment 12-90
 - 12.12.4 Leaving a Shared Memory Segment 12-91
 - 12.12.5 Getting the Current Version Number 12-92
 - 12.12.6 Waiting for a Version Number 12-93
 - 12.12.7 Freeing a Shared Memory Segment 12-94
 - 12.12.8 Program Example 12-94
- 12.13 Using Shared Memory Under UNIX System V 12-96
- 12.14 Shared Memory Data Structures 12-97
 - 12.14.1 Getting Shared Memory Segments 12-100
 - 12.14.2 Example Program Using shmget 12-103
 - 12.14.3 Controlling Shared Memory 12-106
 - 12.14.4 Example Program Using shmctl 12-107
 - 12.14.5 Operations for Shared Memory 12-113
 - 12.14.6 Example Program Using shmop 12-114

12.1 Introduction

This chapter describes the XENIX C library functions that let programs share the resources of the XENIX system. The functions give a program the means to queue for the use and control of a given resource and to synchronize its use with use by other programs.

This chapter explains how to:

- Allocate memory for dynamic storage
- Lock a file to ensure exclusive use by a program
- Use semaphores to control access to a resource
- Use shared data to allow interaction between programs
- Use message queues to communicate between processes

XENIX System V supports two sets of each of these operations (except message queues): one set for XENIX and one for UNIX System V. The two memory allocation packages described are available in both systems. The file locking, semaphores, and shared data packages described are valid only for one system or the other. It is not possible to use these operations on both XENIX and UNIX in a compatible fashion.

These sets of operations will be referred to in this chapter as either *XENIX operations* or *UNIX System V operations*. The UNIX System V operations are compatible with AT&T UNIX System V and should be used when software is intended for use on other System V operating systems that comply with the System V Interface Definition.

The XENIX operations are compatible with previous versions of XENIX and should be used only if you are working with software which uses XENIX style operations. Programs using the XENIX operations must be linked with the XENIX library, using the `-lx` option.

12.2 Allocating Memory

Some programs require significant changes to the size of their allocated memory space during different phases of their execution. The memory allocation functions of the standard C library let programs allocate space dynamically. This means a program can request a given number of bytes of storage for its exclusive use at the moment it needs the memory, then free this memory after it has finished using it.



C Library Guide

There are four memory allocation functions; they are described as follows:

Function	Description
malloc	Allocates memory for the first time. These functions
calloc	allocate a given number of bytes and return a pointer to the new memory.
realloc	Reallocates an existing memory, letting it to be used in a different way.
free	Returns allocated memory to the system.

Note that there are two versions of the **malloc** function available in XENIX System V: the standard version, and one that allocates memory more quickly. For more information, see **malloc(S)** in the *XENIX Programmer's Reference*.

12.2.1 Allocating Space for a Variable

The **malloc** function allocates space for a variable containing a given number of bytes. The function call has the following form:

```
malloc (size)
```

where *size* is an unsigned integer that gives the number of bytes to be allocated. For example, the following function call allocates four bytes of storage:

```
table = malloc (4)
```

The function normally returns a pointer to the starting address of the allocated space, but will return a null pointer value if there is not enough space to allocate.

In the following program fragment, **malloc** is used to allocate space for ten different strings, each of different length. In addition, **malloc** returns the beginning address of each string to the array of pointers called *string*. In the following example, the strings are read from the standard input. Note that the **strlen** function is used to get the size in bytes of each string.

```

int i;
char *temp, *strings[10];
unsigned isize;

for ( i=0; i<10; i++) {
    scanf("%s", temp);
    isize = strlen(temp);
    string[i] = malloc(isize);
}

```



12.2.2 Allocating Memory for an Array

The `calloc` function allocates storage for a given array and initializes each element in the new array to zero. The function call has the following form:

`calloc` (*n*, *size*)

where

- *n* is the number of elements in the array.
- *size* is the number of bytes in each element.

The function normally returns a pointer to the starting address of the allocated memory, but will return a null pointer value if there is not enough memory. For example, the following function call allocates sufficient memory for a 10-element array:

```
table = calloc (10,4)
```

Each element has four bytes. You use this function in programs that must process large arrays without knowing the size of an array in advance. For example, in the following program fragment, `calloc` is used to allocate storage for an array of values read from the standard input:

```

int i;
char *table;
unsigned inum;

scanf("%d", &inum);
table = calloc (inum, 4);
for (i=0; i<inum; i++)
    scanf("%d", table[i]);

```

Note that the number of elements is read from the standard input before the elements are read.

C Library Guide

12.2.3 Reallocating Memory

The **realloc** function reallocates the memory at a given address without changing the contents of the memory. The function call has the following form:

```
realloc (ptr, size)
```

where

- *ptr* is a pointer to the starting address of the memory to be reallocated.
- *size* is an unsigned number giving the new size in bytes of the reallocated memory.

The function normally returns a pointer to the starting address of the allocated memory, but will return a null pointer value if there is not enough memory to allocate.

This function is typically used to keep storage as compact as possible. For example, the following program fragment uses **realloc** to remove table entries:

```
main ()
{
char *table;
int i;
unsigned inum;

for (i=inum; i>-1; i--) {
    printf("%d\n", strings[i]);
    strings = realloc(strings, i*4);
}
```

In this example, an entry is removed after it has been printed at the standard output, by reducing the size of the allocated memory from its current length to the length given by ‘i*4’.

12.2.4 Freeing Unused Memory

The **free** function frees unused memory that had been previously allocated by a **malloc**, **calloc**, or **realloc** function call. The function call has the following form:

```
free (ptr)
```

where *ptr* is the pointer to the starting address of the memory to be freed. This pointer must be the return value of a **malloc**, **calloc**, or **realloc** function.

You use this function to free memory that is no longer used or to free memory to be used for other purposes. For example, in the following program fragment **free** frees the allocated memory pointed to by strings if the first element is equal to zero:

```
main ()
{
  char *table;

  if ( table[0] == -1 )
    free (table);
```

12.2.5 Tuning Memory Allocation

The memory allocation package available in the standard C library, *libc.a*, uses a linear search through all blocks to allocate space, starting at a roving start pointer. This algorithm is space-efficient and gives good performance as long as the total number of blocks allocated is small. However, with a large number of blocks, this algorithm has serious performance problems.

XENIX System V includes a new memory allocation package that uses a different time/space algorithm. The new library functions allocate space quickly, but use space inefficiently when the number of blocks allocated is small. Programs that were close to running out of memory using the standard package cannot be easily changed to the new package.

This new package improves performance for programs that make heavy use of dynamic-memory allocation. This algorithm provides several tunable parameters so you can customize the algorithm. Instrumentation is provided to help you in the choice of values for these parameters.

The new library package contains the same basic functions with the same functionality: **malloc**, **free**, **realloc**, and **calloc**. It also contains the new functions **mallopt** and **mallinfo**.

The new package is provided in *libmalloc.a*, and can be linked by setting the **-lmalloc** flag on the **cc** command line. The standard package is kept in the standard C library, *libc.a*. The *libc.a* library is the default library.



C Library Guide

Note

Unless you specify the new package with **-malloc**, the standard package is automatically linked with your program. If you leave programs alone, they will work exactly the way they used to work.

The interface to both packages is described in **malloc(S)** in the *XENIX Programmer's Reference*. The new interface is similar to the standard interface with functions having the same names. This lets you use the new package without changing code, but also protects the naive user from the interface change that would occur if the new package replaced the standard package. In addition, it insures that all modules will use the same allocation routines that an application program uses, avoiding fragmentation.

The major difference between the standard and new packages is that, with the standard **malloc**, after a block is freed, but before another is allocated, the data in the freed block is valid. By default, the new package does not have this property. A compiler option lets you use this property, at the cost of two extra words of overhead per block. For more information on the memory allocation option, see the *C User's Guide*.

The new **malloc** has the following additional functions:

Function	Description
<code>malloc</code>	Provides control over the allocation algorithm.
<code>mallinfo</code>	Provides instrumentation describing memory usage.

This information can be used to determine optimal **malloc** operation using **malloc**.

12.2.6 Optimizing Memory Allocation

The **malloc** function provides control over the allocation algorithm. The function call has the following form:

```
malloc (cmd, value);
```

where *cmd* sets the variables that let you tune the algorithm to allocate memory in the most efficient manner for the application.

The values available for *cmd* are:

Value	Description
M_MXFAST	Sets the maximum size of blocks that are very quickly allocated in large groups.
M_NBLKS	Sets the size of the large groups allocated when blocks less than the size of <i>maxfast</i> are encountered.
M_GRAIN	Sets the <i>grain</i> used when rounding values.
M_KEEP	Preserves the data in a freed block until the next allocation (for compatibility with the other malloc).

The **malloc** function can be called repeatedly, until the first block is allocated.

12.2.7 Gathering Memory Allocation Information

The **mallinfo** function provides instrumentation describing memory usage. It returns a structure that is defined in the `<malloc.h>` include file. The function call has the following form:

```
#include <malloc.h>

struct mallinfo mi;
mi=mallinfo();
```

For more information, see **malloc(S)** in the *XENIX Programmer's Reference*.

12.2.8 Accessing Additional Memory Segments

The **brkctl(S)** system call lets you access additional data segments. Use it only when **malloc(S)** is not sufficient. It is not available under UNIX System V or XENIX System V 1.0.

Small and medium model programs can use **brkctl** to access additional memory in a far data segment. Be sure to use the **-Me** option to **cc(CP)** when compiling programs using **brkctl** to enable the use of the **far** keyword. For most applications, the **-lbrkctl** option to **cc(CP)** should be used to cause a special **brkctl** library to be linked with the program. This

C Library Guide

library will simulate the use of an additional segment, through shared memory, if the **brkctl** fails.

The **brkctl** function uses several other functions that manipulate the size of a data segment. The most useful function on systems with segmented architecture is the **BR_NEWSEG** command, which acts similar to a **malloc** function as shown in the following program example.

Example

```
#include <sys/brk.h>

#define FNULL (int far*)0
#define FAILURE (int far*)-1

main()
{
    int i, j;
    int far*fp, far*brkctl(); /*both far* are necessary*/

    fp=brkctl(BR_NEWSEG, 40000L, FNULL);
    if(fp== FAILURE){
        perror("brkctl failed");
        exit(1);
    }
    for(i=0; i<20000; ++i)
        fp[i]=i+1;
    for(i=0; i<20000; ++i)
        printf("%d\n", fp[i]);
}
```

Note that this example allocates 40,000 bytes in a far data segment and fills this memory with the integers from 1 to 20,000. Since *fp* is a far pointer, it cannot be passed to the standard small or medium library functions (e.g. **strcpy**) because they expect near pointers.

12.3 Overview of File Locking

Locking a file is a way to synchronize file use when several processes may require access to a single file. The standard C library provides three file locking functions: **locking**, **lockf**, and **fcntl**. These functions lock any given section of a file, preventing all other processes that wish to use the section from gaining access. A process can lock the entire file or only a small portion. In any case, only the locked section is protected; all other sections can be accessed by other processes as usual. For more information, see **fcntl(S)**, **lockf(S)**, and **locking(S)** in the *XENIX Programmer's Reference*.

This section provides a brief overview of the differences between file locking techniques under UNIX System V and XENIX. The UNIX System V file locking functions are **lockf(S)** and **fcntl(S)**. The XENIX file locking function is **locking(S)**.

Function	Description
fcntl	Controls regions of open files allowing reads or writes of locked areas.
lockf	Uses semaphores to lock a file specified by the file descriptor <i>fdes</i> .
locking	Locks or unlocks regions a file for reading and writing.

The syntax and arguments for **lockf** and **locking** are essentially the same. The only difference is the preprocessor defines for the commands (for example, **F_LOCK** vs. **LK_LOCK**). The **fcntl** function is called in a manner similar to **ioctl(S)**. There is no need to do a **seek(S)** before locking with **fcntl**, since it can do its own seek.

All three locking functions are enforced by semaphore control. When another process tries to access an area that is locked, that process suspends execution or gets an error return. Note that it is possible to lock a region beyond the end of a file in all three functions.

12.4 Locking Files Under XENIX

File locking protects a file from the damage that may be caused if several processes try to read or write to the file at the same time. It also provides unhindered access to any portion of a file for a controlling process. File locking is performed under XENIX using the **locking(S)** function.

Before you can lock a file, however, you must prepare it by using the **open** and **lseek** functions.

To use the **locking** function, you must add the following include file to the beginning of the program:

```
#include <sys/locking.h>
```

The *sys/locking.h* file contains definitions for the modes used with the function.

C Library Guide

12.4.1 Preparing a File for Locking

Before a file can be locked, it must first be opened using the **open** function, then properly positioned using the **lseek** function, which in turn moves the file's character pointer to the first byte to be locked.

The **open** function is used once at the beginning of the program to open the file. The **lseek** function can be used any number of times to move the character pointer to each new section to be locked. For example, the following statements prepare the first 100 bytes at file position 1024 from the beginning of the *reservations* file for locking:

```
fd = open("reservations", O_RDONLY);
lseek(fd, 1024, 0);
```

12.4.2 Locking a File

The **locking** function locks one or more bytes of a given file. The function call has the following form:

locking (*filedes*, *mode*, *size*)

where

- *filedes* is the file descriptor of the file to be locked.
- *mode* is an integer value that defines the type of lock to be applied to the file.
- *size* is a long integer value giving the size in bytes of the portion of the file section to be locked or unlocked.

The *mode* can be **LK_LOCK** for locking the given bytes, or **LK_UNLCK** for unlocking them. For example, in the following program fragment, **locking** locks 100 bytes at the current character pointer position in the file given by “*fd*”:

```
#include <sys/locking.h>

main () {
    int fd;

    fd = open("data", O_RDWR);
    locking(fd, LK_LOCK, 100L);
```

The function normally returns the number of bytes locked, but will return -1 if it encounters an error.

12.4.3 Program Example Using Locking

This section shows you how to lock and unlock a small section in a file using the **locking** function. In the following program, the function locks 100 bytes in a *data* file that is opened for reading and writing. The function accesses the locked portion of the file. Then **locking** is used again to unlock the file.

12

Example

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/locking.h>

main() {
    int fd, err;
    char *data;

    fd = open("data", O_RDWR);    /* Open data for R/W */
    if (fd == -1) {
        perror("");
    }
    else {
        lseek(fd, 100L, 0);    /* Seek to pos 100 */
        err = locking(fd, LK_LOCK, 100L);    /* Lock bytes 100-200 */
        if (err == -1) {
            /* process error return */
        }

        /* read or write bytes 100 - 200 in the file */

        lseek(fd, 100L, 0);    /* Seek to pos 100 */
        locking(fd, LK_UNLCK, 100L);    /* Unlock bytes 100-200 */
    }
}
```

12.5 Locking Files Under UNIX System V

Mandatory file, advisory file, and record locking can synchronize program that access the same stores of data simultaneously.

Advisory file and record locking can coordinate self-synchronizing processes. In mandatory locking, the standard I/O subfunctions and I/O

C Library Guide

system calls enforce the locking protocol. In this way, at the cost of a little efficiency, mandatory locking double checks the programs against accessing the data out of sequence.

The remainder of this section describes how file and record locking capabilities can be used.

12.5.1 Terminology

The following terms are used in locking:

Term	Description
Record	A contiguous set of bytes in a file. The system does not impose any record structure on files. This can be done by the programs that use the files.

Cooperating Processes

Processes that work together in some well defined fashion to accomplish the tasks at hand. Processes that share files must request permission to access the files before using them. File access permissions must be carefully set to restrict non-cooperating processes from accessing those files. The system uses the term *process* interchangeably with the cooperating process to refer to a task obeying such protocols.

Read (Share) Locks

Used to gain limited access to sections of files. When a read lock is in place on a record, other processes can also read lock that record, in whole or in part. No other process, however, can have or obtain a write lock on an overlapping section of the file. If a process holds a read lock it can assume that no other process will be writing or updating that record at the same time. This access method also permits many processes to read the given record. This might be necessary when searching a file, without the contention involved if a write or exclusive lock were to be used.

Write (Exclusive) Locks

Used to gain complete control over sections of files. When a write lock is in place on a record, no other process can read or write lock that record, in whole or in part. If a process holds a write lock it can assume that no other process will be reading or writing that record at the same time.

Mandatory Locking

A form of record locking that does interact with the I/O subsystem. Access to locked records is enforced by the system calls **chsize**, **creat**, **open**, **read**, and **write**.

If a record is locked, then the system restricts access of that record by any other process according to the type of lock on the record. The control over records should still be performed by requesting an appropriate record lock before I/O operations, but an additional check is made by the system before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers an extra synchronization check, but at the cost of some additional system overhead.

12.5.2 File Protection

Access permissions for system files control who can read, write, or execute a file. These permissions can only be set by the owner of the file or by the super-user. Directory permissions will also affect how a file can be used and by whom. Note that if the directory permissions allow anyone to write in it, then files within the directory can be removed, even if those files do not have read, write or execute permission for that user. If your application warrants the use of record locking, make sure that the permissions on your files and directories are set properly. A record lock, even a mandatory record lock, will only protect the portions of the files that are locked. Other parts of these files can still be corrupted.

Only a few programs and/or system administrators should be able to read or write a data base. This can be done easily by setting the set-group-ID bit (using **chmod(C)**) of the data base accessing programs. The files can then be accessed by a few programs that obey the record locking protocol. An example of such file protection, although record locking is not used, is the **mail(C)** command. In that command only the **mail** command and the particular user can read and write in the unread mail files.

C Library Guide

12.5.3 Opening a File for Record Locking

Before you can lock a file or segment of a file you must have a valid open file descriptor. If read locks are to be done, then the file must be opened with at least read accessibility and likewise for write locks and write accessibility. The following example opens the file for both read and write access:

Example

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>

int fd;          /* file descriptor */
char *filename;

main(argc, argv)
int argc;
char *argv[];
{
    extern void exit(), perror();

    /* get data base file name from command line and open the
     * file for read and write access.
     */
    if (argc < 2) {
        (void) fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(2);
    }
    filename = argv[1];
    fd = open(filename, O_RDWR);
    if (fd < 0) {
        perror(filename);
        exit(2);
    }
    .
    .
    .
}
```

The file is now open for you to perform both locking and I/O functions. You can then set a lock.

12.5.4 Setting a File Lock

There are several ways you can set a lock on a file. How you set a lock on a file will depend on how you want the lock to interact with the rest of the program, how you want the lock to perform, and your portability needs.

Two methods are given here, one using the **fcntl(S)** system call, the other using the */usr/group* standards compatible **lockf(S)** library function call.

Locking an entire file is just a special case of record locking. For both **fcntl** and **lockf** the concept and the effect of the lock are the same. The file is locked starting at a byte offset of zero (0) until the end of the maximum file size. This point extends beyond any real end of the file so that no lock can be placed on this file beyond this point. To do this, the value of the size of the lock is set to zero. The following sample code uses the **fcntl(S)** system call.

Example

```
#include <fcntl.h>
#define MAX_TRY 10
int try;
struct flock lck;

try = 0;

/* set up the record locking structure, the address of which
 * is passed to the fcntl system call.
 */
lck.l_type = F_WRLCK; /* setting a write lock */
lck.l_whence = 0; /* offset l_start from beginning of file */
lck.l_start = 0L;
lck.l_len = 0L; /* until the end of the file address space */

/* Attempt locking MAX_TRY times before giving up. */
while (fcntl(fd, F_SETLK, &lck) < 0) {
    if (errno == EAGAIN || errno == EACCES) {
        /* there might be other error cases in which
         * you might try again.
         */
        if (++try < MAX_TRY) {
            (void) sleep(2);
            continue;
        }
        (void) fprintf(stderr, "File busy try again later!\n");
        return;
    }
    perror("fcntl");
    exit(2);
}

.
.
.
```

C Library Guide

This portion of code tries to lock a file. This is attempted several times until one of the following things happens:

- the file is locked
- an error occurs
- it gives up trying because `MAX_TRY` has been exceeded

The following sample of code uses the `lockf(S)` function.

Example

```
#include <unistd.h>
#define MAX_TRY 10
int try;
try = 0;

/* make sure the file pointer
 * is at the beginning of the file.
 */
lseek(fd, 0L, 0);

/* Attempt locking MAX_TRY times before giving up.
 */
while (lockf(fd, F_TLOCK, 0L) < 0) {
    if (errno == EAGAIN || errno == EACCES) {
        /* there might be other errors cases in which
         * you might try again.
         */
        if (++try < MAX_TRY) {
            sleep(2);
            continue;
        }
        (void) fprintf(stderr, "File busy try again later!\n");
        return;
    }
    perror("lockf");
    exit(2);
}

.
.
.
```

Note that the `lockf(S)` sample appears to be more simple, but the `fcntl(S)` sample is more flexible. Using the `fcntl(S)` method, it is possible to set the type and start of the lock request simply by setting a few structure variables. The `lockf(S)` library system call merely sets write (exclusive) locks; an additional system call (`lseek(S)`) is required to specify the start of the lock.

12.5.5 Setting and Removing Record Locks

Locking a record is done the same way as locking a file except for the differing starting point and length of the lock. In the following sample session there are two records (these records may be in the same or different file) that must be updated simultaneously so that other processes get a consistent view of this information. (This type of problem comes up, for example, when updating the inter-record pointers in a doubly-linked list.) To do this you must decide the following questions:

12

- What do you want to lock?
- For multiple locks, what order do you want to lock and unlock the records?
- What do you do if you succeed in getting all the required locks?
- What do you do if you fail to get all the locks?

In managing record locks, you must plan a failure strategy in case you cannot obtain all the required locks. Different programs might:

- wait a certain amount of time, and try again
- abort the procedure and warn the user
- let the process sleep until signaled that the lock has been freed
- some combination of these

In the sample session you will try to insert an entry into a doubly-linked list. For the example, assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be changed or promoted to a write lock so that the record can be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. If there are processes with pending write locks that are sleeping on the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Promoting (or demoting) a write lock to a read lock carries no restrictions. In either case, the lock is merely reset with the new lock type. Because the `/usr/group lockf` function does not have read locks, lock promotion is not applicable to that call. The following is an example of record locking with lock promotion.

C Library Guide

```
struct record {
    .
    .           /* data portion of record */
    .
    long prev; /* index to previous record in the list */
    long next; /* index to next record in the list */
};
/* Lock promotion using fcntl(S)
 * When this function is entered it is assumed that there are read
 * locks on "here" and "next".
 * If write locks on "here" and "next" are obtained:
 *   Set a write lock on "this".
 *   Return index to "this" record.
 * If any write lock is not obtained:
 *   Restore read locks on "here" and "next".
 *   Remove all other locks.
 *   Return a -1.
 */
long
set3lock (this, here, next)
long this, here, next;
{
    struct flock lck;

    lck.l_type = F_WRLCK; /* setting a write lock */
    lck.l_whence = 0; /* offset l_start from beginning of file */
    lck.l_start = here;
    lck.l_len = sizeof(struct record);

    /* promote lock on "here" to write lock */
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        return (-1);
    }
    /* lock "this" with write lock */
    lck.l_start = this;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Lock on "this" failed;
         * demote lock on "here" to read lock. */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLKW, &lck);
        return (-1);
    }
    /* promote lock on "next" to write lock */
    lck.l_start = next;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Lock on "next" failed;
         * demote lock on "here" to read lock, */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLK, &lck);
        /* and remove lock on "this". */
        lck.l_type = F_UNLCK;
        lck.l_start = this;
        (void) fcntl(fd, F_SETLK, &lck);
        return (-1); /* cannot set lock, try again or quit */
    }
    return (this);
}
```

The locks on these three records were all set to wait (sleep) if another process was blocking them from being set. This was done with the `F_SETLKW` command. If the `F_SETLK` command was used instead, the `fcntl` system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections.

The following is another example using the `lockf` function. Since there are no read locks, all (write) locks are referenced generically as locks.



Example

```

/* Lock promotion using lockf(S)
 * When this function is entered it is assumed that there are
 * no locks on "here" and "next".
 * If locks are obtained:
 *   Set a lock on "this".
 *   Return index to "this" record.
 * If any lock is not obtained:
 *   Remove all other locks.
 *   Return a -1.
 */

#include <unistd.h>

long
set3lock (this, here, next)
long this, here, next;

{
    /* lock "here" */
    (void) lseek (fd, here, 0);
    if (lockf (fd, F_LOCK, sizeof (struct record)) < 0) {
        return (-1);
    }
    /* lock "this" */
    (void) lseek (fd, this, 0);
    if (lockf (fd, F_LOCK, sizeof (struct record)) < 0) {

        /* Lock on "this" failed.
         * Clear lock on "here".
         */
        (void) lseek (fd, here, 0);
        (void) lockf (fd, F_ULOCK, sizeof (struct record));
        return (-1);
    }
}

```


C Library Guide

Example (cont.)

```
/* lock "next" */
(void) lseek(fd, next, 0);
if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {

    /* Lock on "next" failed.
     * Clear lock on "here",
     */
    (void) lseek(fd, here, 0);
    (void) lockf(fd, F_ULOCK, sizeof(struct record));

    /* and remove lock on "this".
     */
    (void) lseek(fd, this, 0);
    (void) lockf(fd, F_ULOCK, sizeof(struct record));
    return (-1); /* cannot set lock, try again or quit */
}
return (this);
}
```

Locks are removed in the same manner as they are set, only the lock type is different (`F_UNLCK` or `F_ULOCK`). An unlock cannot be blocked by another process and will only affect locks that were placed by this process. The unlock only affects the “lck” section of the file defined in the first example. It is possible to unlock or change the type of lock on a subsection of a previously set lock. This may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

12.5.6 Getting Lock Information

You can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test of the lock or as a means to find locks on a file. To do this, a lock is set up as in the previous examples and the `F_GETLK` command is used in the `fcntl` call. If the lock passed to `fcntl` would be blocked, the first blocking lock is returned to the process through the structure passed to `fcntl`. That is, the lock data passed to `fcntl` is overwritten by blocking lock information. This information includes `l_pid` and `l_sysid`, that are only used by `F_GETLK`. (For systems that do not support a distributed architecture the value in `l_sysid` should be ignored.) These fields uniquely identify the process holding the lock.

For example, if a lock passed to `fcntl` using the `F_GETLK` command would not be blocked by another process’ lock, then the `l_type` field is changed to `F_UNLCK` and the remaining fields in the structure are unaffected. You can use this capability to print all the file segments

locked by other processes. Note that if there are several read locks over the same segment only one of these will be found.

Example

```

struct flock lck;

/* Find and print "write lock" blocked segments of this file. */
(void) printf("sysid pid type start length\n");
lck.l_whence = 0;
lck.l_start = 0L;
lck.l_len = 0L;
do {
    lck.l_type = F_WRLCK;
    (void) fcntl(fd, F_GETLK, &lck);
    if (lck.l_type != F_UNLCK) {
        (void) printf("%5d %5d %c %8d %8d\n",
            lck.l_sysid,
            lck.l_pid,
            (lck.l_type == F_WRLCK) ? 'W' : 'R',
            lck.l_start,
            lck.l_len);
        /* if this lock goes to the end of the address
         * space, no need to look further, so break out.
         */
        if (lck.l_len == 0)
            break;
        /* otherwise, look for new lock after the one
         * just found.
         */
        lck.l_start += lck.l_len;
    }
} while (lck.l_type != F_UNLCK);

```

The `fcntl` call with the `F_GETLK` command will always return correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

The `lockf` function with the `F_TEST` command can also be used to test if there is a process blocking a lock. This function does not, however,

C Library Guide

return the information about where the lock actually is and which process owns the lock. A function using **lockf** to test for a lock on a file follows:

Example

```
/* find a blocked record. */

/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero (0)
 * to test until the end of the file address space.
 */
if (lockf(fd, F_TEST, 0L) < 0) {
    switch (errno) {
        case EACCES:
        case EAGAIN:
            (void) printf("file is locked by another process\n");
            break;
        case EBADF:
            /* bad argument passed to lockf */
            perror("lockf");
            break;
        default:
            (void) printf("lockf: unknown error <%d>\n", errno);
            break;
    }
}
```

When a process forks, the *child process* receives a copy of the file descriptors that the *parent process* has opened. The parent and child also share a common file pointer for each file. If the parent were to seek to a point in the file, the child's file pointer would also be at that location. This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, as described by **l_start**, when using a **l_whence** value of 1. If both the parent and child process set locks on the same file, there is a possibility that a lock will be set using a file pointer that was reset by the other process. This problem appears in the **lockf(S)** function call as well and is a result of the */usr/group* requirements for record locking.

If forking is used in a record locking program, the child process should close and reopen the file if either locking method is used. This will result in the creation of a new and separate file pointer that can be manipulated without this problem occurring. Another solution is to use the **fcntl** system call with a **l_whence** value of 0 or 2. This makes the locking function atomic, so that even processes sharing file pointers can be locked without difficulty.

12.5.7 Handling Deadlocks

There is a certain level of deadlock detection/avoidance built into the record locking facility. This deadlock handling provides the same level of protection granted by the */usr/group* standard **lockf** call. This deadlock detection is only valid for processes that are locking files or records on a single system. Deadlocks can only potentially occur when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call will fail and set **errno** to the deadlock error number. If a process wishes to avoid the use of the system's deadlock detection it should set its locks using **F_SETLK** instead of **F_SETLKW**.

12.6 Message Operations

This section describes the system calls for the message type of Inter-Process Communication (IPC). The message type of IPC lets processes (executing programs) communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions called messages. Processes using this type of IPC can perform two operations, sending and receiving.

Before a message can be sent or received by a process, a process must have the operating system generate the necessary software mechanisms by using the **msgget(S)** system call. While doing this, the process becomes the owner/creator of the message facility and specifies the initial operation permissions for all other processes, including itself. Subsequently, the owner/creator can relinquish ownership or change the operation permissions using the **msgctl(S)** system call. However, the creator remains the creator as long as the facility exists. Other processes with permission can use **msgctl** to perform various other control functions.

Processes that have permission and are attempting to send or receive a message can suspend execution if they are unsuccessful at performing their operation. That is, a process that is attempting to send a message can wait until the process that is to receive the message is ready, and vice versa. A process that specifies that execution is to be suspended is performing a *blocking message operation*. A process that does not let its execution be suspended is performing a *nonblocking message operation*.

C Library Guide

A process performing a blocking message operation can be suspended until one of three conditions occurs:

- It is successful.
- It receives a signal.
- The facility is removed.

System calls make these message capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns applicable information. Otherwise, a known error code (-1) is returned to the process, and an external error number variable `errno` is set accordingly.

Before a message can be sent or received, a uniquely identified message queue and data structure must be created using `msgget(S)`. The unique identifier created is called the `msqid` (message queue identifier); it is used to identify or reference the associated message queue and data structure.

The message queue is used to store (header) information about each message that is being sent or received. This information includes the following for each message:

- pointer to the next message on queue
- message type
- message text size
- message text address

There is one associated data structure for the uniquely identified message queue. This data structure contains the following information:

- operation permissions data (operation permission structure)
- pointer to first message on the queue
- pointer to last message on the queue
- current number of bytes on the queue
- number of messages on the queue
- maximum number of bytes on the queue
- process identification number (PID) of last message sender

- PID of last message receiver
- last message send time
- last message receive time
- last change time

Note

All include files discussed in this chapter are located in the */usr/include* or */usr/include/sys* directories.

In the C programming language, the data structure definition for the message queue is as follows:

```

struct msg
{
    struct msg    *msg_next; /* ptr to next message on q */
    long         msg_type; /* message type */
    short        msg_ts; /* message text size */
    short        msg_spot; /* message text map address */
};

```

The structure **msg** is located in the */usr/include/sys/msg.h* include file.

The structure definition for the associated message queue data structure is as follows:

```

struct msqid_ds
{
    struct ipc_perm msg_perm; /* operation permission struct */
    struct msg      *msg_first; /* ptr to first message on q */
    struct msg      *msg_last; /* ptr to last message on q */
    ushort         msg_cbytes; /* current # bytes on q */
    ushort         msg_qnum; /* # of messages on q */
    ushort         msg_qbytes; /* max # of bytes on q */
    ushort         msg_lspid; /* pid of last msgsnd */
    ushort         msg_lrpid; /* pid of last msgrcv */
    time_t         msg_stime; /* last msgsnd time */
    time_t         msg_rtime; /* last msgrcv time */
    time_t         msg_ctime; /* last change time */
};

```

C Library Guide

The structure **msqid_ds** is located in the `<sys/msg.h>` include file also. Note that the **msg_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown as follows:

```
struct ipc_perm
{
    ushort    uid; /* owner's user id */
    ushort    gid; /* owner's group id */
    ushort    cuid; /* creator's user id */
    ushort    cgid; /* creator's group id */
    ushort    mode; /* access modes */
    ushort    seq; /* slot usage sequence number */
    key_t    key; /* key */
};
```

The structure **ipc_perm** is located in the `<sys/ipc.h>` include file; it is common for all IPC facilities.

The **msgget(S)** system call performs two tasks when only the **IPC_CREAT** flag is set in the **msgflg** argument that it receives:

- getting a new **msqid** and creating an associated message queue and data structure for it
- returning an existing **msqid** that already has an associated message queue and data structure

The task performed is determined by the value of the **key** argument passed to the **msgget** system call. For this task, if the **key** is not already in use for an existing **msqid**, a new **msqid** is returned with an associated message queue and data structure created for the **key**. This occurs provided no system-tunable parameters would be exceeded.

There is also a provision for specifying a **key** of value zero which is known as the private **key** (**IPC_PRIVATE = 0**); when specified, a new **msqid** is always returned with an associated message queue and data structure created for it unless a system-tunable parameter would be exceeded.

For the second task, if a **msqid** exists for the **key** specified, the value of the existing **msqid** is returned. If you do not wish to have an existing **msqid** returned, specify (set) the **IPC_EXCL** flag in the **msgflg** argument passed to the system call. For more information, see “Example Program Using **msgget**.”

When performing this task, the process that calls **msgget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains with the creator. The creator of the message queue also determines the initial operation permissions for it. For more information, see “Controlling Message Queues.”

Once a uniquely identified message queue and data structure are created, you can use message operations **msgop(S)** and message control **msgctl(S)**.

Message operations consist of sending and receiving messages. System calls are provided for each of these operations; they are **msgsnd(S)** and **msgrcv(S)**. For more information, see “Operations for Messages.”

Message control is performed using the **msgctl** system call. It helps you control the message facility in the following ways:

- Determines the associated data structure status for **msqid**.
- Changes operation permissions for a message queue.
- Changes the **size (msg_qbytes)** of the message queue for a particular **msqid**.
- Removes a particular **msqid** from the operating system along with its associated message queue and data structure.

For more information on the **msgctl** system call, see “Controlling Message Queues.”

12.6.1 Getting Message Queues

This section describes the **msgget(S)** system call along with an example program illustrating its use. The **msgget** system call returns the message queue identifier associated with **key**. The following include files are located in the */usr/include/sys* directory of the XENIX operating system:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key, msgflg)
key_t key;
int msgflg;
```


C Library Guide

The following line informs you that **msgget** is a function with two formal arguments that return an integer type value upon successful completion (**msqid**):

```
int msgget (key, msgflg)
```

The following two lines declare the types of the formal arguments:

```
key_t key;  
int msgflg;
```

where **key_t** is declared by a **typedef** in the `<types.h>` include file to be an integer.

The integer returned from this function upon successful completion is the message queue identifier (**msqid**).

The process calling the **msgget** system call must supply two arguments to be passed to the formal **key** and **msgflg** arguments.

A new **msqid** with an associated message queue and data structure is provided if:

- **key** is equal to `IPC_PRIVATE`
- **key** is a unique number, and **msgflg** ANDed with `IPC_CREAT` is “true” (not zero)

The value passed to the **msgflg** argument must be an integer value and specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes and execution modes determine the user, group, and other attributes of the **msgflg** argument. These are collectively referred to as *operation permissions*. The following table reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

12

A specific octal value is found by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the `<msg.h>` include file that can be used for the user (OWNER).

Control commands are predefined constants (represented by all uppercase letters). The names of the constants that apply to the `msgget` system call along with their values follow. They are hereafter referred to as *flags* and are defined in the include file.

Flag	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for `msgflg` is therefore a combination of operation permissions and flags. After determining the value for the operation permissions as previously described, you can specify the desired flags. This is accomplished by bitwise OR-ing (|) them with the operation permissions; the bit positions and values for the flags in relation to those of the operation permissions make this possible. It is illustrated as follows:

Variable		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
OR-ed by user	=	0 0 4 0 0	0 000 000 100 000 000
msgflg	=	0 1 4 0 0	0 000 001 100 000 000

C Library Guide

The **msgflg** value can be set by using the names of the flags in conjunction with the octal operation permissions value:

```
msgid = msgget (key, (IPC_CREAT | 0400));  
msgid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by **msgget(S)**, success or failure of this system call depends upon the argument values for **key** and **msgflg** or system-tunable parameters. The system call will attempt to return a new **msgid** if one of the following conditions is true:

- Key is equal to **IPC_PRIVATE**
- Key does not already have a **msgid** associated with it, and (**msgflg** ANDed **IPC_CREAT**) is “true” (not zero).

The **key** argument can be set to **IPC_PRIVATE** in the following way:

```
msgid = msgget (IPC_PRIVATE, msgflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the **MSGMNI** system-tunable parameter always causes a failure. The **MSGMNI** system-tunable parameter determines the maximum number of unique message queues (**msgid**'s) in the system.

The second condition is satisfied if the value for **key** is not already associated with a **msgid** and the bitwise AND of **msgflg** and **IPC_CREAT** is “true” (not zero). This means that the **key** is unique (not in use) within the system for this facility type and that the **IPC_CREAT** flag is set (**msgflg** | **IPC_CREAT**). You can test if a flag is set by using the following bitwise AND (&):

```
msgflg:=:x 1 x x x:(x = don't care)  
& IPC_CREAT:=:0 1 0 0 0:  
  
result:=:0 1 0 0 0:(not zero)
```

Since the result is not zero, the flag is set or “true.”

IPC_EXCL is another flag that works in conjunction with **IPC_CREAT** to have the system call fail if, and only if, a **msgid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **msgid** when it has not. In other words, when both **IPC_CREAT** and **IPC_EXCL** are specified, a new **msgid** is returned if the system call is successful.

For more information on associated data structures, see **msgget(S)** in the *XENIX Programmer's Reference*. The specific failure conditions with error names are contained there also.

12.6.2 Example Program Using msgget

The program example in this section is a menu driven program that exercises all possible combinations the **msgget(S)** system call. This program presents the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) with the required include files as specified by **msgget(S)** in the *XENIX Programmer's Reference*. Note that the `<errno.h>` include file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those named in **msgget(S)**. These names make the program more readable, and are legal since they are local to the program. The variables declared for this program and their purposes are as follows:

Variable	Description
key	Passes the value for the desired key .
opperm	Stores the desired operation permissions.
flags	Stores the desired flags (flags).
opperm_flags	Stores the combination from the logical OR-ing of the opperm and flags variables; it then passes the msgflg argument in the system call.
msqid	Returns the message queue identification number for a successful system call or the error code (-1) for an unsuccessful one.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and for the flag combinations (flags) that are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This lets you look at the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 36-51).

C Library Guide

The system call is made and the results stored at the address of the **msqid** variable (line 53).

Since the **msqid** variable now contains a valid message queue identifier or the error code (-1), it is tested to see if an error occurred (line 55). If **msqid** equals -1, a message indicates that an error resulted, and the external **errno** variable is displayed (lines 57, 58).

If no error occurred, the returned message queue identifier is displayed (line 62).

The following program example is for the **msgget(S)** system call. In the example, the source program file is named *msgget.c* and the executable file is named *msgget*.

Note that if you run the following code example, you must set the flag **IPC_CREAT** before **IPC_EXECL**. If you attempt to set **IPC_EXECL** before **IPC_CREAT**, an error condition will result.

Example

```
1  /*This is a program to illustrate
2  **the message get, msgget(),
3  **system call capabilities.*/

4  #include <stdio.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/msg.h>
8  #include <errno.h>

9  /*Start of main C language program*/
10 main()
11 {
12     key_t key;           /*declare as long integer*/
13     int opperm, flags;
14     int msqid, opperm_flags;
15     /*Enter the desired key*/
16     printf("Enter the desired key in hex = ");
17     scanf("%x", &key);

18     /*Enter the desired octal operation
19     permissions.*/
20     printf("\nEnter the operation\n");
21     printf("permissions in octal = ");
22     scanf("%o", &opperm);
```

Example (cont.)

```

23     /*Set the desired flags.*/
24     printf("\nEnter corresponding number to\n");
25     printf("set the desired flags:\n");
26     printf("No flags           = 0\n");
27     printf("IPC_CREAT             = 1\n");
28     printf("IPC_EXCL                = 2\n");
29     printf("IPC_CREAT and IPC_EXCL = 3\n");
30     printf("           Flags           = ");

31     /*Get the flag(s) to be set.*/
32     scanf("%d", &flags);

33     /*Check the values.*/
34     printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35            key, opperm, flags);

36     /*Incorporate the control fields (flags) with
37        the operation permissions*/
38     switch (flags)
39     {
40     case 0: /*No flags are to be set.*/
41         opperm_flags = (opperm | 0);
42         break;
43     case 1: /*Set the IPC_CREAT flag.*/
44         opperm_flags = (opperm | IPC_CREAT);
45         break;
46     case 2: /*Set the IPC_EXCL flag.*/
47         opperm_flags = (opperm | IPC_EXCL);
48         break;
49     case 3: /*Set the IPC_CREAT and IPC_EXCL flags.*/
50         opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
51     }

52     /*Call the msgget system call.*/
53     msgqid = msgget (key, opperm_flags);

54     /*Perform the following if the call is unsuccessful.*/
55     if(msgqid == -1)
56     {
57         printf ("\nThe msgget system call failed!\n");
58         printf ("The error number = %d\n", errno);
59     }

60     /*Return the msgqid upon successful completion.*/
61     else
62         printf ("\nThe msgqid = %d\n", msgqid);
63     exit(0);
64 }

```

C Library Guide

12.6.3 Controlling Message Queues

This section describes how to use the **msgctl(S)** system call along with an example program that exercises all of its capabilities. The **msgctl** system call has the following syntax:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

The **msgctl** system call requires three arguments to be passed to it and returns an integer value. Upon successful completion, **msgctl** returns a zero value; when unsuccessful, it returns a -1.

The **msqid** variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget** system call.

The **cmd** argument can be replaced by one of the following flags:

Flag	Description
IPC_STAT	Returns the status information contained in the associated data structure for the specified msqid , and places it in the data structure pointed to by buf in the user memory area.
IPC_SET	For the specified msqid , sets the effective user and group identification, operation permissions, and the number of bytes for the message queue.
IPC_RMID	Removes the specified msqid along with its associated message queue and data structure.

You must be the owner/creator of the process or the super-user to use the IPC_SET or IPC_RMID flags. Read permission is required to use the IPC_STAT flag.

The details of this system call are discussed in the following example program for **msgget**.

12.6.4 Example Program Using `msgctl`

The example program in this section is a menu driven program that exercises all possible combinations `msgctl(S)`. This program presents the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) with the include files specified by `msgctl` in the *XENIX Programmer's Reference*. Note that in this program, `errno` is declared as an external variable, and therefore, the `<errno.h>` file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those named in `msgctl`. Their declarations are self-explanatory. These names make the program more readable, and are legal since they are local to the program. The variables are described as follows:

Variable	Purpose
<code>uid</code>	Stores the <code>IPC_SET</code> value for the effective user identification.
<code>gid</code>	Stores the <code>IPC_SET</code> value for the effective group identification.
<code>mode</code>	Stores the <code>IPC_SET</code> value for the operation permissions.
<code>bytes</code>	Stores the <code>IPC_SET</code> value for the number of bytes in the message queue (<code>msg_qbytes</code>).
<code>rtm</code>	Stores the return integer value from the system call.
<code>msgid</code>	Stores and passes the message queue identifier to the system call.
<code>command</code>	Stores the code for the desired flag so that subsequent processing can be performed on it.
<code>choice</code>	Determines which member is to be changed for the <code>IPC_SET</code> flag.
<code>msgid_ds</code>	Receives the specified message queue identifier's data structure when an <code>IPC_STAT</code> flag is performed.

C Library Guide

buf A pointer passed to the system call which locates the data structure in the user memory area where the `IPC_STAT` flag is to place its return values or where the `IPC_SET` command gets the values to set.

Note that the `msqid_ds` data structure in this program (line 16) uses the data structure located in the `<msg.h>` include file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The next important thing to observe is that although `buf` is declared to be a pointer to a data structure of the `msqid_ds` type, it must also be initialized to contain the address of the user memory area data structure (line 17). This completes the explanation for the required declarations.

First, the program prompts for a valid message queue identifier which is stored at the address of the `msqid` variable (lines 19, 20). This is required for every `msgctl` system call.

Now you can enter the code for the desired flag (lines 21-27), and it is stored at the address of the command variable. The code is tested to determine the flag for subsequent processing. The flags and codes are described as follows:

Flag Code and Description

`IPC_STAT` (code 1) The system call is performed (lines 37, 38) and the status information returned is printed out (lines 39-46); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful (line 106), the status information of the last successful call is printed out. In addition, an error message is displayed and the `errno` variable is printed out (lines 108, 109). If the system call is successful, a message indicates this along with the message queue identifier used (lines 111-114).

`IPC_SET` (code 2) The first thing done is to get the current status information for the message queue identifier specified (lines 50-52). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this flag until corrected.

The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 53-59). This code is stored at the address of the choice variable (line 60). Now, depending upon the member picked, the program prompts for the new value (lines 66-95). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (lines 96-98). Depending upon success or failure, the program returns the same messages as for IPC_STAT above.

IPC_RMID (code 3) The system call is performed (lines 100-103), and the **msqid** along with its associated message queue and data structure are removed from the system. Note that **buf** is not required as an argument to perform this control command and its value can be NULL. Depending upon the success or failure, the program returns the same messages as for the other control commands.

The following is an example program for the **msgctl** system call. In the example the source program file is named *msgctl.c* and the executable file is named *msgctl*.

Example

```

1  /*This is a program to illustrate
2  **the message control, msgctl(),
3  **system call capabilities.
4  */

5  /*Include necessary include files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/msg.h>

10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     int uid, gid, mode, bytes;
15     int rtrn, msqid, command, choice;
16     struct msqid_ds msqid_ds, *buf;
17     buf = &msqid_ds;

18     /*Get the msqid, and command.*/
19     printf("Enter the msqid = ");
20     scanf("%d", &msqid);
21     printf("\nEnter the number for\n");
22     printf("the desired command:\n");
23     printf("IPC_STAT    = 1\n");

```

C Library Guide

Example (cont.)

```
24     printf("IPC SET      = 2\n");
25     printf("IPC_RMID    = 3\n");
26     printf("Entry      = ");
27     scanf("%d", &command);
28     /*Check the values.*/
29     printf ("\nmsqid =%d, command = %d\n",
30            msqid, command);

31     switch (command)
32     {
33     case 1: /*Use msgctl() to duplicate
34            the data structure for
35            msqid in the msqid_ds area pointed
36            to by buf and then print it out.*/
37         rtm = msgctl(msqid, IPC_STAT,
38                    buf);
39         printf ("\nThe USER ID = %d\n",
40                buf->msg_perm.uid);
41         printf ("The GROUP ID = %d\n",
42                buf->msg_perm.gid);
43         printf ("The operation permissions = 0%o\n",
44                buf->msg_perm.mode);
45         printf ("The msg_qbytes = %d\n",
46                buf->msg_qbytes);
47         break;
48     case 2: /*Select and change the desired
49            member(s) of the data structure.*/
50         /*Get the original data for this msqid
51            data structure first.*/
52         rtm = msgctl(msqid, IPC_STAT, buf);
53         printf ("\nEnter the number for the\n");
54         printf ("member to be changed:\n");
55         printf ("msg_perm.uid   = 1\n");
56         printf ("msg_perm.gid   = 2\n");
57         printf ("msg_perm.mode  = 3\n");
58         printf ("msg_qbytes   = 4\n");
59         printf ("Entry       = ");

60         scanf("%d", &choice);
61         /*Only one choice is allowed per
62            pass as an illegal entry will
63            cause repetitive failures until
64            msqid_ds is updated with
65            IPC_STAT.*/

66         switch(choice){
67         case 1:
68             printf ("\nEnter USER ID = ");
```

Example (cont.)

```

69         scanf ("%d", &uid);
70         buf->msg_perm.uid = uid;
71         printf("\nUSER ID = %d\n",
72             buf->msg_perm.uid);
73         break;
74     case 2:
75         printf("\nEnter GROUP ID = ");
76         scanf ("%d", &gid);
77         buf->msg_perm.gid = gid;
78         printf("\nGROUP ID = %d\n",
79             buf->msg_perm.gid);
80         break;
81     case 3:
82         printf("\nEnter MODE = ");
83         scanf ("%o", &mode);
84         buf->msg_perm.mode = mode;
85         printf("\nMODE = %o\n",
86             buf->msg_perm.mode);
87         break;
88
89     case 4:
90         printf("\nEnter msg_bytes = ");
91         scanf ("%d", &bytes);
92         buf->msg_qbytes = bytes;
93         printf("\nmsg_qbytes = %d\n",
94             buf->msg_qbytes);
95         break;
96     }
97
98     /*Do the change.*/
99     rtn = msgctl(msqid, IPC_SET,
100         buf);
101     break;
102
103     case 3: /*Remove the msqid along with its
104             associated message queue
105             and data structure.*/
106     rtn = msgctl(msqid, IPC_RMID, NULL);
107     }
108     /*Perform the following if the call is unsuccessful.*/
109     if(rtn == -1)
110     {
111         printf ("\nThe msgctl system call failed!\n");
112         printf ("The error number = %d\n", errno);
113     }
114     /*Return the msqid upon successful completion.*/
115     else
116     printf ("\nMsgctl was successful for msqid = %d\n",
117         msqid);
118     exit (0);
119 }

```

12.6.5 Operations for Messages

This section describes the **msgsnd(S)** and **msgrcv(S)** system calls, along with an example program that exercises all of their capabilities. The **msgop(S)** system call has the following syntax:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

12.6.6 Sending Messages

The **msgsnd** system call requires that you pass four arguments to it and it returns an integer value. Upon successful completion, a zero value is returned; when unsuccessful, **msgsnd** returns a -1. The **msgsnd** arguments are described as follows:

Argument	Description
msqid	Must be a valid, non-negative, integer value. In other words, it must have already been created using the msgget system call.
msgp	A pointer to a structure in the user-memory area that contains the type of the message and the message to be sent.
msgsz	Specifies the length of the character array in the data structure pointed to by the msgp argument. This is the length of the message. The maximum size of this array is determined by the MSGMAX system-tunable parameter.

msgflg Lets the blocking message operation perform if the `IPC_NOWAIT` flag is not set ($(\text{msgflg} \ \& \ \text{IPC_NOWAIT}) == 0$); this would occur if the total number of bytes allowed on the specified message queue were in use (`msg_qbytes` or `MSGMNB`), or the total system-wide number of messages on all queues were equal to the system imposed limit (`MSGTQL`). If the `IPC_NOWAIT` flag is set, the system call will fail and return a -1.

A `msg_qbytes` data structure member can be lowered from `MSGMNB` using the `msgctl IPC_SET` flag, but only the super-user can raise it afterwards.

12.6.7 Receiving Messages

The `msgrcv` system call requires five arguments to be passed to it, and it returns an integer value. Upon successful completion, a value equal to the number of bytes received is returned when unsuccessful it returns a -1. The arguments are described as follows:

Argument Description

msgqid	Must be a valid, non-negative, integer value. In other words, it must have already been created by using the <code>msgget</code> system call.
msgp	A pointer to a structure in the user memory area that will receive the message type and the message text.
msgsz	Specifies the length of the message to be received. If its value is less than the message in the array, an error can be returned if desired; see the <code>msgflg</code> argument.
msgtyp	Picks the first message on the message queue of the particular type specified. If it is equal to zero, the first message on the queue is received; if it is greater than zero, the first message of type <code>msgtyp</code> is received; if it is less than zero, the lowest type that is less than or equal to the absolute value of <code>msgtyp</code> is received.
msgflg	Lets the blocking message operation perform if the <code>IPC_NOWAIT</code> flag is not set ($(\text{msgflg} \ \& \ \text{IPC_NOWAIT}) == 0$); this would occur if there is not a message on the message queue of the desired type (<code>msgtyp</code>) to be received. If the <code>IPC_NOWAIT</code> flag is set, the system call

C Library Guide

will fail immediately when there is not a desired message on the queue. The **msgflg** system call can also specify that it fails if the message is longer than the **size** to be received. This is done by not setting the **MSG_NOERROR** flag in the **msgflg** argument (**(msgflg & MSG_NOERROR) == 0**). If the **MSG_NOERROR** flag is set, the message is truncated to the length specified by the **msgsz** argument of **msgrcv**.

Further details of this system call are discussed in the following example program. For more information, see “Example Program Using msgget.”

12.6.8 Example Program Using msgop

The example program in this section is a menu driven program that exercises all possible combinations of the **msgsnd(S)** and **msgrcv(S)** system calls. This program shows the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This example begins (lines 5-9) by including the required include files as specified by the **msgop**. Note that in this program **errno** is declared as an external variable, and therefore, the **<errno.h>** file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in **msgop**. Their declarations are self-explanatory. These names make the program more readable, and this is legal since they are local to the program. The variables are described as follows:

Variable	Purpose
sndbuf	A buffer that contains a message to be sent (line 13); it uses the msgbuf1 data structure as a template (lines 10-13). The msgbuf1 structure (lines 10-13) is almost an exact duplicate of the msgbuf structure contained in the <msg.h> include file. The only difference is that the character array for msgbuf1 contains the maximum message size (8192), where in msgbuf it is set to one (1) to satisfy the compiler. For this reason, msgbuf cannot be used directly as a template for the user-written program. It is there so you can determine its members.
rcvbuf	A buffer that receives a message (line 13); it uses the msgbuf1 data structure as a template (lines 10-13).

<code>*msgp</code>	A pointer (line 13) to both the <code>sndbuf</code> and <code>rcvbuf</code> buffers.
<code>i</code>	A counter for inputting characters from the keyboard, storing them in the array, and keeping track of the message length for the <code>msgsnd</code> system call; it is also used as a counter to output the received message for the <code>msgrcv</code> system call.
<code>c</code>	Receives the input character from the <code>getchar</code> function (line 50).
<code>flag</code>	Stores the code of <code>IPC_NOWAIT</code> for the <code>msgsnd</code> system call (line 61).
<code>flags</code>	Stores the code of the <code>IPC_NOWAIT</code> or <code>MSG_NOERROR</code> flags for the <code>msgrcv</code> system call (line 117).
<code>choice</code>	Stores the code for sending or receiving (line 30).
<code>rtm</code>	Stores the return values from all system calls.
<code>msqid</code>	Stores and passes the desired message queue identifier for both system calls.
<code>msgsz</code>	Stores and passes the size of the message to be sent or received.
<code>msgflg</code>	Passes the value of flag for sending or the value of flags for receiving.
<code>msgtyp</code>	Specifies the message type for sending, or picks a message type for receiving.

Note that a `msqid_ds` data structure is set up in the program (line 21) with a pointer that is initialized to point to it (line 22); this lets the data structure members that are affected by message operations be observed. They are observed by using the `msgctl` (`IPC_STAT`) system call.

The first thing the program prompts for is whether to send or receive a message. You must enter a corresponding code for the desired operation, and it is stored at the address of the `choice` variable (lines 23-30). Depending upon the code, the program proceeds as in the following `msgsnd` or `msgrcv` sections.

C Library Guide

Using `msgsnd`

When the code is to send a message, the `msgp` pointer is initialized (line 33) to the address of the send data structure, `sndbuf`. Next, a message type must be entered for the message; it is stored at the address of the variable `msgtyp` (line 42), and then (line 43) it is put into the `mtype` member of the data structure pointed to by `msgp`.

The program now prompts for a message to be entered from the keyboard and enters a loop of getting and storing characters into the `mtext` array of the data structure (lines 48-51). This will continue until an end of file is recognized, which for the `getchar` function is CTRL-D immediately following RETURN. When this happens, the `size` of the message is determined by adding one to the `i` counter (lines 52, 53) as it stores the message beginning in the zero array element of `mtext`. Keep in mind that the message also contains the terminating characters, and the message will therefore appear to be three characters short of `msgsz`.

The message is immediately echoed from the `mtext` array of the `sndbuf` data structure to provide feedback (lines 54-56).

You must now decide whether to set the `IPC_NOWAIT` flag. The program requests a code of a 1 be entered for *yes* or anything else for *no* (lines 57-65). It is stored at the address of the flag variable. If a 1 is entered, `IPC_NOWAIT` is logically OR-ed with `msgflg`; otherwise, `msgflg` is set to zero.

The program performs the `msgsnd` system call (line 69). If it is unsuccessful, a failure message is displayed along with the error number (lines 70-72). If it is successful, the returned value, which should be zero, is printed (lines 73-76).

Every time a message is successfully sent, the three elements are three members of the associated data structure that are updated. They are described as follows:

Element	Description
<code>msg_qnum</code>	Represents the total number of messages on the message queue; it is incremented by 1.
<code>msg_lspid</code>	Contains the Process Identification (PID) number of the last process sending a message; it is set accordingly.
<code>msg_stime</code>	Contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) of the last message sent; it is set accordingly.

These members are displayed after every successful message send operation (lines 79-92).

Using `msgrcv`

If the code specifies that a message is to be received, the program continues execution as in the following paragraphs.

The `msgp` pointer is initialized to the `rcvbuf` data structure (line 99).

The message queue identifier of the next message is stored at the address of `msqid` (lines 100-103).

The message type is requested, and it is stored at the address of `msgtyp` (lines 104-107).

The program then requests code for the desired combination of control flags, and stores the code at the address of `flags` (lines 108-117). Depending upon the selected combination, `msgflg` is set accordingly (lines 118-133).

The program requests the number of bytes to be received and stores this number at the address of `msgsz` (lines 134-137).

The program can now perform the `msgrcv` system call (line 144). If it is unsuccessful, the program displays a message and error number (lines 145-148). If successful, a message indicates so, and the program displays the number of bytes returned, followed by the received message (lines 153-159).

When a message is successfully received, there are three elements of the associated data structure that are updated; they are described as follows:

Element	Description
<code>msg_qnum</code>	Contains the number of messages on the message queue; it is decremented by one.
<code>msg_lrpid</code>	Contains the process identification (PID) of the last process receiving a message; it is set accordingly.
<code>msg_rtime</code>	Contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) that the last process received a message; it is set accordingly.

C Library Guide

The example program for the **msgop** system call follows. In the example the source file is named *msgop.c* and the executable file is named *msgop*.

Example

```
1  /*This is a program to illustrate
2  **the message operations, msgop(),
3  **system call capabilities.
4  */

5  /*Include necessary include files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/msg.h>

10 struct msgbuf1 {
11     long mtype;
12     char mtext[8192];
13 } sndbuf, rcvbuf, *msgp;

14 /*Start of main C language program*/
15 main()
16 {
17     extern int errno;
18     int i, c, flag, flags, choice;
19     int rtm, msqid, msgsz, msgflg;
20     long mtype, msgtyp;
21     struct msqid_ds msqid_ds, *buf;
22     buf = &msqid_ds;

23     /*Select the desired operation.*/
24     printf("Enter the corresponding\n");
25     printf("code to send or\n");
26     printf("receive a message:\n");
27     printf("Send           = 1\n");
28     printf("Receive          = 2\n");
29     printf("Entry            = ");
30     scanf("%d", &choice);

31     if(choice == 1) /*Send a message.*/
32     {
33         msgp = &sndbuf; /*Point to user send structure.*/

34         printf("\nEnter the msqid of\n");
35         printf("the message queue to\n");
36         printf("handle the message = ");
37         scanf("%d", &msqid);
```

Example (cont.)

```

38      /*Set the message type.*/
39      printf("\nEnter a positive integer\n");
40      printf("message type (long) for the\n");
41      printf("message = ");
42      scanf("%d", &msgtyp);
43      msgp->mtype = msgtyp;

44      /*Enter the message to send.*/
45      printf("\nEnter a message: \n");

46      /*A control-d (^d) terminates as
47      EOF.*/

48      /*Get each character of the message
49      and put it in the mtext array.*/
50      for(i = 0; ((c = getchar()) != EOF); i++)
51          sndbuf.mtext[i] = c;

52      /*Determine the message size.*/
53      msgsz = i + 1;

54      /*Echo the message to send.*/
55      for(i = 0; i < msgsz; i++)
56          putchar(sndbuf.mtext[i]);

57      /*Set the IPC_NOWAIT flag if
58      desired.*/
59      printf("\nEnter a 1 if you want the\n");
60      printf("the IPC_NOWAIT flag set: ");
61      scanf("%d", &flag);
62      if(flag == 1)
63          msgflg |= IPC_NOWAIT;
64      else
65          msgflg = 0;

66      /*Check the msgflg.*/
67      printf("\nmsgflg = 0%o\n", msgflg);

68      /*Send the message.*/
69      rtrn = msgsnd(msgqid, msgp, msgsz, msgflg);
70      if(rtrn == -1)
71          printf("\nMsgsnd failed. Error = %d\n",
72              errno);
73      else {
74          /*Print the value of test which
75          should be zero for successful.*/
76          printf("\nValue returned = %d\n", rtrn);

77          /*Print the size of the message
78          sent.*/

```

C Library Guide

Example (cont.)

```
79         printf("\nMsgsz = %d\n", msgsz);
80
81         /*Check the data structure update.*/
82         msgctl(msqid, IPC_STAT, buf);
83         /*Print out the affected members.*/
84         /*Print the incremented number of
85         messages on the queue.*/
86         printf("\nThe msg_qnum = %d\n",
87             buf->msg_qnum);
88         /*Print the process id of the last sender.*/
89         printf("The msg_lspid = %d\n",
90             buf->msg_lspid);
91         /*Print the last send time.*/
92         printf("The msg_stime = %d\n",
93             buf->msg_stime);
94     }
95     if(choice == 2) /*Receive a message.*/
96     {
97         /*Initialize the message pointer
98         to the receive buffer.*/
99         msgp = &rcvbuf;
100
101         /*Specify the message queue which contains
102         the desired message.*/
103         printf("\nEnter the msqid = ");
104         scanf("%d", &msqid);
105
106         /*Specify the specific message on the queue
107         by using its type.*/
108         printf("\nEnter the msgtyp = ");
109         scanf("%d", &msgtyp);
110
111         /*Configure the control flags for the
112         desired actions.*/
113         printf("\nEnter the corresponding code\n");
114         printf("to select the desired flags: \n");
115         printf("No flags                = 0\n");
116         printf("MSG_NOERROR                    = 1\n");
117         printf("IPC_NOWAIT                       = 2\n");
118         printf("MSG_NOERROR and IPC_NOWAIT      = 3\n");
119         printf("Flags                            = ");
120         scanf("%d", &flags);
121
122         switch(flags) {
123             /*Set msgflg by ORing it with the appropriate
124             flags (constants).*/
125             case 0:
126                 msgflg = 0;
127                 break;
```

Example (cont.)

```

124         case 1:
125             msgflg |= MSG_NOERROR;
126             break;
127         case 2:
128             msgflg |= IPC_NOWAIT;
129             break;
130         case 3:
131             msgflg |= MSG_NOERROR | IPC_NOWAIT;
132             break;
133     }

134     /*Specify the number of bytes to receive.*/
135     printf("\nEnter the number of bytes\n");
136     printf("to receive (msgsz) = ");
137     scanf("%d", &msgsz);
138     /*Check the values for the arguments.*/
139     printf("\nmsgid = %d\n", msgid);
140     printf("\nmsgtyp = %d\n", msgtyp);
141     printf("\nmsgsz = %d\n", msgsz);
142     printf("\nmsgflg = %o\n", msgflg);

143     /*Call msgrcv to receive the message.*/
144     rtrn = msgrcv(msgid, msgp, msgsz, msgtyp, msgflg);

145     if(rtrn == -1) {
146         printf("\nMsgrcv failed. ");
147         printf("Error = %d\n", errno);
148     }
149     else {
150         printf ("\nMsgctl was successful\n");
151         printf("for msgid = %d\n",
152             msgid);

153         /*Print the number of bytes received,
154            it is equal to the return
155            value.*/
156         printf("Bytes received = %d\n", rtrn);

157         /*Print the received message.*/
158         for(i = 0; i<=rtrn; i++)
159             putchar(rcvbuf.mtext[i]);
160     }
161     /*Check the associated data structure.*/
162     msgctl(msgid, IPC_STAT, buf);
163     /*Print the decremented number of messages.*/
164     printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
165     /*Print the process id of the last receiver.*/
166     printf("The msg_lpid = %d\n", buf->msg_lpid);
167     /*Print the last message receive time*/
168     printf("The msg_rtime = %d\n", buf->msg_rtime);
169 }
170 }

```

C Library Guide

12.7 Overview of Semaphores

The standard C library provides a group of functions, called the semaphore functions that you can use to control access to a given system resource. These functions create, open, and request control of *semaphores*. You can use semaphores to control a system resource, such as a data file, by requiring that a process gain control of the semaphore before attempting to access the resource. A process that wishes to take control of a semaphore away from another process must wait until that process relinquishes control.

XENIX System V supports two sets of system calls for dealing with semaphore operations. These are referred to in subsequent descriptions as either XENIX semaphore or UNIX System V semaphores.

The XENIX semaphore operations are compatible with previous releases of XENIX. Semaphores are regular files that have names and entries in the file system, but contain no data. Unlike other files, semaphores cannot be accessed by more than one process at a time. The system calls for manipulating XENIX semaphores are: **opensem**, **creatsem**, **sigsem**, **nbwaitsem**, and **waitsem**.

The UNIX System V semaphore operations are compatible with AT&T UNIX System V. The system calls for manipulating UNIX semaphore are: **semop**, **semctl**, and **semget**.

XENIX and UNIX semaphores are not compatible. The operations mentioned are valid only for one type of semaphore, and cannot be applied to the other type. This section describes the XENIX semaphore operations in detail since these system calls are unique to the XENIX operating system. The next section describes the UNIX system semaphore operations in detail. For additional information on UNIX semaphore operations, see **intro(S)**, **semop(S)**, **semctl(S)**, and **semget(S)** in the *XENIX Programmer's Reference*.

12.8 Using Semaphores Under XENIX

There are five XENIX semaphore functions; they are described as follows:

Function	Description
creatsem	Creates a semaphore, returning a semaphore number that can be opened and used in subsequent semaphore functions.

<code>opensem</code>	Opens an existing semaphore for use by the given process.
<code>waitsem</code>	Requests control of a given semaphore for the calling process. If the semaphore is available, control is given immediately. Otherwise, the process waits.
<code>nbwaitsem</code>	Requests control of a given semaphore for the calling process. If the semaphore is available, control is given immediately. If the semaphore is not available, nbwaitsem does not cause the calling process to wait and returns an error value.
<code>sigsem</code>	Causes a process to relinquish control of a given semaphore and to signal this fact to all processes waiting for the semaphore.

12

12.8.1 Creating a Semaphore

The **creatsem** function creates a semaphore, returning a semaphore number that can be used in subsequent semaphore functions. The function call has the following form:

creatsem (*sem_name*, *mode*)

where

- *sem_name* is a character pointer to the name of the semaphore.
- *mode* is an integer value that defines the access mode of the semaphore.

Semaphore names have the same syntax as regular filenames. The names must be unique. The function normally returns an integer semaphore number that is used in subsequent semaphore functions to refer to the semaphore. The function returns -1 if it encounters an error, such as creating a semaphore that already exists or using the name of an existing regular file.

C Library Guide

The function is typically used at the beginning of one process to clearly define the semaphores it intends to share with other processes. For example, in the following program fragment **creatsem** creates a semaphore named *tty1* before proceeding with its tasks:

```
main ()
{
  int tty1;
  FILE fttty1;

  tty1 = creatsem("tty1", 0777);
  fttty1 = fopen("/dev/tty01", "w");
  /* Program body. */
}
```

Note that **fopen** is used immediately after **creatsem** to open the */dev/tty01* file for writing. This is one way to make the association between a semaphore and a device clear.

The mode “0777” defines the semaphore’s access permissions. The permissions are similar to the permissions of a regular file. A semaphore can have read permission for the owner, for users in the same group as the owner, and for all other users. The write and execution permissions have no meaning. Thus, “0777” means read permission for all users.

No more than one process ever need create a given semaphore; all other processes simply open the semaphore with the **opensem** function. Once created or opened, a semaphore can be accessed only by using the **waitsem**, **nbwaitsem**, or **sigsem** functions. The **creatsem** function can be used more than once during execution of a process. In particular, it can be used to reset a semaphore if a process fails to relinquish control before terminating. Before resetting a semaphore, you must remove the associated semaphore file using the **unlink** function.

12.8.2 Opening a Semaphore

The **opensem** function opens an existing semaphore for use by the given process. The function call has the following form:

```
opensem (sem_name)
```

where *sem_name* is a pointer to the name of the semaphore. This must be the same name used when creating the semaphore. The function returns a semaphore number that can be used in subsequent semaphore functions to refer to the semaphore. The function returns -1 if it encounters an error, such as trying to open a semaphore that does not exist or using the name of an existing regular file.

A process uses this function before it requests control of a given semaphore. A process need not use the function if it also created the semaphore. For example, in the following program fragment **opensem** is used to open the semaphore named *semaphore1*:

```
main ()
{
  int sem1;

  if ( (sem1 = opensem("semaphore1")) != -1)
    waitsem(sem1);
```

12

In this example, the semaphore number is assigned to the variable *sem1*. If the number is not -1, then *sem1* is used in the semaphore function, **waitsem**, which requests control of the semaphore.

You must not open a semaphore more than once during execution of a process. Although the **opensem** function does not return an error value, opening a semaphore more than once can lead to a system deadlock.

12.8.3 Requesting Control of a Semaphore

The **waitsem** function requests control of a given semaphore for the calling process. If the semaphore is available, control is given immediately. Otherwise, the process waits. The function call has the following form:

waitsem (*sem_num*)

where *sem_num* is the semaphore number of the semaphore to be controlled. If the semaphore is not available (if it is under control of another process), the function forces the requesting process to wait. If other processes are already waiting for control, the request is placed next in a queue of requests. When the semaphore becomes available, the first process to request control receives it. When this process relinquishes control, the next process receives control, and so on. The function returns a -1 if it encounters an error, such as a request for a semaphore that does not exist or a request for a semaphore that is locked to a dead process.

C Library Guide

The **waitsem** function is used whenever a given process wishes to access the device or system resource associated with the semaphore. For example, in the following program fragment, **waitsem** signals the intention to write to the file given by *tty1*:

```
main ()
{
  int tty1;
  FILE fttty1;

  waitsem( tty1 );
  fprintf( fttty1, "Changing tty driver\n");
}
```

The function waits until the current controlling process relinquishes control of the semaphore before returning to the next statement.

12.8.4 Checking the Status of a Semaphore

The **nbwaitsem** function requests control of a given semaphore, but does not cause the calling process to wait if the requested semaphore is not available. In other words, if the semaphore is not available, the function returns an error value. Otherwise, it gives immediate control of the semaphore to the calling process. The function call has the following form:

nbwaitsem (*sem_num*)

where *sem_num* is the semaphore number of the semaphore to be checked. The function returns a -1 if another process has control of the semaphore or if it encounters an error, such as a request for a semaphore that does not exist. The function also returns a -1 if the process controlling the requested semaphore terminates without relinquishing control of the semaphore.

The **nbwaitsem** function lets you take control of a semaphore without blocking, instead of using **waitsem**.

12.8.5 Relinquishing Control of a Semaphore

The **sigsem** function causes a process to relinquish control of a given semaphore and to signal this fact to all processes waiting for the semaphore. The function call has the following form:

sigsem (*sem_num*)

where *sem_num* is the semaphore number of the semaphore to relinquish.

The process must have created or opened the semaphore previously. Furthermore, the process must have previously taken control of the semaphore with the **waitsem** or **nwaitsem** function. The function returns a -1 if it encounters an error, such as trying to relinquish control of a semaphore that does not exist.

You use this function after a process has finished accessing a device or system resource not responding to a semaphore. This lets waiting processes take control. For example, in the following program fragment **sigsem** signals the end of control of the semaphore *tty1*:

```
main ()
{
  int tty1;
  FILE temp, fttty1;

  waitsem( tty1 );
  while ((c=fgetc(temp)) != EOF)
    fputc(c, fttty1);
  sigsem( tty1 );
```

This example also signals the end of the copy operation to the semaphore's corresponding device, given by *ftty1*.

Note that a semaphore can become locked to a dead process if the process fails to relinquish control before terminating. In such a case, the **creatsem** function must reset the semaphore.

12.8.6 Program Example

This section shows you how to use the XENIX semaphore functions to control the access of a system resource. The following program creates five different processes that vie for control of a semaphore. Each process requests control of the semaphore five times, holding control for one second, then releasing it. Although the following program performs no meaningful work, it illustrates the use of XENIX semaphores.

C Library Guide

Example

```
#define      NPROC 5

char  semf[] = "_kesemfXXXXXX";
int   sem_num;
int   holdsem = 5;

main()
{
    register i, chid;

    mktemp(semf);
    if ((sem_num = creatsem(semf, 0777)) < 0)
        err("creatsem");
    for (i = 1; i < NPROC; ++i) {
        if((chid = fork()) < 0)
            err("No fork");
        else if(chid == 0) {
            if((sem_num = opensem(semf)) < 0)
                err("opensem");
            doit(i);
            exit(0);
        }
    }
    doit(0);
    for (i = 1; i < NPROC; ++i)
        while(wait((int *)0) < 0)
            ;
    unlink(semf);
}

doit(id)
{
    while(holdsem--> 0) {
        if(waitsem(sem_num) < 0)
            err("waitsem");
        printf("%d\n", id);
        sleep(1);
        if(sigsem(sem_num) < 0)
            err("sigsem");
    }
}

err(s)
char *s;
{
    perror(s);
    exit(1);
}
```

Program Notes

The program contains a number of global variables. The *semf* array contains the semaphore name used by the **creatsem** and **opensem** functions. The variable *sem_num* is the semaphore number returned by **creatsem** and **opensem** and used eventually in **waitsem** and **sigsem**. Finally, the variable *holdsem* contains the number of times each process requests control of the semaphore.

The main program function uses the **mktemp** function to create a unique name for the semaphore and then uses the name with **creatsem** to create the semaphore. Once the semaphore is created, the main process begins to create child processes. These processes will eventually vie for control of the semaphore. As each child process is created, it opens the semaphore and calls the **doit** function. When control returns from **doit**, the child process terminates. The parent process also calls the **doit** function, then waits for termination of each child process, and finally deletes the semaphore with the **unlink** function.

The **doit** function calls the **waitsem** function to request control of the semaphore. The function waits until the semaphore is available; it then prints the process ID number (*PID*) to the standard output, waits one second, and relinquishes control using the **sigsem** function.

Each step of the program is checked for possible errors. If an error is encountered, the program calls the **err** function. This function prints an error message and terminates the program.

12.9 Using Semaphores Under UNIX System V

This section describes the system calls for the semaphore type of Inter-Process Communication (IPC). The semaphore type of IPC lets processes communicate through the exchange of semaphore values. A semaphore is a positive integer (0 through 32,767). Since many applications require the use of more than one semaphore, the UNIX operating system has the ability to create sets or arrays of semaphores. A semaphore set can contain one or more semaphores up to a limit set by the system administrator. The tunable parameter, **SEMMSL**, has a default value of 25. You can create semaphore sets using the **semget(S)** system call.

The process performing the **semget(S)** system call becomes the owner/creator, determines how many semaphores are in the set, and sets the operation permissions for the set. This process can subsequently relinquish ownership of the set or change the operation permissions using the **semctl(S)** (semaphore control). The creating process always remains

C Library Guide

the creator as long as the facility exists. Other processes with permission can use **semctl** to perform other control functions.

Provided a process has alter permission, it can manipulate a semaphore. Each semaphore within a set can be manipulated in two ways with the **semop(S)** system call:

- incremented
- decremented

To increment a semaphore, pass an integer value to the **semop(S)** system call. To decrement a semaphore, pass a minus (-) value.

The system ensures that only one process can manipulate a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary manner.

A process can test for a semaphore value to be greater than a certain value by attempting to decrement the semaphore by one more than that value. If the process is successful, then the semaphore value is greater than that certain value. Otherwise, the semaphore value is not. While doing this, the process can have its execution suspended (IPC_NOWAIT flag not set) until the semaphore value would permit the operation (other processes increment the semaphore), or the semaphore facility is removed.

Blocking and Nonblocking Semaphore Operation

The ability to suspend execution is called a *blocking semaphore operation*. This ability is also available for a process that is testing a semaphore to become zero or equal to zero; only read permission is required for this test and it is accomplished by passing a value of zero to the **semop(S)** system call.

On the other hand, if the process is not successful and the process does not request to have its execution suspended, it is called a *nonblocking semaphore operation*. In this case, the process is returned a known error code (-1), and the external **errno** variable is set accordingly.

The blocking semaphore operation lets processes communicate based on the values of semaphores at different points in time. Remember also that IPC facilities remain in the UNIX operating system until removed by a permitted process or until the system is reinitialized.

Operating on a semaphore set is done by using the **semop(S)**, semaphore operation system call.

When a set of semaphores is created, the first semaphore in the set is semaphore number zero. The last semaphore number in the set is one less than the total number in the set.

An array of these blocking/nonblocking semaphore operations can be performed on a set containing more than one semaphore. When performing an array of operations, the blocking/nonblocking semaphore operations can be applied to any or all of the semaphores in the set. Also, the operations can be applied in any order of semaphore number. However, no operations are done until they can all be done successfully.

For example, if a process has successfully completed three of six operations on a set of ten semaphores but is *blocked* from performing the fourth operation, no changes are made to the set until the fourth and remaining operations are successfully performed. Additionally, any operation preceding or succeeding the blocked operation (including the blocked operation) can specify that at such time that all operations can be performed successfully, that the operation be undone. Otherwise, the operations are performed and the semaphores are changed, or one nonblocking operation is unsuccessful and none are changed. All of this is commonly referred to as being *atomically performed*.

The ability to undo operations requires the system to maintain an array of *undo structures* corresponding to the array of semaphore operations to be performed. If necessary, each semaphore operation that is to be undone has an associated *adjust variable* used for undoing the operation.

Remember, any unsuccessful nonblocking operation for a single semaphore or a set of semaphores causes immediate return with no operations performed at all. When this occurs, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

System calls make these semaphore capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

12.9.1 Semaphore Data Structures and Arrays

Before semaphores can be used (operated on or controlled) a uniquely identified **data structure** and **semaphore set** (array) must be created. The unique identifier is called the semaphore identifier (**semid**); **semid** identifies or references a particular data structure and semaphore set.

C Library Guide

The semaphore set contains a predefined number of structures in an array, one structure for each semaphore in the set. A user can select the number of semaphores (**nsems**) in a semaphore set. The following members are in each structure within a semaphore set:

- a semaphore text map address
- the process identification (PID) performing last operation
- the number of processes awaiting the semaphore value to become greater than its current value
- the number of processes awaiting the semaphore value to equal zero

There is one associated data structure for the uniquely identified semaphore set. This data structure contains information related to the semaphore set as follows:

- operation permissions data (operation permissions structure)
- pointer to first semaphore in the set (array)
- number of semaphores in the set
- last semaphore operation time
- last semaphore change time

The C programming language data structure definition for the semaphore set (array member) is as follows:

```
struct sem
{
    ushort  semval;      /* semaphore text map address */
    short   sempid;     /* pid of last operation */
    ushort  semncnt;    /* # awaiting semval > cval */
    ushort  semzcnt;    /* # awaiting semval = 0 */
};
```

It is located in the `<sys/sem.h>` include file.

Likewise, the structure definition for the associated semaphore data structure is as follows:

```

struct semid_ds
{
    struct ipc_perm sem_perm; /* operation permission struct */
    struct sem      *sem_base; /* ptr to first semaphore in set */
    ushort         sem_nsems; /* # of semaphores in set */
    time_t         sem_otime; /* last semop time */
    time_t         sem_ctime; /* last change time */
};

```

12

It is also located in the `<sys/sem.h>` include file. Note that the **sem_perm** member of this structure uses **ipc_perm** as a template.

The **ipc_perm** data structure is the same for all IPC facilities, and it is located in the `<sys/ipc.h>` include file. It is shown in “Message Operations.”

The **semget(S)** system call performs the following two tasks when only the **IPC_CREAT** flag is set in the **semflg** argument that it receives:

- gets a new **semid** and creates an associated data structure and semaphore set for it
- returns an existing **semid** that already has an associated data structure and semaphore set

The task performed is determined by the value of the **key** argument passed to the **semget(S)** system call. For the first task, if the **key** is not already in use for an existing **semid**, a new **semid** is returned with an associated data structure and semaphore set created for it, provided no system-tunable parameter would be exceeded.

There is also a provision for specifying a **key** of value **IPC_PRIVATE** which is known as the private **key**. When specified, a new **semid** is always returned with an associated data structure and semaphore set created for it, unless a system-tunable parameter would be exceeded. When the **ipcs** command is performed, the **KEY** field for the **semid** is all zeros.

When performing the first task, the process that calls **semget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator. (For more information, see “Controlling Semaphores.”) The creator of the semaphore set also determines the initial operation permissions for the facility.

C Library Guide

For the second task, if a **semid** exists for the **key** specified, the value of the existing **semid** is returned. If it is not desired to have an existing **semid** returned, a flag (**IPC_EXCL**) can be specified (set) in the **semflg** argument passed to the system call. The system call will fail if it is passed a value for the number of semaphores (**nsems**) that is greater than the number actually in the set. If you do not know how many semaphores are in the set, use 0 for **nsems**. For more information, see “Example Program Using semget.”

Once a uniquely identified semaphore set and data structure are created, semaphore operations [**semop(S)**] and semaphore control [**semctl**] can be used.

Semaphore operations consist of incrementing, decrementing, and testing for zero. A single system call is used to perform these operations. It is called **semop**. For details of this system call, see “Operations on Semaphores.”

The **semctl(S)** system call performs semaphore control. These control operations let you control the semaphore facility by:

- Returning the value of a semaphore
- Setting the value of a semaphore
- Returning the process identification (PID) of the last process performing an operation on a semaphore set
- Returning the number of processes waiting for a semaphore value to become greater than its current value
- Returning the number of processes waiting for a semaphore value to equal zero
- Getting all semaphore values in a set and placing them in an array in user memory
- Setting all semaphore values in a semaphore set from an array of values in user memory
- Placing all data structure member values and status of a semaphore set into user memory area
- Changing operation permissions for a semaphore set

- Removing a particular **semid** from the UNIX operating system along with its associated data structure and semaphore set

For details of the **semctl(S)** system call, see “Controlling Semaphores.”

12.10 Getting Semaphores

This section contains a detailed description of using the **semget(S)** system call along with an example program illustrating its use. The function call has the following syntax:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key, nsems, semflg)
key_t key;
int nsems, semflg;
```

The following line in the syntax informs you that **semget** is a function with three formal arguments that returns an integer type value upon successful completion (**semid**):

```
int semget (key, nsems, semflg)
```

The next two lines declare the types of the formal arguments:

```
key_t key;
int nsems, semflg;
```

where **key_t** is declared by a **typedef** in the **types.h** include file to be a long integer.

Upon successful completion, the integer returns the semaphore set identifier (**semid**).

As declared, the process calling the **semget** system call must supply three actual arguments to be passed to the formal **key**, **nsems**, and **semflg** arguments.

A new **semid** with an associated semaphore set and data structure is provided if either

- **key** is equal to **IPC_PRIVATE**
- or
- **key** is passed a unique hexadecimal integer, and **semflg** ANDed with **IPC_CREAT** is “true” (not zero).

C Library Guide

The value passed to the **semflg** argument must be an integer type value and will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes and execute modes determine the user/group/other attributes of the **semflg** argument. They are collectively referred to as *operation permissions*. The following table reflects the numeric values (expressed in octal notation) for the valid operation permissions codes:

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants defined in the `<sem.h>` include file that can be used for the user (OWNER). They are as follows:

```
SEM_A    0200    /* alter permission by owner */
SEM_R    0400    /* read permission by owner  */
```

The following table contains the names of the constants that apply to the **semget(S)** system call along with their values.

Value	Flag
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for **semflg** is, therefore, a combination of operation permissions and flags. After determining the value for the operation permissions as previously described, the desired flags can be specified. This specification is accomplished by bitwise OR-ing (|) them with the operation permissions; the bit positions and values for the flags in relation to those of the operation permissions make this possible. It is illustrated as follows:

Name		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
CW ORed by User	=	0 0 4 0 0	0 000 000 100 000 000
semflg	=	0 1 4 0 0	0 000 001 100 000 000

The **semflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
semid = semget (key, nsems, (IPC_CREAT | 0400));
semid = semget (key, nsems, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by **semget(S)**, success or failure of this system call depends upon the actual argument values for **key**, **nsems**, **semflg** or system-tunable parameters. The system call will attempt to return a new **semid** if one of the following conditions is true:

- **Key** is equal to **IPC_PRIVATE**
- **Key** does not already have a **semid** associated with it, and (**semflg** ANDed **IPC_CREAT**) is “true” (not zero).

Key Argument **IPC_PRIVATE**

The **key** argument can be set to **IPC_PRIVATE** in the following way:

```
semid = semget (IPC_PRIVATE, nsems, semflg);
```

C Library Guide

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the SEMMNI, SEMMNS, or SEMMSL system-tunable parameters will always cause a failure. The parameters are described as follows:

Parameter	Description
SEMMNI	Determines the maximum number of unique semaphore sets (semid 's) in the system.
SEMMNS	Determines the maximum number of semaphores in all semaphore sets system wide.
SEMMSL	Determines the maximum number of semaphores in each semaphore set.

The second condition is satisfied if the value for **key** is not already associated with a **semid**, and the bitwise ANDing of **semflg** and IPC_CREAT is "true" (not zero). This means that the **key** is unique (not in use) within the system for this facility type and that the IPC_CREAT flag is set (**semflg** | IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
semflg = x 1 x x x    (x = immaterial)
& IPC_CREAT = 0 1 0 0 0
result = 0 1 0 0 0    (not zero)
```

Since the result is not zero, the flag is set or "true." SEMMNI, SEMMNS, and SEMMSL apply here also, just as for the first condition.

IPC_EXCL is another flag used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a **semid** exists for the specified key provided. This is necessary to prevent the process from thinking that it has received a new (unique) **semid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new **semid** is returned if the system call is successful. Any value for **semflg** returns a new **semid** if the key equals IPC_PRIVATE and no system-tunable parameters are exceeded.

For more information on associated data-structure initialization, see **semget(S)** in the *XENIX Programmer's Reference*.

12.10.1 Example Program Using semget

The example program in this section is a menu driven program that exercises all possible combinations the **semget(S)** system call.

This program presents the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by **semget(S)**. Note that the `<errno.h>` include file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those listed in **semget**. These names make the program more readable, and this is legal since they are local to the program. The variables are described as follows:

Variable	Description
key	Passes the value for the desired key.
opperm	Stores the desired operation permissions.
flags	Stores the desired flags (flags).
opperm_flags	Stores the combination from the logical OR-ing of the opperm and flags variables; it is then used in the system call to pass the semflg argument.
semid	Returns the semaphore set identification number for a successful system call or the error code (-1) for an unsuccessful one.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and the flag combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be possible. This lets you observe the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 36-52).

Then, the number of semaphores for the set is requested (lines 53-57), and its value is stored at the address of **nsems**.

C Library Guide

The system call is made, and the result is stored at the address of the **semid** variable (lines 60, 61).

Since the **semid** variable now contains a valid semaphore set identifier or the error code (-1), it is tested to see if an error occurred (line 63). If **semid** equals -1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 65, 66). Remember that the external **errno** variable is only set when a system call fails; it should only be tested immediately following system calls.

If no error occurred, the returned semaphore set identifier is displayed (line 70).

The example program for the **semget(S)** system call follows. In the example the source program file is named *semget.c* and the executable file is named *semget*.

Example

```
1  /*This is a program to illustrate
2  **the semaphore get, semget(),
3  **system call capabilities.*/
4  #include <stdio.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/sem.h>
8  #include <errno.h>
9  /*Start of main C language program*/
10 main()
11 {
12     key_t key;    /*declare as long integer*/
13     int opperm, flags, nsems;
14     int semid, opperm_flags;
15     /*Enter the desired key*/
16     printf("\nEnter the desired key in hex = ");
17     scanf("%lx", &key);
18     /*Enter the desired octal operation
19     permissions.*/
20     printf("\nEnter the operation\n");
21     printf("permissions in octal = ");
22     scanf("%o", &opperm);
```

Example (cont.)

```

23     /*Set the desired flags.*/
24     printf("\nEnter corresponding number to\n");
25     printf("set the desired flags:\n");
26     printf("No flags           = 0\n");
27     printf("IPC_CREAT           = 1\n");
28     printf("IPC_EXCL             = 2\n");
29     printf("IPC_CREAT and IPC_EXCL = 3\n");
30     printf("           Flags       = ");
31     /*Get the flags to be set.*/
32     scanf("%d", &flags);

33     /*Error checking (debugging)*/
34     printf ("\nkey =0x%x, opperm = 0%o, flags = 0%\n",
35            key, opperm, flags);
36     /*Incorporate the control fields (flags) with
37        the operation permissions.*/
38     switch (flags)
39     {
40     case 0: /*No flags are to be set.*/
41         opperm_flags = (opperm | 0);
42         break;
43     case 1: /*Set the IPC_CREAT flag.*/
44         opperm_flags = (opperm | IPC_CREAT);
45         break;
46     case 2: /*Set the IPC_EXCL flag.*/
47         opperm_flags = (opperm | IPC_EXCL);
48         break;
49     case 3: /*Set the IPC_CREAT and IPC_EXCL
50            flags.*/
51         opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
52     }

53     /*Get the number of semaphores for this set.*/
54     printf("\nEnter the number of\n");
55     printf("desired semaphores for\n");
56     printf("this set (25 max) = ");
57     scanf("%d", &nsems);

58     /*Check the entry.*/
59     printf("\nNsems = %d\n", nsems);

60     /*Call the semget system call.*/
61     semid = semget(key, nsems, opperm_flags);

62     /*Perform the following if the call is unsuccessful.*/
63     if(semid == -1)
64     {
65         printf("The semget system call failed!\n");
66         printf("The error number = %d\n", errno);
67     }
68     /*Return the semid upon successful completion.*/
69     else
70         printf("\nThe semid = %d\n", semid);
71     exit(0);
72 }

```

C Library Guide

12.10.2 Controlling Semaphores

This section describes the **semctl(S)** system call along with an example program that exercises all of its capabilities. The **semctl(S)** system call has the following syntax:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun
{
    int val;
    struct semid_ds *buf;
    ushort *array;
} arg;
```

The **semctl(S)** system call requires four arguments to be passed to it, and it returns an integer value. The **semid** argument must be a valid, non-negative, integer value that has already been created by using the **semget(S)** system call.

The **semnum** argument is used to select a semaphore by its number. This relates to array (atomically performed) operations on the set. When a set of semaphores is created, the first semaphore is number 0, and the last semaphore has the number of one less than the total in the set.

The **cmd** argument can be replaced by one of the following flags:

Flag	Description
GETVAL	Returns the value of a single semaphore within a semaphore set.
SETVAL	Sets the value of a single semaphore within a semaphore set.
GETPID	Returns the Process Identifier (PID) of the process that performed the last operation on the semaphore within a semaphore set.

GETNCNT	Returns the number of processes waiting for the value of a particular semaphore to become greater than its current value.
GETZCNT	Returns the number of processes waiting for the value of a particular semaphore to be equal to zero.
GETALL	Returns the values for all semaphores in a semaphore set.
SETALL	Sets all semaphore values in a semaphore set.
IPC_STAT	Returns the status information contained in the associated data structure for the specified semid , and places it in the data structure pointed to by the buf pointer in the user memory area; arg.buf is the union member that contains the value of buf .
IPC_SET	For the specified semaphore set (semid), sets the effective user/group identification and operation permissions.
IPC_RMID	Removes the specified (semid) semaphore set along with its associated data structure.

12

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID flag. Read/write permission is required as applicable for the other flags.

The **arg** argument is used to pass the system call the appropriate union member for the flag to be performed:

- **arg.val**
- **arg.buf**
- **arg.array**

For more information, see “Example Program Using semget.”

12.10.3 Example Program Using semctl

The program example in this section is a menu driven program that exercises all possible combinations of **semctl(S)**.

C Library Guide

This program presents the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required include files as specified by **semctl(S)**. Note that in this program **errno** is declared as an external variable, and therefore the `<errno.h>` include file does not have to be included.

Variable, structure, and union names have been chosen to be as close as possible to those listed in *semctl*. These names make the program more readable, and this is legal since they are local to the program. These are described as follows:

Name	Description
semid_ds	Receives the specified semaphore set identifier's data structure when an IPC_STAT flag is performed.
c	Receives the input values from the scanf(S) function, (line 117) when performing a SETALL flag.
i	A counter that increments through the union arg.array when displaying the semaphore values for a GETALL (lines 97-99) flag, and when initializing the arg.array when performing a SETALL (lines 115-119) flag.
length	A variable that tests for the number of semaphores in a set against the i counter variable (lines 97, 115).
uid	Stores the IPC_SET value for the effective user identification.
gid	Stores the IPC_SET value for the effective group identification.
mode	Stores the IPC_SET value for the operational permissions.
retrn	Stores the return integer from the system call which depends upon the control command or a -1 when unsuccessful.
semid	Stores and passes the semaphore set identifier to the system call.
semnum	Stores and passes the semaphore number to the system call.

<code>cmd</code>	Stores the code for the desired flag so that subsequent processing can be performed on it.
<code>choice</code>	Determines which member (<code>uid</code> , <code>gid</code> , <code>mode</code>) of the <code>IPC_SET</code> flag should be changed.
<code>arg.val</code>	Passes the system call a value to set (<code>SETVAL</code>) or stores (<code>GETVAL</code>) a value returned from the system call for a single semaphore (union member).
<code>arg.buf</code>	A pointer passed to the system call which locates the data structure in the user memory area where the <code>IPC_STAT</code> flag is to place its return values, or where the <code>IPC_SET</code> command gets the values to set (union member).
<code>arg.array</code>	Stores the set of semaphore values when getting (<code>GETALL</code>) or initializing (<code>SETALL</code>) (union member).

12

Note that the `semid_ds` data structure in this program (line 14) uses the data structure located in the `<sem.h>` include file of the same name as a template for its declaration.

The `arg` union (lines 18-22) serves three purposes in one:

- The compiler allocates enough storage to hold the program's largest member. Note that the array is declared to have 25 elements (0 through 24). This number corresponds to the maximum number of semaphores allowed per set (`SEMMSL`), a system-tunable parameter.
- The `buf` pointer member (`arg.buf`) of the union is declared to be a pointer to a data structure of the `semid_ds` type, it must also be initialized to contain the address of the user memory area data structure (line 24).
- The pointer does not need to be reinitialized later. If it was used to increment through the array, it would need to be reinitialized just before calling the system call.

C Library Guide

How semctl Works

First, the program prompts for a valid semaphore set identifier, which is stored at the address of the **semid** variable (lines 25-27). This is required for all **semctl(S)** system calls.

Then, the code for the desired flag must be entered (lines 28-42), and the code is stored at the address of the **cmd** variable. The code is tested to determine the flag for subsequent processing. The flags and code descriptions are described as follows:

Flag	Description
GETVAL	(code 1) A message prompting for a semaphore number is displayed (lines 49, 50). When it is entered, it is stored at the address of the semnum variable (line 51). Then, the system call is performed, and the semaphore value is displayed (lines 52-55). If the system call is successful, a message indicates this along with the semaphore set identifier used (lines 195, 196); if the system call is unsuccessful, an error message is displayed along with the value of the external errno variable (lines 191-193).
SETVAL	(code 2) A message prompting for a semaphore number is displayed (lines 56, 57). When it is entered, it is stored at the address of the semnum variable (line 58). Next, a message prompts for the value the semaphore is to be set to, and it is stored as the arg.val member of the union (lines 59, 60). Then, the system call is performed (lines 61, 63). Depending upon success or failure, the program returns the same messages as for GETVAL above.
GETPID	(code 3) The system call is made immediately since all required arguments are known (lines 64-67) and the PID of the process performing the last operation is displayed. Depending upon success or failure, the program returns the same messages as for GETVAL above.
GETNCNT	(code 4) A message prompting for a semaphore number is displayed (lines 68-72). When entered, it is stored at the address of the semnum variable (line 73). Then, the system call is performed, and the number of processes waiting for the semaphore to become greater than its current value is displayed (lines 74-77).

Depending upon success or failure, the program returns the same messages as GETVAL.

- GETZCNT** (code 5) A message prompting for a semaphore number is displayed (lines 78-81). When it is entered, it is stored at the address of the **semnum** variable (line 82). Then the system call is performed and the number of processes waiting for the semaphore value to become equal to zero is displayed (lines 83, 86). Depending upon success or failure, the program returns the same messages as GETVAL.
- GETALL** (code 6) The program first performs an **IPC_STAT** control command to determine the number of semaphores in the set (lines 88-93). The length variable is set to the number of semaphores in the set (line 91). Next, the system call is made and, upon success, the **arg.array** union member contains the values of the semaphore set (line 96). Now, a loop is entered which displays each element of the **arg.array** from zero to one less than the value of length (lines 97-103). The semaphores in the set are displayed on a single line, separated by a space. Depending upon success or failure, the program returns the same messages as GETVAL.
- SETALL** (code 7) The program first performs an **IPC_STAT** control command to determine the number of semaphores in the set (lines 106-108). The length variable is set to the number of semaphores in the set (line 109). Next, the program prompts for the values to be set and enters a loop which takes values from the keyboard and initializes the **arg.array** union member to contain the desired values of the semaphore set (lines 113-119). The loop puts the first entry into the array position for semaphore number zero and ends when the semaphore number that is filled in the array equals one less than the value of length. The system call is then made (lines 120-122). Depending upon success or failure, the program returns the same messages as GETVAL.
- IPC_STAT** (code 8) The system call is performed (line 127), and the status information returned is printed out (lines 128-139); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful, the status information of the last successful one is printed out. In addition, an error message is

C Library Guide

displayed, and the **errno** variable is printed out (lines 191, 192).

IPC_SET (code 9) The program gets the current status information for the semaphore set identifier specified (lines 143-146). This is necessary because this example program provides for changing only one member at a time, and the **semctl(S)** system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this flag until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 147-153). This code is stored at the address of the "choice" variable (line 154). Now, depending upon the member picked, the program prompts for the new value (lines 155-178). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (line 181). Depending upon success or failure, the program returns the same messages as **GETVAL**.

IPC_RMID (code 10) The system call is performed (lines 183-185). The **semid** along with its associated data structure and semaphore set is removed from the UNIX operating system. Depending upon success or failure, the program returns the same messages as for the other control commands.

The program example for the **semctl(S)** system call follows. In the example the source program file is named *semctl.c* and the executable file is named *semctl*.

Example

```

1  /*This is a program to illustrate
2  **the semaphore control, semctl(),
3  **system call capabilities.
4  */

5  /*Include necessary include files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/sem.h>

10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     struct semid_ds semid_ds;
15     int c, i, length;
16     int uid, gid, mode;
17     int retrn, semid, semnum, cmd, choice;
18     union semun {
19         int val;
20         struct semid_ds *buf;
21         ushort array[25];
22     } arg;

23     /*Initialize the data structure pointer.*/
24     arg.buf = &semid_ds;

25     /*Enter the semaphore ID.*/
26     printf("Enter the semid = ");
27     scanf("%d", &semid);

28     /*Choose the desired command.*/
29     printf("\nEnter the number for\n");
30     printf("the desired cmd:\n");
31     printf("GETVAL      = 1\n");
32     printf("SETVAL      = 2\n");
33     printf("GETPID      = 3\n");
34     printf("GETNCNT     = 4\n");
35     printf("GETZCNT     = 5\n");
36     printf("GETALL      = 6\n");
37     printf("SETALL      = 7\n");
38     printf("IPC_STAT    = 8\n");
39     printf("IPC_SET     = 9\n");
40     printf("IPC_RMID    = 10\n");
41     printf("Entry      = ");
42     scanf("%d", &cmd);

43     /*Check entries.*/
44     printf("\nsemid =%d, cmd = %d\n\n",
45           semid, cmd);

```

C Library Guide

Example (cont.)

```
46     /*Set the command and do the call.*/
47     switch (cmd)
48     {
49     case 1: /*Get a specified value.*/
50         printf("\nEnter the semnum = ");
51         scanf("%d", &semnum);
52         /*Do the system call.*/
53         retrn = semctl(semid, semnum, GETVAL, 0);
54         printf("\nThe semval = %d\n", retrn);
55         break;
56     case 2: /*Set a specified value.*/
57         printf("\nEnter the semnum = ");
58         scanf("%d", &semnum);
59         printf("\nEnter the value = ");
60         scanf("%d", &arg.val);
61         /*Do the system call.*/
62         retrn = semctl(semid, semnum, SETVAL, arg.val);
63         break;
64     case 3: /*Get the process ID.*/
65         retrn = semctl(semid, 0, GETPID, 0);
66         printf("\nThe sempid = %d\n", retrn);
67         break;
68     case 4: /*Get the number of processes
69             waiting for the semaphore to
70             become greater than its current
71             value.*/
72         printf("\nEnter the semnum = ");
73         scanf("%d", &semnum);
74         /*Do the system call.*/
75         retrn = semctl(semid, semnum, GETNCNT, 0);
76         printf("\nThe semncnt = %d", retrn);
77         break;
78     case 5: /*Get the number of processes
79             waiting for the semaphore
80             value to become zero.*/
81         printf("\nEnter the semnum = ");
82         scanf("%d", &semnum);
83         /*Do the system call.*/
84         retrn = semctl(semid, semnum, GETZCNT, 0);
85         printf("\nThe semzcnt = %d", retrn);
86         break;
87     case 6: /*Get all of the semaphores.*/
88         /*Get the number of semaphores in
89         the semaphore set.*/
90         retrn = semctl(semid, 0, IPC_STAT, arg.buf);
91         length = arg.buf->sem_nsems;
92         if(retrn == -1)
93             goto ERROR;
94         /*Get and print all semaphores in the
95         specified set.*/
96         retrn = semctl(semid, 0, GETALL, arg.array);
97         for (i = 0; i < length; i++)
```

Example (cont.)

```

98     {
99         printf("%d", arg.array[i]);
100        /*Separate each
101        semaphore.*/
102        printf("%c", ' ');
103    }
104    break;

105    case 7: /*Set all semaphores in the set.*/
106        /*Get the number of semaphores in
107        the set.*/
108        retrn = semctl(semid, 0, IPC_STAT, arg.buf);
109        length = arg.buf->sem_nsems;
110        printf("Length = %d\n", length);
111        if(retrn == -1)
112            goto ERROR;
113        /*Set the semaphore set values.*/
114        printf("\nEnter each value:\n");
115        for(i = 0; i < length ; i++)
116            {
117                scanf("%d", &c);
118                arg.array[i] = c;
119            }
120        /*Do the system call.*/
121        retrn = semctl(semid, 0, SETALL, arg.array);
122        break;

123    case 8: /*Get the status for the semaphore set.*/
124        /*Get and print the current status values.*/
125        retrn = semctl(semid, 0, IPC_STAT, arg.buf);
126        printf ("\nThe USER ID = %d\n",
127                arg.buf->sem_perm.uid);
128        printf ("The GROUP ID = %d\n",
129                arg.buf->sem_perm.gid);
130        printf ("The operation permissions = 0%o\n",
131                arg.buf->sem_perm.mode);
132        printf ("The number of semaphores in set = %d\n",
133                arg.buf->sem_nsems);
134        printf ("The last semop time = %d\n",
135                arg.buf->sem_otime);
136
137        printf ("The last change time = %d\n",
138                arg.buf->sem_ctime);
139        break;

140

141    case 9: /*Select and change the desired
142            member of the data structure.*/
143        /*Get the current status values.*/
144        retrn = semctl(semid, 0, IPC_STAT, arg.buf);
145        if(retrn == -1)
146            goto ERROR;
147        /*Select the member to change.*/
148        printf("\nEnter the number for the\n");
149        printf("member to be changed:\n");
150        printf("sem_perm.uid = 1\n");
151        printf("sem_perm.gid = 2\n");

```

C Library Guide

Example (cont.)

```
152     printf("sem_perm.mode = 3\n");
153     printf("Entry      = ");
154     scanf("%d", &choice);
155     switch(choice) {
156     case 1: /*Change the user ID.*/
157         printf("\nEnter USER ID = ");
158         scanf ("%d", &uid);
159         arg.buf->sem_perm.uid = uid;
160         printf("\nUSER ID = %d\n",
161             arg.buf->sem_perm.uid);
162         break;
163
164     case 2: /*Change the group ID.*/
165         printf("\nEnter GROUP ID = ");
166         scanf ("%d", &gid);
167         arg.buf->sem_perm.gid = gid;
168         printf("\nGROUP ID = %d\n",
169             arg.buf->sem_perm.gid);
170         break;
171
172     case 3: /*Change the mode portion of
173             the operation
174             permissions.*/
175         printf("\nEnter MODE = ");
176         scanf ("%o", &mode);
177         arg.buf->sem_perm.mode = mode;
178         printf("\nMODE = 0%o\n",
179             arg.buf->sem_perm.mode);
180         break;
181     }
182     /*Do the change.*/
183     retm = semctl(semid, 0, IPC_SET, arg.buf);
184     break;
185
186 case 10: /*Remove the semid along with its
187         data structure.*/
188     retm = semctl(semid, 0, IPC_RMID, 0);
189 }
190 /*Perform the following if the call is unsuccessful.*/
191 if(retm == -1)
192 {
193     ERROR:
194     printf ("\n\nThe semctl system call failed!\n");
195     printf ("The error number = %d\n", errno);
196     exit (0);
197 }
198 printf ("\n\nThe semctl system call was successful\n");
199 printf ("for semid = %d\n", semid);
200 exit (0);
201 }
```

This section contains a detailed description of using the **semop(S)** system call along with an example program that exercises all of its capabilities.

The syntax found in **semop(S)** is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf **sops;
unsigned nsops;
```

The **semop(S)** system call requires three arguments to be passed to it, and it returns an integer value. Semop returns a zero value upon successful completion, if unsuccessful it returns a -1.

Argument	Description
semid	Must be a valid, non-negative, integer value. In other words, it must have already been created by using the semget(S) system call.
sops	A pointer to an array of structures in the user memory area that contains the following for each semaphore to be changed: <ul style="list-style-type: none"> - the semaphore number - the operation to be performed - the flag (flags)

The ****sops** declaration means that a pointer can be initialized to the address of the array, or the array name can be used since it is the address of the first element of the array. **Sembuf** is the *tag* name of the data structure used as the template for the structure members in the array; it is located in the `<sys/sem.h>` include file.

nsops	Specifies the length of the array (the number of structures in the array). The maximum size of this array is determined by the SEMOPM system-tunable parameter. Therefore, a maximum of SEMOPM operations can be performed for each semop(S) system call.
-------	--

C Library Guide

The semaphore number determines the particular semaphore (within the set) on which the operation is to be performed.

The following values determine the operation to be performed:

- A positive integer value means to increment the semaphore value by the value of the positive integer.
- A negative integer value means to decrement the semaphore value by the absolute value of the negative integer.
- A value of zero means to test if the semaphore value is equal to zero.

The following flags can be used:

Flag	Description
IPC_NOWAIT	Can be set for any operations in the array. The system call will return unsuccessfully without changing any semaphore values at all if any operation for which IPC_NOWAIT is set cannot be performed successfully. The system call will be unsuccessful when trying to decrement a semaphore more than its current value, or when testing for a semaphore to be equal to zero when it is not.
SEM_UNDO	Will undo any operations in the array to be undone. Undoing is accomplished by using an array of adjust values for the operations that are to be undone. This is especially useful for resetting semaphore values at process exit time using <code>exit(S)</code> .

12.10.4 Example Program Using semop

The program example in this section is a menu driven program that exercises all possible combinations of the `semop(S)` system call.

This program presents the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required include files as specified by `shmop(S)`. Note that in this program `errno` is declared as an external variable, and therefore, the `<errno.h>` include file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the syntax. These names make the program more readable and this is legal since the declarations are local to the program. The declared variables are described as follows:

Variable	Description
<code>sembuf[10]</code>	An array buffer (line 14) containing a maximum of ten sembuf type structures; SEMOPM , the maximum number of operations on a semaphore set for each semop(S) system call.
<code>sops</code>	A pointer (line 14) to sembuf[10] for the system call and for accessing the structure members within the array.
<code>retrn</code>	Stores the return values from the system call.
<code>flags</code>	Stores the code of the IPC_NOWAIT or SEM_UNDO flags for the semop(S) system call (line 60).
<code>i</code>	A counter (line 32) for initializing the structure members in the array; prints out each structure in the array (line 79).
<code>nsops</code>	Specifies the number of semaphore operations for the system call; must be less than or equal to SEMOPM .
<code>semid</code>	Stores the desired semaphore set identifier for the system call.

First, the program prompts for a semaphore set identifier that the system call is to perform operations on (lines 19-22). The semaphore identifier is stored at the address of the **semid** variable (line 23).

A message is displayed requesting the number of operations to be performed on this set (lines 25-27). The number of operations is stored at the address of the **nsops** variable (line 28).

Next, a loop is entered to initialize the array of structures (lines 30-77). The semaphore number, operation, and operation command (flags) are entered for each structure in the array. The number of structures equals the number of semaphore operations (**nsops**) to be performed for the system call, so **nsops** is tested against the **i** counter for loop control. Note

C Library Guide

that **sops** is used as a pointer to each element (structure) in the array, and **sops** is incremented just like **i**.

After the array is initialized, all of its elements are printed out for feedback (lines 78-85).

The **sops** pointer is set to the address of the array (lines 86, 87). **Sembuf** could be used directly, if desired, instead of **sops** in the system call.

The system call is made (line 89), and depending upon success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using the **semctl** GETALL flag.

The example program for the **semop(S)** system call follows. In the example the source program file is named *semop.c* and the executable file is named *semop*.

Example

```
1  /*This is a program to illustrate
2  **the semaphore operations, semop(),
3  **system call capabilities.
4  */
5
6  /*Include necessary include files.*/
7  #include <stdio.h>
8  #include <sys/types.h>
9  #include <sys/ipc.h>
10 #include <sys/sem.h>
11 /*Start of main C language program*/
12 main()
13 {
14     extern int errno;
15     struct sembuf sembuf[10], *sops;
16     char string[];
17     int retrn, flags, sem_num, i, semid;
18     unsigned nsops;
19     sops = sembuf; /*Pointer to array sembuf.*/
20
21     /*Enter the semaphore ID.*/
22     printf("\nEnter the semid of\n");
23     printf("the semaphore set to\n");
24     printf("be operated on = ");
25     scanf("%d", &semid);
26     printf("\nsemid = %d", semid);
27
28     /*Enter the number of operations.*/
29     printf("\nEnter the number of semaphore\n");
30     printf("operations for this set = ");
31     scanf("%d", &nsops);
32     printf("\nnosops = %d", nsops);
```

Example (cont.)

```

30      /*Initialize the array for the
31      number of operations to be performed.*/
32      for(i = 0; i < nsops; i++, sops++)
33      {
34          /*This determines the semaphore in
35          the semaphore set.*/
36          printf("\nEnter the semaphore\n");
37          printf("number (sem_num) = ");
38          scanf("%d", &sem_num);
39          sops->sem_num = sem_num;
40          printf("\nThe sem_num = %d", sops->sem_num);

41          /*Enter a (-)number to decrement,
42          an unsigned number (no +) to increment,
43          or zero to test for zero. These values
44          are entered as a string and converted
45          to integer values.*/
46          printf("\nEnter the operation for\n");
47          printf("the semaphore (sem_op) = ");
48          scanf("%s", string);
49          sops->sem_op = atoi(string);
50          printf("\nsem_op = %d\n", sops->sem_op);

51          /*Specify the desired flags.*/
52          printf("\nEnter the corresponding\n");
53          printf("number for the desired\n");
54          printf("flags:\n");
55          printf("No flags                = 0\n");
56          printf("IPC_NOWAIT                    = 1\n");
57          printf("SEM_UNDO                        = 2\n");
58          printf("IPC_NOWAIT and SEM_UNDO        = 3\n");
59          printf("          Flags                = ");
60          scanf("%d", &flags);

61          switch(flags)
62          {
63          case 0:
64              sops->sem_flg = 0;
65              break;
66          case 1:
67              sops->sem_flg = IPC_NOWAIT;
68              break;
69          case 2:
70              sops->sem_flg = SEM_UNDO;
71              break;
72          case 3:
73              sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
74              break;
75          }
76          printf("\nFlags = 0%o\n", sops->sem_flg);
77      }

```

C Library Guide

Example (cont.)

```
78      /*Print out each structure in the array.*/
79      for(i = 0; i < nsops; i++)
80      {
81          printf("\nsem_num = %d\n", sembuf[i].sem_num);
82          printf("sem_op = %d\n", sembuf[i].sem_op);
83          printf("sem_flg = %o\n", sembuf[i].sem_flg);
84          printf("%c", ' ');
85      }

86      sops = sembuf; /*Reset the pointer to
87                    sembuf[0].*/

88      /*Do the semop system call.*/
89      retrn = semop(semid, sops, nsops);
90      if(retrn == -1) {
91          printf("\nSemop failed. ");
92          printf("Error = %d\n", errno);
93      }
94      else {
95          printf ("\nSemop was successful\n");
96          printf("for semid = %d\n", semid);

97          printf("Value returned = %d\n", retrn);
98      }
99  }
```

12.11 Overview of Shared Memory

Shared memory is a method in which one process shares its allocated data space with another. Shared memory lets processes pool information in a central location and directly access that information without the burden of creating pipes or temporary files.

The standard C library provides several functions to create, add, access, signal, and free shared-memory segments. XENIX System V supports two sets of system calls for dealing with shared-memory operations. These are referred to in subsequent descriptions as either *XENIX shared memory* or *UNIX System V shared memory*.

The XENIX shared-memory operations are compatible with previous releases of XENIX. The system calls for manipulating XENIX shared memory are: **sdget**, **sdfree**, **sdenter**, **sdleave**, **sdgetv**, and **sdwait**.

The system calls for manipulating UNIX shared memory are: **shmat**, **shmdt**, **shmctl**, and **shmget**.

XENIX and UNIX shared memory are not compatible. The operations mentioned are valid only for one type of shared memory and cannot be mixed with the other type. This section describes the use of the XENIX shared-memory operations since these system calls are unique to the XENIX operating system. For more information on UNIX shared-memory operations, see **intro(S)**, **shmctl(S)**, and **shmget(S)** in the *XENIX Programmer's Reference*.



12.12 Using Shared Memory

The XENIX shared memory functions are described in this section. They are described as follows:

<code>sdget</code>	Creates a shared memory segment for the current process and attaches the segment to the process's data space. Can also be used to attach an existing shared-memory segment to a process's data space.
<code>sdenter</code>	Signals a process's intention to access the contents of a shared memory segment.
<code>sdleave</code>	Signals a process's intention to leave a shared memory segment after reading or modifying its contents.
<code>sdgetv</code>	Returns the current version number of the given memory segment.
<code>sdwaitv</code>	Causes a process to wait until the version number for the given segment is no longer equal to a given version number.
<code>sdfree</code>	Detaches the current process from the given shared memory segment.

To use the shared data functions, you must put the following line at the beginning of the program:

```
#include <sys/sd.h>
```

This *sd.h* file contains definitions for the manifest constants and other macros used by the functions.

C Library Guide

12.12.1 Creating a Shared Memory Segment

The **sdget** function creates a shared memory segment for the current process and attaches the segment to the process's data space. The function call has the following form:

```
sdget (path, flags [, size, mode] )
```

where

- *path* is a character pointer to a valid pathname.
- *flags* is a long integer value which defines how the segment should be created.
- *size* is an integer value which defines the size in bytes of the segment to be created.
- *mode* is an integer value which defines the access permissions to be given to the segment.

The *flag* can be a combination of SD_CREAT for creating the segment, and SD_RDONLY for attaching the segment for reading only or SD_WRITE for attaching the segment for reading and writing. You can also use SD_UNLOCK for allowing simultaneous access to the shared segment by multiple processes. The values can be combined by a logical OR. The function returns the address of the segment if it has been successfully created. Otherwise, the function returns -1.

A process uses this function to create a segment that it will share with several other processes. For example, in the following program fragment, **sdget** creates a segment and attaches it for reading and writing. The address of the new segment is assigned to *shared* as follows:

```
#include <sys/sd.h>

main ()
{
    char *shared;

    shared = sdget ( "/tmp/share", SD_CREAT|SD_WRITE, 512L, 0777 );
}
```

When the segment is created, the size “512L” and the mode “0777” are used to define the segment's size in bytes and access permissions. Access permissions are similar to permissions given to regular files. A segment may have read or write permission for the owner of the process, for users belonging to the same group as the owner, and for all other users.

Execute permission for a segment has no meaning. For example, the mode “0777” means read and write permission for everyone, but “0660” means read and write permissions for the owner and group processes only. When first created, a segment is filled with zeroes.

Note

The SD_UNLOCK flag used on systems without hardware support for shared data may severely degrade the execution performance of the program.

12

12.12.2 Attaching a Shared Memory Segment

The `sdget` function can also be used to attach an existing shared memory segment to a process’s data space. In this case, the function call has the following form:

`sdget(path, flags)`

where

- *path* is a character pointer to the pathname of a shared memory segment created by some other process.
- *flags* is an integer value which defines how the segment should be attached.

The *flag* may be SD_RDONLY for attaching the segment for reading only, or SD_WRITE for attaching the segment for reading and writing. If the function is successful, it returns the address of the new segment. Otherwise, it returns -1.

You use this function to attach any shared memory segment a process may wish to access. For example, in the following fragment, the program

C Library Guide

uses **sdget** to attach the segments associated with the files */tmp/share1* and */tmp/share2*, for reading and writing. The addresses of the new segments are assigned to the pointer variables, *share1* and *share2*, as follows:

```
#include <sys/sd.h>

main ()
{
    char *share1, *share2;

    share1 = sdget( "/tmp/share1", SD_WRITE );
    share2 = sdget( "/tmp/share2", SD_WRITE );

}
```

sdget returns an error value to any process that attempts to access a shared memory segment without the necessary permissions. You define the segment permissions when you create the segment.

12.12.3 Entering a Shared Memory Segment

The **sdenter** function signals a process's intention to access the contents of a shared memory segment. A process should not access the contents of the segment unless it enters the segment. The function call has the following form:

sdenter (*addr*, *flags*)

where

- *addr* is a character pointer to the segment to be accessed.
- *flag* is an integer value which defines how the segment is to be accessed.

The *flag* may be **SD_RDONLY** for indicating read only access to the segment, **SD_WRITE** for indicating write access to the segment, or **SD_NOWAIT** for returning an error if the segment is locked and another process is currently accessing it. These values can also be combined by a logical OR. The function normally waits for the segment to become available before allowing access to it. A segment is not available if the segment has been created without an **SD_UNLOCK** flag and another process is currently accessing it.

Once a process enters a segment, it can examine and modify the contents of the segment depending upon the read and write permissions established with **sdenter**. For example, in the following program fragment, **sdenter** enters the segment for reading and writing, then sets the first value in the segment to 0 if it is equal to 255:

```
#include <sys/sd.h>

main ()
{
    char *share;

    share = sdget( "/tmp/share", SD_WRITE );

    sdenter(share, SD_WRITE);
        if ( share[0] == 255 )
            share[0] = 0;
        .
        .
        .
}

```

12

In general, it is unwise to stay in a shared memory segment any longer than it takes to examine or modify the desired location. The **sdleave** function should be used after each access. When in a shared memory segment, a program should avoid using system functions. System functions can disrupt the normal operations required to support shared data and may cause some data to be lost. In particular, if a program creates a shared memory segment that cannot be shared simultaneously, the program must not call the **fork** function when it is also accessing that segment.

12.12.4 Leaving a Shared Memory Segment

The **sdleave** function signals a process's intention to leave a shared memory segment after reading or modifying its contents. The function call has the following form:

sdleave (*addr*)

where *addr* is a **char** pointer to the desired segment. The function returns -1 if it encounters an error, otherwise it returns 0. The return value is always an integer.

You should use this function after each access of the shared memory to terminate the access. If the segment's lock flag is set (segment not created with **SD_UNLOCK**), you must use the function after each access

C Library Guide

to let other processes access the segment. For example, in the following program fragment, **sdleave** terminates each access to the segment given by *share*.

```
#include <sys/sd.h>

main ()
{
  int i = 0;
  char c, *share;

  share = sdget("/tmp/share", SD_RDONLY);

  sdenter(share, SD_RDONLY);
  c = *share;
  sdleave(share);

  while (c!=0) {
    putchar(c);
    i++;
    sdenter(share, SD_RDONLY);
    c = share[i];
    sdleave(share);
  }
}
```

12.12.5 Getting the Current Version Number

The **sdgetv** function returns the current version number of the given memory segment. The function call has the following form:

sdgetv (*addr*)

where *addr* is a character pointer to the desired segment. A segment's version number is initially zero, but it is incremented by one whenever a process leaves the segment using the **sdleave** function. Thus, the version number is a record of the number of times the segment has been accessed. The function's return value is always an integer. It returns -1 if it encounters an error.

You use this function to choose an action based on the current version number of the segment. For example, in the following program fragment, **sdgetv** determines whether or not **sdenter** should be used to enter the segment given by *shared*:

```
#include <sys/sd.h>

main ()
{
  char *shared;

  if (sdgetv(shared) > 10)
    sdenter(shared, SD_RDONLY);
```

12

In this example, the segment is entered if the current version number of the segment is greater than 10.

12.12.6 Waiting for a Version Number

The **sdwaitv** function causes a process to wait until the version number for the given segment is no longer equal to a given version number. The function call has the following form:

sdwaitv (*addr*, *vnum*)

where

- *addr* is a character pointer to the desired segment
- *vnum* is an integer value which defines the version number to wait on

The function normally returns the new version number. It returns -1 if it encounters an error. The return value is always an integer.

You use this function to synchronize the actions of two separate processes. For example, in the following program fragment, **sdwait** waits

C Library Guide

while the process corresponding to the version number, *vnum*, performs its operations in the segment:

```
#include <sys/sd.h>

main ()
{
  char *share;
  int change;

  vnum = sdgetv( share );
  i=0;
  if ( sdwaitv( share, vnum ) == -1 )
    fprintf(stderr, "Cannot find segment\n");
  else
    sdenter(share, SD_RDONLY);
```

If an error occurs while the program is waiting, an error message is printed.

12.12.7 Freeing a Shared Memory Segment

The **sdfree** function detaches the current process from the given shared memory segment. The function call has the following form:

sdfree (*addr*)

where *addr* is a character pointer to the segment to be freed. The function returns the integer value 0, if the segment is freed. Otherwise, it returns -1.

If the process is currently accessing the segment, **sdfree** automatically calls **sdleave** to leave the segment before freeing it.

The contents of segments that have been freed by all attached processes are destroyed. To reaccess the segment, a process must recreate it using the **sdget** function and the SD_CREAT flag.

12.12.8 Program Example

This section shows how to use the shared memory functions to share a single memory segment between two processes. The following program attaches a memory segment named */tmp/share* and then uses it to transfer information to and from the child and parent processes.

Example (cont.)

```

#include <sys/sd.h>

main()
{
    char *share, message[12];
    int i, vnum;

    share = sdget("/tmp/share",SD_CREAT|SD_WRITE, 512L, 0777);

    if (fork()==0) {
        for (i=0; i<4; i++) {
            sdenter(share, SD_WRITE);
            strncpy(message, share, 12);
            strncpy(share,"Shared data", 12);
            vnum=sdgetv(share);
            sdleave(share);
            sdwaitv(share, vnum+1);
            printf("Child: %d - %s\n", i, message);
        }
        sdenter(share, SD_WRITE);
        strncpy(message, share, 12);
        strncpy(share,"Shared data", 12);
        sdleave(share);
        printf("Child: %d - %s\n", i, message);
        exit(0);
    }

    for (i=0; i<5; i++) {
        sdenter(share, SD_WRITE);
        strncpy(message, share, 12);
        strncpy(share,"Data shared", 12);
        vnum=sdgetv(share);
        sdleave(share);
        sdwaitv(share, vnum+1);
        printf("Parent: %d - %s\n", i, message);
    }

    sdfree(share);
}

```

In this program, the child process inherits the memory segment created by the parent process. Each process accesses the segment five times. During the access, a process copies the current contents of the segment to the *message* variable and replaces the message with one of its own. It then displays *message* and continues the loop.

To synchronize access to the segment, both the parent and child use the **sdgetv** and **sdwaitv** functions. While a process still has control of the segment, it uses **sdgetv** to assign the current version number to the variable, *vnum*. It then uses this number in a call to **sdwaitv** to force itself to wait until the other process has accessed the segment. Note that the

C Library Guide

argument to **sdwaitv** is actually *vnum+1*. Since *vnum* was assigned before the **sdleave** call, it is exactly one less than the version number after the **sdleave** call. It is assigned before the **sdleave** call to ensure that the other process does not modify the current version number before the current process has a chance to assign it to *vnum*.

The last time the child process accesses the segment, it displays the message and exits without calling the **sdwaitv** function. This is to prevent the process from waiting forever, since the parent has already exited and can no longer modify the current version number.

12.13 Using Shared Memory Under UNIX System V

This section describes the system calls for the shared memory type of Inter-Process Communication (IPC). The shared memory type of IPC lets two or more processes (executing programs) share memory, and consequently, the data contained there. This is done by letting processes set up access to a common memory address space. This sharing occurs on a segment basis, which is memory management hardware dependent.

This sharing of memory provides fast means of exchanging data between processes.

A process initially creates a shared memory segment facility using the **shmget(S)** system call. Upon creation, this process sets the overall operation permissions for the shared memory segment facility, sets its size in bytes, and can specify that the shared memory segment is for reference only (read-only) upon attachment. If the memory segment is not specified to be for reference only, all other processes with appropriate operation permissions can read from or write to the memory segment.

The following two operations can be performed on a shared memory segment:

Operation	Description
shmat(S)	<i>(shared memory attach)</i> Lets processes associate themselves with the shared memory segment if they have permission. They can then read or write as allowed.
shmdt(S)	<i>(shared memory detach)</i> Lets processes disassociate themselves from a shared memory segment. Therefore, they lose the ability to read from or write to the shared memory segment.

The original owner/creator of a shared memory segment can relinquish ownership to another process using the **shmctl(S)** system call. However, the creating process remains the creator until the facility is removed or the system is reinitialized. Other processes with permission can perform other control functions on the shared memory segment using the **shmctl(S)** system call.

For more information on shared memory system calls, see **shmget(S)**, **shmctl(S)**, **shmat(S)**, and **shmdt(S)** in the *XENIX Programmer's Reference*. They make shared memory capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

12

12.14 Shared Memory Data Structures

The sharing of memory between processes occurs on a virtual segment basis. There is one and only one instance of an individual shared memory segment existing in the operating system at any point in time.

Before sharing of memory can be realized, a uniquely identified shared-memory segment and data structure must be created. The unique identifier created is called **shmid** (shared memory identifier); it is used to identify or reference the associated data structure. The data structure includes the following for each shared memory segment:

- operation permissions
- segment size
- segment descriptor
- process identification of the process performing the last operation
- process identification of creator
- current number of processes attached
- in memory number of processes attached
- last attach time

C Library Guide

- last detach time
- last change time

The C programming language data structure definition for the shared memory segment data structure is located in the */usr/include/sys/shm.h* include file. It is as follows:

```
/*
**  There is a shared mem id data structure for
**  each segment in the system.
*/

struct shm_id {
    struct ipc_perm    shm_perm;    /* operation permission struct */
    int                shm_segsz;   /* segment size */
    ushort             shm_ptbl;    /* addr of ds segment */
    ushort             shm_lpid;    /* pid of last shmop */
    ushort             shm_cpid;    /* pid of creator */
    ushort             shm_nattch;  /* used only for shminfo */
    ushort             shm_cnattch; /* used only for shminfo */
    time_t             shm_atime;   /* last shmat time */
    time_t             shm_dtime;   /* last shmdt time */
    time_t             shm_ctime;   /* last change time */
};
```

Note that the **shm_perm** member of this structure uses **ipc_perm** as a template.

The **ipc_perm** data structure is the same for all IPC facilities, and it is located in the *<sys/ipc.h>* include file. It is shown in “Message Operations.”

The **shmget(S)** system call performs two tasks when only the **IPC_CREAT** flag is set in the **shmflg** argument that it receives:

- Gets a new **shm_id** and creates an associated shared memory segment data structure for it
- Returns an existing **shm_id** that already has an associated shared memory segment data structure

The task performed is determined by the value of the **key** argument passed to the **shmget(S)** system call. For the first task, if the **key** is not already in use for an existing **shm_id**, a new **shm_id** is returned with an associated shared memory segment data structure created for it provided no system-tunable parameters would be exceeded.

There is also a provision for specifying a **key** of value `IPC_PRIVATE` which is known as the private key; when specified, a new **shmid** is always returned with an associated shared memory segment data structure created for it unless a system-tunable parameter would be exceeded. When the **ipcs** command is performed, the **KEY** field for the **shmid** is all zeros.

For the second task, if a **shmid** exists for the **key** specified, the value of the existing **shmid** is returned. If it is not desired to have an existing **shmid** returned, a flag (`IPC_EXCL`) can be specified (set) in the **shmflg** argument passed to the system call. The details of using this system call are discussed in “Example Program Using `shmget`.”

When performing the first task, the process that calls **shmget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see Section 12.14.3, “Controlling Shared Memory.” The creator of the shared memory segment also determines the initial operation permissions for it.

Once a uniquely identified shared memory segment data structure is created, **shmop** (shared memory segment operations) and **shmctl(S)** (control operations) can be used.

Shared memory segment operations consist of attaching and detaching shared memory segments. System calls are provided for each of these operations; they are **shmat(S)** and **shmdt(S)**. For details of these system calls, see “Operations for Shared Memory.”

Shared memory segment control uses the **shmctl(S)** system call. It permits you to control the shared memory facility in the following ways:

- determines the associated data structure status for a shared memory segment (**shmid**)
- changes operation permissions for a shared memory segment
- removes a particular **shmid** from the operating system along with its associated shared memory segment data structure

For more information on the **shmctl(S)** system call, see “Controlling Shared Memory.”

C Library Guide

12.14.1 Getting Shared Memory Segments

This section describes the **shmget(S)** system call along with an example program illustrating its use. The syntax for **shmget(S)** is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key, size, shmflg)
key_t key;
unsigned int size;
int shmflg;
```

All of these include files are located in the */usr/include/sys* directory of the operating system. The following syntax line informs you that **shmget(S)** is a function with three formal arguments that returns an integer type value, upon successful completion (**shmid**):

```
int shmget (key, size, shmflg)
```

The following two lines declare the types of the formal arguments:

```
key_t key;
unsigned int size;
int shmflg;
```

The variable **key_t** is declared by a **typedef** in the **types.h** include file to be a long integer.

Upon successful completion, the integer returns **shmid**.

As declared, the process calling the **shmget(S)** system call must supply three arguments to be passed to the formal **key**, **size**, and **shmflg** arguments.

A new **shmid**, with an associated shared-memory data structure, is provided if either

- **key** is equal to **IPC_PRIVATE**
- or
- **key** is passed a unique long integer, and **shmflg** ANDed with **IPC_CREAT** is “true” (not zero).

The value passed to the **shmflg** argument must be an integer type value and will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes and execution modes determine the user/group/other attributes of the **shmflg** argument. They are collectively referred to as *operation permissions*. *Control fields* pass the value of a predefined constant to **shmflg**. The following list reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

12

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by the user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the **shm.h** header file that can be used for the user (OWNER). They are as follows:

```
SHM_R 0400
SHM_W 0200
```

The following list contains the names of the constants that apply to the **shmget** system call along with their values:

Flag	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

C Library Guide

The value for **shmflg** is, therefore, a combination of operation permissions and flags. After determining the value for the operation permissions, the desired flags can be specified. You can accomplish this by bitwise ORing (**|**) them with the operation permissions; the bit positions and values for the flags in relation to those of the operation permissions make this possible. It is illustrated as follows:

Name		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
ORed by User	=	0 0 4 0 0	0 000 000 100 000 000
shmflg	=	0 1 4 0 0	0 000 001 100 000 000

You can set the **shmflg** value by using the names of the flags in conjunction with the octal operation permissions value:

```
shmld = shmget (key, size, (IPC_CREAT | 0400));  
shmld = shmget (key, size, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by **shmget(S)**, success or failure of this system call depends upon the argument values for **key**, **size**, and **shmflg**, or on system-tunable parameters. The system call will attempt to return a new **shmld** if one of the following conditions is true:

- **key** is equal to **IPC_PRIVATE**
- **key** does not already have a **shmld** associated with it, and (**shmflg** & **IPC_CREAT**) is “true” (not zero)

The **key** argument can be set to **IPC_PRIVATE** in the following way:

```
shmld = shmget (IPC_PRIVATE, size, shmflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the **SHMMNI** system-tunable parameter always causes a failure. The **SHMMNI** system-tunable parameter determines the maximum number of unique shared memory segments (**shmlds**) in the operating system.

The second condition is satisfied if the value for **key** is not already associated with a **shmld** and the bitwise ANDing of **shmflg** and **IPC_CREAT** is “true” (not zero). This means that the **key** is unique (not in use) within the UNIX operating system for this facility type and that the **IPC_CREAT**

flag is set (**shmflg** | **IPC_CREAT**). The bitwise ANDing (&) is illustrated as follows:

```

      shmflg = x 1 x x x    (x = immaterial)
& IPC_CREAT = 0 1 0 0 0
      result = 0 1 0 0 0    (not zero)

```

Because the result is not zero, the flag is set or “true.” **SHMMNI** applies here also, just as for condition one.

IPC_EXCL is another flag used in conjunction with **IPC_CREAT** to have the system call fail if, and only if, a **shmid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **shmid** when it has not. In other words, when both **IPC_CREAT** and **IPC_EXCL** are specified, a unique **shmid** is returned if the system call is successful. Any value for **shmflg** returns a new **shmid** if the **key** equals **IPC_PRIVATE**.

The system call will fail if the value for the **size** argument is less than **SHMMIN** or greater than **SHMMAX**. These tunable parameters specify the minimum and maximum shared memory segment **sizes**.

For specific associated data structure initialization for successful completion, see **shmget(S)** in the *XENIX Programmer's Reference*. The specific failure conditions with error names are contained there also.

12.14.2 Example Program Using shmget

The example program in this section is a menu driven program that exercises all possible combinations the **shmget(S)** system call.

This program presents the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-7) by including the required include files as specified by **shmget(S)**. Note that the *<errno.h>* include file is included as opposed to declaring **errno** as an external variable; either method will work.

C Library Guide

Variable names have been chosen to be as close as possible to those in the syntax for the system call. These names make the program more readable, and this is legal since they are local to the program. The variables are described as follows:

Variable	Description
key	Passes the value for the desired key .
opperm	Stores the desired operation permissions.
flags	Stores the desired flags.
opperm_flags	Stores the combination from the logical ORing of the opperm and flags variables; it is then used in the system call to pass the shmflg argument.
shmid	Returns the shared memory segment identification number for a successful system call or the error code (-1) for an unsuccessful one
size	Specifies the shared memory segment size.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and finally for the flag combinations (**flags**) which are selected from a menu (lines 14-31). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 35-50).

A display then prompts for the **size** of the shared memory segment, and it is stored at the address of the **size** variable (lines 51-54).

The system call is made and the result is stored at the address of the **shmid** variable (line 56).

Since the **shmid** variable now contains a valid **shmid** or the error code (-1), it is tested to see if an error occurred (line 58). If **shmid** equals -1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 60, 61). If no error occurred, the returned shared memory segment identifier is displayed (line 65).

The example program for the **shmget(S)** system call follows. In the example the source program file is named *shmget.c* and the executable file is named *shmget*.

Example

```

1  /*This is a program to illustrate
2  **the shared memory get, shmget(),
3  **system call capabilities.*/

4  #include <sys/types.h>
5  #include <sys/ipc.h>
6  #include <sys/shm.h>
7  #include <errno.h>

8  /*Start of main C language program*/
9  main()
10 {
11     key_t key;           /*declare as long integer*/
12     int opperm, flags;
13     int shmid, size, opperm_flags;
14     /*Enter the desired key*/
15     printf("Enter the desired key in hex = ");
16     scanf("%lx", &key);

17     /*Enter the desired octal operation
18     permissions.*/
19     printf("\nEnter the operation\n");
20     printf("permissions in octal = ");
21     scanf("%o", &opperm);

22     /*Set the desired flags.*/
23     printf("\nEnter corresponding number to\n");
24     printf("set the desired flags:\n");
25     printf("No flags           = 0\n");
26     printf("IPC_CREAT           = 1\n");
27     printf("IPC_EXCL           = 2\n");
28     printf("IPC_CREAT and IPC_EXCL = 3\n");
29     printf("Flags           = ");
30     /*Get the flag(s) to be set.*/
31     scanf("%d", &flags);

32     /*Check the values.*/
33     printf("\nkey =0x%lx, opperm = 0%o, flags = 0%o\n",
34           key, opperm, flags);

35     /*Incorporate the control fields (flags) with
36     the operation permissions*/
37     switch (flags)
38     {
39     case 0: /*No flags are to be set.*/
40         opperm_flags = (opperm | 0);
41         break;
42     case 1: /*Set the IPC_CREAT flag.*/
43         opperm_flags = (opperm | IPC_CREAT);
44         break;
45     case 2: /*Set the IPC_EXCL flag.*/
46         opperm_flags = (opperm | IPC_EXCL);
47         break;

```

C Library Guide

Example (cont.)

```
48     case 3:    /*Set the IPC_CREAT and IPC_EXCL flags.*/
49         opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
50     }

51     /*Get the size of the segment in bytes.*/
52     printf ("\nEnter the segment");
53     printf ("\nsize in bytes = ");
54     scanf ("%d", &size);

55     /*Call the shmget system call.*/
56     shmid = shmget (key, size, opperm_flags);

57     /*Perform the following if the call is unsuccessful.*/
58     if(shmid == -1)
59     {
60         printf ("\nThe shmget system call failed!\n");
61         printf ("The error number = %d\n", errno);
62     }
63     /*Return the shmid upon successful completion.*/
64     else
65         printf ("\nThe shmid = %d\n", shmid);
66     exit(0);
67 }
```

12.14.3 Controlling Shared Memory

This section gives a detailed description the **shmctl(S)** system call along with an example program that exercises all of its capabilities. The syntax found in **shmctl(S)** is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmctl *buf;
```

The **shmctl(S)** system call requires three arguments to be passed to it, and **shmctl(S)** returns an integer value. Upon successful completion, **shmctl** returns a zero value; and if unsuccessful, **shmctl** returns a -1.

The **shmid** variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget(S)** system call.

The **cmd** argument can be replaced by one of following flags:

Flag	Description
IPC_STAT	Returns the status information contained in the associated data structure for the specified shmid and places it in the data structure pointed to by the buf pointer in the user memory area.
IPC_SET	For the specified shmid , sets the effective user and group identification, and operation permissions.
IPC_RMID	Removes the specified shmid along with its associated shared memory segment data structure.

12

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID flag. A process must have read permission to perform the IPC_STAT flag.

For more information, see “Example Program Using shmget.”

12.14.4 Example Program Using shmctl

The program example in this section is a menu driven program that exercises all possible combinations of **shmctl(S)**.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by **shmctl(S)**. Note in this program that **errno** is declared as an external variable, and therefore, the *<errno.h>* include file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables are described as follows:

Variable	Description
uid	Stores the IPC_SET value for the effective user identification.

C Library Guide

<code>gid</code>	Stores the <code>IPC_SET</code> value for the effective group identification.
<code>mode</code>	Stores the <code>IPC_SET</code> value for the operation permissions.
<code>retrn</code>	Stores the return integer value from the system call.
<code>shmid</code>	Stores and passes the shared memory segment identifier to the system call.
<code>command</code>	Stores the code for the desired flag so that subsequent processing can be performed on it.
<code>choice</code>	Determines which member for the <code>IPC_SET</code> flag that is to be changed.
<code>shmid_ds</code>	Receives the specified shared memory segment identifier's data structure when an <code>IPC_STAT</code> flag is performed.
<code>buf</code>	A pointer passed to the system call which locates the data structure in the user memory area where the <code>IPC_STAT</code> flag is to place its return values, or where the <code>IPC_SET</code> command gets the values to set.

Note that the `shmid_ds` data structure in this program (line 16) uses the data structure located in the `shm.h` include file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

Note also that although the `*buf` pointer is declared to be a pointer to a data structure of the `shmid_ds` type, it must also be initialized to contain the address of the user memory area data structure (line 17).

Now that all of the required declarations have been explained for this program, this is how it works:

First, the program prompts for a valid shared memory segment identifier which is stored at the address of the `shmid` variable (lines 18-20). This is required for every `shmctl(S)` system call.

Now you can enter the code for the desired flag (lines 21-29); this is stored at the address of the `command` variable. The code is tested to

determine the flag for subsequent processing. The flags and code descriptions are described as follows:

Flag	Code Description
IPC_STAT	(code 1) The system call is performed (lines 39, 40) and the status information returned is printed out (lines 41-71). Note that if the system call is unsuccessful (line 146), the status information of the last successful call is printed out. In addition, an error message is displayed and the errno variable is printed out (lines 148, 149). If the system call is successful, a message indicates this along with the shared memory segment identifier used (lines 151-154).
IPC_SET	(code 2) The first thing done is to get the current status information for the message queue identifier specified (lines 90-92). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user-memory area for one of these members, it would cause repetitive failures for this flag until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 93-98). This code is stored at the address of the <i>choice</i> variable (line 99). Now, depending upon the member picked, the program prompts for the new value (lines 105-127). The value is placed at the address of the appropriate member in the user-memory-area data structure, and the system call is made (lines 128-130). Depending upon success or failure, the program returns the same messages as for IPC_STAT above.
IPC_RMID	(code 3) The system call is performed (lines 132-135), and the shmid along with its associated message queue and data structure are removed from the operating system. Note that the buf pointer is not required as an argument to perform this control command and its value can be NULL. Depending upon the success or failure, the program returns the same messages as for the other control commands.

C Library Guide

The program example for the `shmctl(S)` system call follows. In the example, the source program file is named `shmctl.c` and the executable file is named `shmctl`.

Example

```
1  /*This is a program to illustrate
2  **the shared memory control, shmctl(),
3  **system call capabilities.
4  */

5  /*Include necessary include files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/shm.h>

10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     int uid, gid, mode;
15     int retn, shmid, command, choice;
16     struct shm_id_s shm_id_s, *buf;
17     buf = &shm_id_s;

18     /*Get the shmid, and command.*/
19     printf("Enter the shmid = ");
20     scanf("%d", &shmid);
21     printf("\nEnter the number for\n");
22     printf("the desired command:\n");

23     printf("IPC_STAT   = 1\n");
24     printf("IPC_SET     = 2\n");
25     printf("IPC_RMID    = 3\n");
26     printf("Entry       = ");
27     scanf("%d", &command);

28     /*Check the values.*/
29     printf ("\nshmid =%d, command = %d\n",
30            shmid, command);

31     switch (command)
32     {
33     case 1: /*Use shmctl() to duplicate
34             the data structure for
35             shmid in the shm_id_s area pointed
36             to by buf and then print it out.*/
37             retn = shmctl(shmid, IPC_STAT,
38                          buf);
39             printf ("\nThe USER ID = %d\n",
40                    buf->shm_perm.uid);
41             printf ("The GROUP ID = %d\n",
42                    buf->shm_perm.gid);
43     }
```

Example (cont.)

```

45     printf ("The creator's ID = %d\n",
46             buf->shm_perm.cuid);
47     printf ("The creator's group ID = %d\n",
48             buf->shm_perm.cgid);
49     printf ("The operation permissions = 0%o\n",
50             buf->shm_perm.mode);
51     printf ("The slot usage sequence\n");

52     printf ("number = 0%x\n",
53             buf->shm_perm.seq);
54     printf ("The key= 0%x\n",
55             buf->shm_perm.key);
56     printf ("The segment size = %d\n",
57             buf->shm_segsz);
58     printf ("The pid of last shmop = %d\n",
59             buf->shm_lpid);
60     printf ("The pid of creator = %d\n",
61             buf->shm_cpid);
62     printf ("The current # attached = %d\n",
63             buf->shm_nattch);
64     printf ("The in memory # attached = %d\n",
65             buf->shm_cnattach);
66     printf ("The last shmat time = %d\n",
67             buf->shm_atime);
68     printf ("The last shmdt time = %d\n",
69             buf->shm_dtime);
70     printf ("The last change time = %d\n",
71             buf->shm_ctime);
72     break;

/* Lines 73 - 87 deleted */

88     case 2: /*Select and change the desired
89             member(s) of the data structure.*/

90             /*Get the original data for this shmid
91              data structure first.*/
92             retrn = shmctl(shmid, IPC_STAT, buf);

93             printf("\nEnter the number for the\n");
94             printf("member to be changed:\n");
95             printf("shm_perm.uid   = 1\n");
96             printf("shm_perm.gid   = 2\n");
97             printf("shm_perm.mode  = 3\n");
98             printf("Entry       = ");
99             scanf("%d", &choice);
100            /*Only one choice is allowed per
101             pass as an illegal entry will
102             cause repetitive failures until
103             shmid ds is updated with
104             IPC_STAT.*/

```

C Library Guide

Example (cont.)

```
105         switch(choice) {
106         case 1:
107             printf("\nEnter USER ID = ");
108             scanf ("%d", &uid);
109             buf->shm_perm.uid = uid;
110             printf("\nUSER ID = %d\n",
111                 buf->shm_perm.uid);
112             break;
113         case 2:
114             printf("\nEnter GROUP ID = ");
115             scanf("%d", &gid);
116             buf->shm_perm.gid = gid;
117             printf("\nGROUP ID = %d\n",
118                 buf->shm_perm.gid);
119             break;
120
121         case 3:
122             printf("\nEnter MODE = ");
123             scanf("%o", &mode);
124             buf->shm_perm.mode = mode;
125             printf("\nMODE = 0%o\n",
126                 buf->shm_perm.mode);
127             break;
128         }
129         /*Do the change.*/
130         retrn = shmctl(shmid, IPC_SET,
131             buf);
132         break;
133
134     case 3:    /*Remove the shmid along with its
135               associated
136               data structure.*/
137         retrn = shmctl(shmid, IPC_RMID, (struct shm_id *)0 );
138         break;
139
140     }
141     /*Perform the following if the call is unsuccessful.*/
142     if(retrn == -1)
143     {
144         printf ("\n\rThe shmctl system call failed!\n");
145         printf ("The error number = %d\n", errno);
146     }
147     /*Return the shmid upon successful completion.*/
148     else
149         printf ("\nShmctl was successful for shmid = %d\n",
150             shmid);
151     exit (0);
152 }
```

12.14.5 Operations for Shared Memory

This section gives a detailed description of using the **shmat(S)** and **shmdt(S)** system calls, along with an example program that exercises all of their capabilities. The syntax for **shmop(S)** is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;

int shmdt (shmaddr)
char *shmaddr;
```



Attaching a Shared Memory Segment

The **shmat(S)** system call requires three arguments to be passed to it, and it returns a character pointer value.

Upon successful completion, **shmat** returns the address in core memory where the process is attached to the shared memory segment and when unsuccessful returns (char *)-1.

The **shmid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget(S)** system call.

The **shmaddr** argument can be NULL or user supplied when passed to the **shmat(S)** system call. If it is zero, the operating system picks the address of where the shared memory segment will be attached. If it is user supplied, the address must be a valid address that the operating system would pick. You can improve portability by letting the operating system pick addresses.

C Library Guide

The **shmflg** argument is used to pass the SHM_RND and SHM_RDONLY flags to the **shmat** system call.

For more information, see “Example Program Using shmget.”

Detaching Shared Memory Segments

The **shmdt(S)** system call requires one argument to be passed to it, and **shmdt(S)** returns an integer value.

Upon successful completion, **shmdt** returns a zero and when unsuccessful, **shmdt(S)** returns a -1.

For more information, see “Example Program Using shmget.”

12.14.6 Example Program Using shmop

The program example in this section is a menu driven program that exercises all possible combinations of **shmat(S)** and **shmdt(S)**. The program presents the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by **shmop(S)**. Note that in this program that **errno** is declared as an external variable, and therefore, the `<errno.h>` include file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the syntax. These names make the program more readable, and this is legal since they are local to the program. The variables are described as follows:

Variable	Description
flags	Stores the codes of SHM_RND or SHM_RDONLY for the shmat(S) system call.
addr	Stores the address of the shared memory segment for the shmat(S) and shmdt(S) system calls.
i	A loop counter for attaching and detaching.
attach	Stores the desired number of attach operations.

<code>shmid</code>	Stores and passes the desired shared memory segment identifier.
<code>shmflg</code>	Passes the value of flags to the <code>shmat(S)</code> system call.
<code>retma</code>	Stores the return value from <code>shmat(S)</code> .
<code>retrn</code>	Stores the return value from <code>shmdt</code> .
<code>detach</code>	Stores the desired number of detach operations.

12

This example program combines both the `shmat(S)` and `shmdt(S)` system calls. The program prompts for the number of attachments and enters a loop until they are done for the specified shared memory identifiers. Then, the program prompts for the number of detachments to be performed and enters a loop until they are done for the specified shared memory segment addresses.

Using `shmat`

The program prompts for the number of attachments to be performed, and the value is stored at the address of the `attach` variable (lines 17-21). A loop is entered using the `attach` variable and the `i` counter (lines 23-70) to perform the specified number of attachments. In this loop, the program prompts for a shared memory segment identifier (lines 24-27). The program stores the identifier at the address of the `shmid` variable (line 28).

Next, the program stores a NULL character at the address of the `addr` variable. The operating system then attaches the shared memory segment.

The program then prompts for the desired flags settings (lines 37-44), and stores the settings at the address of the flags variable (line 45). The `flags` variable is tested to determine the value of the variable given to the `shmflg` variable and `shmat(S)` system call (lines 46-60). If the call returns successfully, the system displays the attached address (lines 66-68). If the call returns unsuccessful, the system displays an error code (lines 62-63).

Using `shmdt`

After the `attach` loop completes, the program prompts for the number of detach operations to be performed (lines 71-75), and the value is stored at the address of the `detach` variable (line 76).

C Library Guide

A loop is entered using the `detach` variable and the `i` counter (lines 78-95) to perform the specified number of detachments. In this loop, the program prompts for the address of the shared memory segment to be detached (lines 79-83), and it is stored at the address of the `addr` variable (line 84). Then, the `shmdt(S)` system call is performed (line 87). If successful, the system displays a message stating so, along with the address that the segment was detached from (lines 92,93). If unsuccessful, the error number is displayed (line 89). The loop continues until it finishes.

The example program for the `shmop(S)` system call follows. In the example, the source file program is named `shmop.c` and the executable file is named `shmop`.

Example

```

1  /*This is a program to illustrate
2  **the shared memory operations, shmop(),
3  **system call capabilities.
4  */

5  /*Include necessary include files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/shm.h>
10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     int flags, i, attach;
15     int shmId, shmflg, retrn, detach; char * addr, * retrna;

16     /*Loop for attachments by this process.*/
17     printf("Enter the number of\n");
18     printf("attachments for this\n");
19     printf("process (1-4).\n");
20     printf("    Attachments = ");

21     scanf("%d", &attach);
22     printf("Number of attaches = %d\n", attach);

23     for(i = 1; i <= attach; i++) {
24         /*Enter the shared memory ID.*/
25         printf("\nEnter the shmId of\n");
26         printf("the shared memory segment to\n");
27         printf("be operated on = ");
28         scanf("%d", &shmId);
29         printf("\nshmId = %d\n", shmId);

30         /*Set shmaddr to NULL.*/
31
32
33
34
35         addr = (char *) 0;
36         printf("The desired address = 0x%x\n", addr);

37         /*Specify the desired flags.*/
38         printf("\nEnter the corresponding\n");
39         printf("number for the desired\n");
40         printf("flags:\n");
41         printf("SHM_RND                = 1\n");
42         printf("SHM_RDONLY                = 2\n");
43         printf("SHM_RND and SHM_RDONLY = 3\n");
44         printf("    Flags                = ");
45         scanf("%d", &flags);

```

C Library Guide

Example (cont.)

```
46         shmflg = 0; switch(flags)
47         {
48         case 1:
49             shmflg = SHM_RND;
50             break;
51         case 2:
52             shmflg = SHM_RDONLY;
53             break;
54         case 3:
55             shmflg = SHM_RND | SHM_RDONLY;
56             break;
57         }
58         printf("\nFlags = 0%o\n", shmflg);
59         /*Do the shmat system call.*/
60         retrna = shmat(shmid, addr, shmflg);
61         if(retrna == (char *) -1) {
62             printf("\nShmat failed. ");
63             printf("Error = %d\n", errno);
64         }
65         else {
66             printf ("\nShmat was successful\n");
67             printf("for shmid = %d\n", shmid);
68             printf("The address = 0x%x\n", retrna);
69         }
70     }
71     /*Loop for detachments by this process.*/
72     printf("Enter the number of\n");
73     printf("detachments for this\n");
74     printf("process (1-4).\n");
75     printf("      Detachments = ");
76     scanf("%d", &detach);
77     printf("Number of dettaches = %d\n", detach);
78     for(i = 1; i <= detach; i++) {
79         /*Enter the value for shmaddr.*/
80         printf("\nEnter the value for\n");
81         printf("the shared memory address\n");
82         printf("in hexadecimal:\n");
83         printf("      Shmaddr = ");
84         scanf("%x", &addr);
85         printf("The desired address = 0x%x\n", addr);
86         /*Do the shmdt system call.*/
87         retrn = shmdt(addr);
88         if(retrn == -1) {
89             printf("Error = %d\n", errno);
90         }
91         else {
92             printf ("\nShmdt was successful\n");
93             printf("for address = 0x%x\n", addr);
94         }
95     }
96 }
```

Appendix A

Library Routine Error Messages

A.1 Introduction A-1

A.2 errno Values A-1

A.3 Math Errors A-8

A.1 Introduction

This appendix lists and describes the values to which the **errno** variable can be set when an error occurs in a call to a library routine. Note that only some routines set the **errno** variable. The reference pages for the routines that set **errno** upon error explicitly mention the **errno** variable (see Subroutines(S) of the *XENIX Programmer's Reference*). This mention will be found in the "Return Value" section of the reference page. If no mention of **errno** occurs, the routine does not set a value.

An error message is associated with each **errno** value. This message, along with a user-supplied message, can be printed by using the **perror** function.

The value of **errno** reflects the error value for the last call that set **errno**. This value is not automatically cleared by later successful calls. You should test for errors and print error messages immediately after a function call to obtain accurate results.

The include file **errno.h** contains the definitions of the **errno** values.

A.2 errno Values

The following list gives the **errno** values used in the environment, the system error message corresponding to each value, and a brief description of the circumstances that cause each error:

Value	Message	Description
EPERM	Not owner	Indicates an attempt to modify a file when the user is not the owner or a super-user. Also returned for attempts to perform actions allowed only to a super-user.
ENOENT	No such file or directory	Occurs when a specified file doesn't exist, or when a directory specified in a pathname doesn't exist.

C Library Guide

Value	Message	Description
ESRCH	No such process	The process specified by <i>pid</i> in kill or ptrace cannot be found.
EINTR	Interrupted system call	An asynchronous signal (which the user has elected to catch) occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.
EIO	I/O error	A physical I/O error. May occur on a call following the one to which it actually applies.
ENXIO	No such device or address	I/O on a special file refers to a subdevice that doesn't exist or is beyond the limits of the device. It may also occur when, for example, a tape drive is not on line or no disk pack is loaded on a drive.
E2BIG	Argument list too long	An argument list longer than 5120 bytes is presented to a member of the exec family.
ENOEXEC	Exec format error	A request has been made to execute a file with valid permissions but without a valid "magic number" (see a.out (F) in the <i>XENIX User's Reference</i>).
EBADF	Bad file number	Either a file descriptor refers to no open file or a read request was made to a write-only file (or vice versa).

Library Routine Error Messages

Value	Message	Description
ECHILD	No child processes	A wait was executed by a process that had not waited for child processes.
EAGAIN	No more processes	A fork failed because the process table was full or the user had the maximum number of processes.
ENOMEM	Not enough space	During an exec or sbrk , a request was made for more space than was available. Available space is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many registers or if there is not enough swap space during a fork .
EACCES	Permission denied	An attempt was made to access a file in a way denied by the protection system.
EFAULT	Bad address	The system encountered a hardware fault while attempting to use an argument from a system call.
ENOTBLK	Block device required	A non-block file was specified where a block device was required, e.g., mount .



C Library Guide

EBUSY	Device busy	An attempt was made to mount a device that was already mounted or an attempt was made to dismount a device on which there was an active file (open file, current directory, mounted-on file, or active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled.
EEXIST	File exists	An existing file was specified in an inappropriate context, for example, the <i>link</i> file.
EXDEV	Cross-device link	A link to a file on another device was attempted.
ENODEV	No such device	An attempt was made to apply an inappropriate system call to a device, for example, a write-only device.
ENOTDIR	Not a directory	A nondirectory was specified where a directory was required, for example, in a path prefix or as an argument to chdir .
EISDIR	Is a directory	An attempt was made to write on a directory.
EINVAL	Invalid argument	An invalid argument, such as mentioning an undefined signal to signal was passed to a routine. Also set by the math functions described in Subroutines(S) of the <i>XENIX Programmer's Reference</i> .

Library Routine Error Messages

ENFILE	File table overflow	The system's table of open files is full and (temporarily) no more opens can be accepted.
EMFILE	Too many open files	No process can have more than 60 file descriptors open simultaneously.
ENOTTY	Not a typewriter	An attempt was made to write to a device other than the terminal.
ETXTBSY	Text file busy	An attempt was made to execute a pure-procedure program that is currently open for writing (or reading). Also indicates an attempt to open for writing a pure-procedure program that is being executed.
EFBIG	File too large	The size of a file exceeded the maximum file size (1,082,201,088 bytes) or ulimit ; for more information, see ulimit(S) in the <i>XENIX Programmer's Reference</i> .
ENOSPC	No space left on device	During a write to an ordinary file, there was no free space left on the device.
ESPIPE	Illegal seek	An lseek was issued to a pipe.
EROFS	Read-only file system	An attempt to modify a file or directory was made on a device mounted read only.
EMLINK	Too many links	An attempt was made to make more than the maximum number of links (1000) to a file.



C Library Guide

EPIPE	Broken pipe	A write was attempted on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.
EDOM	Math arg out of domain of func	The argument of a function in the math package is out of the domain of the function.
ERANGE	Math result not representable	The value of a function in the math package is not representable within machine precision.
EUCLEAN	File system needs cleaning	An attempt was made to mount a file system whose super-block is not flagged clean. See mount(S) in the <i>XENIX Programmer's Reference</i> .
EDEADLOCK	Would deadlock	A process's attempt to lock a file region would cause a deadlock between processes vying for control of that region.
ENOTNAM	Not a name file	A creatsem , opensem , waitsem , or sigsem was issued using an invalid semaphore identifier. For more information, see the appropriate manual pages in Subroutines(S) of the <i>XENIX Programmer's Reference</i> .

Library Routine Error Messages

- ENAVAIL** Not available **An opensem, waitsem, or sigsem** was issued to a semaphore that has not been initialized by a call to **creatsem**. A **sigsem** was issued to a semaphore out of sequence (that is, before the process has issued the corresponding **waitsem** to the semaphore). An **nbwaitsem** was issued to a semaphore guarding a resource that is currently in use by another process. The semaphore on which a process was waiting has been left in an inconsistent state when the process controlling the semaphore exits without relinquishing control properly (that is, without using a **waitsem** on the semaphore). For more information, see the appropriate manual page in Subroutines(S) of the *XENIX Programmer's Reference*.
- EISNAM** A name file A name file (for example, a semaphore or shared data) was specified when not expected.
- ENOMSG** No message of desired type An effort was made to receive a message of a type that does not exist in the specified message queue (see **msgop** (S) in the *XENIX Programmer's Reference*).
- EIDRM** Identifier removed This error is returned to processes that resume execution due to the removal of an identifier for the file system's name space (see **msgctl** (S), **semctl** (S), and **shmctl** (S) in the *XENIX Programmer's Reference*).
- ENOLCK** No locks available An attempt was made to lock a file region and there are no more free locks.



C Library Guide

A.3 Math Errors

The functions in the math library (the include file *math.h*) will either set **errno** to ERANGE or to EDOM (depending on which function is called). An **errno** value will be set if the calculation to be performed is inappropriate or would result in an overflow.

A list of the functions in the math library and the values that they return follows:

Function	Value	Description
exp, pow	ERANGE	Set only for extremely huge arguments. A huge value is returned in overflow conditions.
pow	EDOM	Returns a huge negative value when x is nonpositive and y is not an integer, or when x and y are both zero.
log, log10	EDOM	Returns a huge negative value when the argument given is nonpositive.
sqrt	EDOM	Returns zero when the argument is negative.
hypot, cabs	-	Both return the following line: $\text{sqrt}(x*x + y*y)$ Overflows are precluded.
sinh, cosh, tanh	-	Both sinh and cosh will return a huge value of an appropriate sign when the correct value would overflow.

Appendix B

Common Libraries

- B.1 Introduction B-1
- B.2 Run-Time Routines B-1
 - B.2.1 Routines Common to MS-DOS and XENIX B-1
 - B.2.2 Routines Specific to MS-DOS B-2
 - B.2.3 ANSI Library B-4
- B.3 Global Variables B-5
 - B.3.1 Variables Common to MS-DOS and XENIX B-5
 - B.3.2 Variables Specific to MS-DOS B-5
- B.4 Include Files B-6
 - B.4.1 Include Files Common to MS-DOS and XENIX B-6
 - B.4.2 Include Files Specific to MS-DOS B-6
 - B.4.3 ANSI Include Files B-7
- B.5 Differences Between Routines Common to MS-DOS and XENIX B-7
 - B.5.1 abort B-7
 - B.5.2 access B-7
 - B.5.3 chdir B-8
 - B.5.4 chmod B-8
 - B.5.5 creat B-8
 - B.5.6 exec B-9
 - B.5.7 fopen, freopen B-10
 - B.5.8 fread B-10
 - B.5.9 fseek B-10
 - B.5.10 fstat B-11
 - B.5.11 ftell B-12
 - B.5.12 ftime B-12
 - B.5.13 fwrite B-12
 - B.5.14 getpid B-12
 - B.5.15 locking B-13
 - B.5.16 log, log10 B-13
 - B.5.17 lseek B-13
 - B.5.18 open B-14

B.5.19 read B-14
B.5.20 signal B-14
B.5.21 stat B-15
B.5.22 system B-15
B.5.23 umask B-16
B.5.24 unlink B-16
B.5.25 utime B-16
B.5.26 write B-16

B.1 Introduction

This appendix lists and describes routines from the Microsoft C Run-Time Library for XENIX that operate compatibly with C library routines on MS-DOS systems. The routines provide an identical interface to a set of operations useful on both XENIX and MS-DOS systems.

With the exception of error returns, the math functions in the Microsoft C Run-Time Library for MS-DOS operate compatibly with the XENIX routines of the same names. Error returns for most math routines in the MS-DOS library have been upgraded for compatibility with UNIX System V math-error handling.



B.2 Run-Time Routines

The following sections list routines from the MS-DOS C library that are compatible with XENIX routines. Routines specific to the MS-DOS environment are also listed.

B.2.1 Routines Common to MS-DOS and XENIX

The following is a list of the routines common to MS-DOS and XENIX:

abort ¹	ceil	execlp ¹	fileno	fwrite ¹
abs	chdir ¹	execv ¹	floor	gcvt
access ¹	chmod ²	execve ¹	fmod	getchar
acos ²	chsize	execvp ¹	fopen ¹	getcwd
asctime	clearerr	execvpe ¹	fprintf	getenv
asin ²	close	exit	fputc	getpid ¹
assert	cos ²	exp	fputs	gets
atan ²	cosh ²	fabs	fread ¹	getw
atan2 ²	creat ¹	fclose	free	gtime
atof	ctime	fcvt	freopen ¹	hypot
atoi	difftime	fdopen	frexp	isalnum
atol	dup	feof	fscanf	isalpha
bessel ³	dup2	ferror	fseek ¹	isascii
bsearch	ecvt	fflush	fstat ¹	iscntrl
cabs	execl ¹	fgetc	ftell ¹	isdigit
calloc	execle ¹	fgets	ftime ¹	isgraph

C Library Guide

islower	modf	scanf	strdup	tmpfile
isprint	onexit	setbuf	strlen	tmpnam
ispunct	perror	setjmp	strncat	toascii
isspace	pow ²	setvbuf	strncmp	tolower
isupper	printf	signal ¹	strncpy	_tolower
isxdigit	putc	sin ²	strpbrk	toupper
ldexp ²	putchar	sinh ²	strrchr	_toupper
lfind	putenv	sprintf	strspn	tzset
localtime	puts	sqrt ²	strtod	umask ¹
locking ¹	putw	srand	strtok	ungetc
log	qsort	sscanf	strtol	unlink ²
log10	rand	stat ¹	swab	utime ¹
longjmp	read ¹	strcat	system ¹	vfprintf
lsearch	realloc	strchr	tan ²	vprintf
lseek ¹	rewind	strcmp	tanh ²	vsprintf
malloc	rmtmp	strcpy	tempnam	write ¹
mktemp	sbrk	strcsfn	time	

¹ Operates differently or has different meaning under MS-DOS than under XENIX. The differences are detailed in "Common Libraries" in this guide.

² Implements UNIX System V-style error returns.

³ The `bessel` routine does not correspond to a single function, but to six functions named `j0`, `j1`, `jn`, `y0`, `y1`, and `yn`. They all implement Unix System V-style error returns.

B.2.2 Routines Specific to MS-DOS

The following routines are available only in the MS-DOS C library. Programmers who are writing code to be ported to XENIX systems should avoid using these routines.

<code>_arc</code>	<code>_dos_keep</code>	<code>_getbkcolor</code>
<code>_bdos</code>	<code>_dos_open</code>	<code>_getch</code>
<code>_bios_disk</code>	<code>_dos_read</code>	<code>_getche</code>
<code>_bios_equiplist</code>	<code>_dos_setblock</code>	<code>_getcolor</code>
<code>_bios_keybrd</code>	<code>_dos_setdate</code>	<code>_getcursposition</code>
<code>_bios_memsize</code>	<code>_dos_setdrive</code>	<code>_getfillmask</code>
<code>_bios_printer</code>	<code>_dos_setfileattr</code>	<code>_getimage</code>
<code>_bios_serialcom</code>	<code>_dos_setftime</code>	<code>_getlinestyle</code>
<code>_bios_timeofday</code>	<code>_dos_settime</code>	<code>_getlogcoord</code>

cgets	_dos_setvect	_getphyscoord
_chain_intr	_dos_write	_getpixel
_clear87	_dosexterr	_getttextcolor
_clearscreen	_ellipse	_getttextposition
_control87	_enable	_getvideoconfig
cprintf	eof	halloc
cscanf	_exit	_harderr
diecetomsbin	fcloseall	_hardresume
_disable	ffree	_hardretn
_displaycursor	fgetchar	_heapchk
dmsbintoiee	_fheapchk	_heapset
_dos_allocmem	_fheapset	_heapwalk
_dos_close	_fheapwalk	hfree
_dos_creat	fiectomsbin	_imagesize
_dos_creatnew	filelength	inp
_dos_findfirst	_floodfill	inpw
_dos_findnext	_flushall	int86
_dos_freemem	_fmalloc	int86x
_dos_getdate	fmsbintoiee	intdos
_dos_getdiskfree	fmsize	intdosx
_dos_getdrive	FP_OFF	isatty
_dos_getfileattr	FP_SEG	itoa
_dos_getftime	_fpreset	kbhit
_dos_gettime	fputchar	labs
_dos_getvect	_freect	_lineto
_lrotl	remove	spawnlp
_lrotr	rename	spawnlpe
ltoa	rmdir	spawnv
_makepath	_rotl	spawnve
max	_rotr	spawnvp
_memavl	_searchenv	spawnvpe
min	segread	_splitpath
mkdir	_selectpalette	stackavail
movedata	_setactivepage	_status
_moveto	_setbkcolor	strcmpi
_msize	_setcliprgn	_strdate
_nfree	_setcolor	strlwr
_nheapchk	_setfillmask	strncmpi
_nheapset	_setlinestyle	strnicmp
_nheapwalk	_setlogorg	strnset
_nmalloc	setmode	strrev



C Library Guide

<code>_nmsize</code>	<code>_setpixel</code>	<code>strset</code>
<code>outp</code>	<code>_setttextcolor</code>	<code>strstr</code>
<code>outpw</code>	<code>_setttextposition</code>	<code>_strtime</code>
<code>_outtext</code>	<code>_setttextwindow</code>	<code>strupr</code>
<code>_pie</code>	<code>_setvideomode</code>	<code>tell</code>
<code>putch</code>	<code>_setviewport</code>	<code>ultoa</code>
<code>_putimage</code>	<code>_setvisualpage</code>	<code>ungetch</code>
<code>_rectangle</code>	<code>sopen</code>	<code>_wragon</code>
<code>_remappalette</code>	<code>spawnl</code>	
<code>_remappalette</code>	<code>spawnle</code>	

B.2.3 ANSI Library

The Microsoft C Run-Time Library includes routines that conform to the Draft Proposed ANSI Standard (ANSI). These routines are listed as follows. Programs which must strictly adhere to ANSI should use only these routines.

<code>abort</code>	<code>difftime</code>	<code>freopen</code>	<code>isspace</code>	<code>printf</code>
<code>abs</code>	<code>div</code>	<code>frexp</code>	<code>isupper</code>	<code>putc</code>
<code>acos</code>	<code>exit</code>	<code>fscanf</code>	<code>isxdigit</code>	<code>putchar</code>
<code>asctime</code>	<code>exp</code>	<code>fseek</code>	<code>labs</code>	<code>puts</code>
<code>asin</code>	<code>fabs</code>	<code>fsetpos</code>	<code>ldexp</code>	<code>qsort</code>
<code>assert</code>	<code>fclose</code>	<code>ftell</code>	<code>ldiv</code>	<code>raise</code>
<code>atan</code>	<code>feof</code>	<code>fwrite</code>	<code>localtime</code>	<code>rand</code>
<code>atan2</code>	<code>ferror</code>	<code>getc</code>	<code>log</code>	<code>realloc</code>
<code>atexit</code>	<code>fflush</code>	<code>getchar</code>	<code>log10</code>	<code>remove</code>
<code>atof</code>	<code>fgetc</code>	<code>getenv</code>	<code>longjmp</code>	<code>rename</code>
<code>atoi</code>	<code>fgetpos</code>	<code>gets</code>	<code>malloc</code>	<code>rewind</code>
<code>atol</code>	<code>fgets</code>	<code>gmtime</code>	<code>memchr</code>	<code>scanf</code>
<code>bsearch</code>	<code>floor</code>	<code>isalnum</code>	<code>memcmp</code>	<code>setbuf</code>
<code>calloc</code>	<code>fmod</code>	<code>isalpha</code>	<code>memcpy</code>	<code>setjmp</code>
<code>ceil</code>	<code>fopen</code>	<code>iscntrl</code>	<code>memmove</code>	<code>setvbuf</code>
<code>clearerr</code>	<code>fprintf</code>	<code>isdigit</code>	<code>memset</code>	<code>signal</code>
<code>clock</code>	<code>fputc</code>	<code>isgraph</code>	<code>mktime</code>	<code>sin</code>
<code>cos</code>	<code>fputs</code>	<code>islower</code>	<code>modf</code>	<code>sinh</code>
<code>cosh</code>	<code>fread</code>	<code>isprint</code>	<code>perror</code>	<code>sprintf</code>
<code>ctime</code>	<code>free</code>	<code>ispunct</code>	<code>pow</code>	<code>sqrt</code>

B.3 Global Variables

The following sections list global variables used in the MS-DOS C library that are also used in XENIX environment. The variables specific to the MS-DOS environment are also listed.

B.3.1 Variables Common to MS-DOS and XENIX



The following is a list of global variables used in the run-time library and available in both the MS-DOS and XENIX environments:

daylight
environ
errno
sys_errlist
sys_nerr
timezone
tzname

Note

Not all values of **errno** available on XENIX are used by the MS-DOS run-time library.

B.3.2 Variables Specific to MS-DOS

The following global variables are available only in the MS-DOS C library. Programmers who are writing code to be ported to XENIX systems should avoid using these variables.

_doserrno
_fmode
_osmajor
_osminor
_psp

C Library Guide

B.4 Include Files

Structure definitions, return value types, and manifest constants used in the descriptions of some of the common routines may vary from environment to environment and are therefore fully defined in a set of include files for each environment. Include files provided with the MS-DOS C library are compatible with include files of the same name on XENIX and UNIX systems.

See “Common Libraries” in this guide for a list of the MS-DOS include files that are compatible with XENIX. (The include files that apply only to MS-DOS environments are also found there.)

B.4.1 Include Files Common to MS-DOS and XENIX

The following MS-DOS include files are compatible with the XENIX (and UNIX) include files of the same name:

assert.h	setjmp.h	sys\timebh.h
ctype.h	signal.h	sys\typesh.h
errno.h	stdio.h	time.h
fcntl.h	sys\locking.h	
math.h	sys\stath.h	

B.4.2 Include Files Specific to MS-DOS

The following include files are used only in MS-DOS environments and do not have counterparts on XENIX and UNIX systems:

conio.h	io.h	stdlib.h
direct.h	process.h	sys\utime.h
dos.h	share.h	
graph.h	stdarg.h	

B.4.3 ANSI Include Files

The include files necessary to use the ANSI run-time library are as listed:

assert.h	math.h	stdio.h
ctype.h	setjmp.h	stdlib.h
float.h	signal.h	string.h
limits.h	stdarg.h	time.h



B.5 Differences Between Routines Common to MS-DOS and XENIX

This section explains how the MS-DOS routines in the common library for XENIX and MS-DOS differ from their XENIX counterparts. These descriptions are intended to be used in conjunction with the more detailed descriptions provided in the Subroutines(S) section of the *XENIX Programmer's Reference*.

B.5.1 abort

The MS-DOS version of the **abort** routine terminates the process by a call to **raise(SIG_ABRT)**. Control is returned to the parent (calling) process with an exit status of 3 and the following message is printed to standard error:

```
Abnormal program termination
```

No core dump occurs on MS-DOS.

B.5.2 access

The **access** routine checks the access to a given file. Under MS-DOS, the real and effective user IDs are nonexistent. The permission (access) setting can be any combination of the following values:

<u>Value</u>	<u>Meaning</u>
04	Read
02	Write
00	Check for existence

The "Execute" access mode (01) is not implemented.

C Library Guide

In case of error, only the **EACCES** and **ENOENT** values may be returned for **errno** on MS-DOS.

B.5.3 chdir

In case of error, only the **ENOENT** value may be returned for **errno** on MS-DOS.

B.5.4 chmod

The **chmod** routine can set the “owner” access permissions for a given file, but all other permission settings are ignored. The mode argument can be any one of the constant expressions shown in the following list’s left-most column; the equivalent XENIX value is shown in the right-most column:

<u>Constant Expression</u>	<u>Meaning</u>	<u>XENIX Value</u>
S_IREAD	Read by owner	0400
S_IWRITE	Write by owner	0200
S_IREAD S_IWRITE	Read and write by owner	0000

The **S_IREAD** and **S_IWRITE** constants are defined in the `sys\stat.h` include file. Note that the **OR** operator (`|`) is used to combine these constants to form read and write permission.

If write permission is not given, the file is treated as a read-only file. Giving write-only permission is allowed, but has no effect; under MS-DOS, all files are readable.

In case of error, only the **ENOENT** value may be returned for **errno** on MS-DOS.

B.5.5 creat

The **creat** routine creates a new file or prepares an existing file for writing. If the file is created successfully, the access permissions are set as defined by the mode argument. Only “owner” permissions are allowed (see the section on “**chmod**”).

In case of error, only the **EACCES**, **EMFILE**, and **ENOENT** values may be returned for **errno** on MS-DOS.

Use of the **open** routine is preferred over **creat** when creating or opening files in both MS-DOS and XENIX environments.

B.5.6 **exec**



The MS-DOS versions of the **execl**, **execle**, **execlp**, **execlpe**, **execv**, **execve**, **execvpe**, and **execvp** routines overlay the calling process, as in the XENIX environment. If there is not enough memory for the new process, the **exec** routine fails and returns to the calling process. Otherwise, the new process begins execution.

Under MS-DOS, the **exec** routines *do not* perform the following functions:

- Use the close-on-exec flag to determine open files for the new process.
- Disable profiling for the new process (profiling is not available under MS-DOS).
- Pass signal settings to the child process. Under MS-DOS, all signals (including signals set to be ignored) are reset to the default in the child process.

The combined size of all arguments (including the program name) in an **exec** routine under MS-DOS must not exceed 128 bytes.

In case of error, the **E2BIG**, **EACCES**, **ENOENT**, **ENOEXEC**, and **ENOMEM** values may be returned for **errno** on MS-DOS. In addition, the **EMFILE** value may be used; under MS-DOS, the file must be opened to determine whether it is executable.

C Library Guide

B.5.7 fopen, freopen

The MS-DOS versions of the **fopen** and **freopen** routines open stream files just as they do in the XENIX environment. However, under MS-DOS the following additional values for the *type* string are available:

Value	Use
t	Opens the file in text mode. Opening a file in this mode causes translation of carriage-return-line-feed (CR-LF) character combinations into a single line feed (LF) on input. Similarly, on output, line feeds are translated into CR-LF combinations.
b	Opens the file in binary mode. This mode suppresses translation.

The MS-DOS and XENIX versions of these routines also differ in their interpretation of append mode (**a** or **a+**). When append mode is specified in the MS-DOS version of **fopen** or **freopen**, the file pointer is repositioned at the end of the file prior to **write** operations. Thus, all **write** operations take place at the end of the file.

In the XENIX versions, all **write** operations take place at the current position of the file pointer. In append mode, the file pointer is initially positioned at the end of the file, but if the file pointer is later repositioned, **write** operations take place at the new position rather than at the end of the file.

B.5.8 fread

The MS-DOS **fread** routine uses the low-level **read** function to carry out **read** operations. If the file has been opened in text mode, **read** replaces each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the CR-LF pairs have been replaced. Thus the return value may not always correspond to the actual number of bytes read. This is considered normal and has no implications for detecting the end of the file.

B.5.9 fseek

Both the MS-DOS and XENIX versions of the **fseek** routine move the file pointer to the given position. However, for streams opened in text mode, the MS-DOS version of **fseek** has limited use because CR-LF translations can cause **fseek** to produce unexpected results. Only two **fseek** operations

are guaranteed to work on streams opened in text mode: seeking with an offset of 0 relative to any of the origin values, and seeking from the beginning of the file with an offset value returned from a call to **ftell**.

B.5.10 **fstat**

MS-DOS does not make as much information available for file handles as it does for full pathnames; thus the MS-DOS version of **fstat** returns less useful information than does the **stat** routine. The MS-DOS **fstat** routine can detect device files, but it must not be used with directories.



The structure returned by **fstat** contains the following members:

Member	Meaning
st_atime	Time of last modification of file (same as st_mtime and st_ctime).
st_ctime	Time of last modification of file (same as st_atime and st_mtime).
st_dev	Either the drive number of the disk containing the file, or the file handle in the case of a device (same as st_rdev).
st_gid	Not used.
st_ino	Not used.
st_mode	User read and write bits reflect the file's permission setting. The S_IFCHR bit is set for a device; otherwise, the S_IFREG bit is set.
st_mtime	Time of last modification of file (same as st_atime and st_ctime).
st_nlink	Always 1.
st_rdev	Either the drive number of the disk containing the file, or the file handle in the case of a device (same as st_dev).
st_size	Size, in bytes, of the file.
st_uid	Not used.

C Library Guide

In case of error, only the **EBADF** value may be returned for **errno** on MS-DOS.

B.5.11 **ftell**

Both the MS-DOS and XENIX versions of the **ftell** routine get the current file-pointer position. In MS-DOS, however, for streams opened in text mode, the value returned by **ftell** may not reflect the physical byte offset, since text mode causes CR-LF translation. The **ftell** routine can be used in conjunction with the **fseek** routine to remember and return to file locations correctly. If you want the actual offset to a file position, open the stream in binary mode and perform type conversions as necessary.

B.5.12 **ftime**

Unlike the system time on XENIX systems, the MS-DOS system time does not include the concept of a default time zone. Instead, **ftime** uses the value of an MS-DOS environment variable named **TZ** to determine the time zone. The user can set the default time zone by setting the **TZ** variable. If **TZ** is not explicitly set, the default time zone corresponds to the Pacific time zone. For details on the **TZ** variable, see “The **daylight**, **timezone**, and **tzname** Variables,” in the “Global Variables and Standard Types” chapter or the Miscellaneous Features(M) section of the *XENIX User's Reference*.

B.5.13 **fwrite**

The MS-DOS **fwrite** routine uses the low-level **write** function to carry out **write** operations. If the file is opened in text mode, every line-feed (LF) character in the output is replaced by a carriage-return-line-feed (CR-LF) pair before being written. This does not affect the return value.

B.5.14 **getpid**

The **getpid** routine returns a process-unique number. Although the number may be used to uniquely identify the process, it does not have the same meaning as the process identification returned by **getpid** in the XENIX environment.

B.5.15 locking

The MS-DOS and XENIX versions of the **locking** routine differ in several respects, as listed:

- On MS-DOS, it is not possible to lock a file only against write access; locking a region of a file prevents both reading and writing in that region. Thus, setting **LK_RLCK** in the **locking** call is equivalent to setting **LK_LOCK**, and setting **LK_NBRLOCK** is equivalent to setting **LK_NBLCK**.
- On MS-DOS, specifying **LK_LOCK** or **LK_RLCK** will *not* cause a program to wait until the specified region of a file is unlocked. Instead, up to ten attempts are made to lock the file (one attempt per second). If the lock is still unsuccessful after 10 seconds, the **locking** function returns an error value.

On XENIX, if the first attempt at locking fails, the locking process “sleeps” (suspends execution) and periodically “wakes” to attempt the lock again. There is no limit on the number of attempts, and the process can continue indefinitely.

- On MS-DOS, locking of overlapping regions of a file is not allowed.
- On MS-DOS, if more than one region of a file is locked, only one region can be unlocked at a time, and the region must correspond to a region that was previously locked. You cannot unlock more than one region at a time, even if the regions are adjacent.

B.5.16 log, log10

Passing a 0 to **log** or **log10** sets the **errno** variable to **EDOM** on XENIX, instead of setting it to **ERANGE** as it does on MS-DOS.

B.5.17 lseek

In case of error, only the **EBADF** and **EINVAL** values may be returned for **errno** on MS-DOS.

C Library Guide

B.5.18 open

Both the MS-DOS and XENIX versions of the **open** routine open a file by its handle. However, with MS-DOS, two additional *oflag* values (**O_BINARY** and **O_TEXT**) are available and the **O_NDELAY** and **O_SYNCW** values are not available.

The **O_BINARY** flag causes the file to be opened in binary mode, regardless of the default mode setting. Similarly, the **O_TEXT** flag causes the file to be opened in text mode.

In case of error, only the **EACCES**, **EEXIST**, **EMFILE**, and **ENOENT** values may be used for **errno** on MS-DOS.

B.5.19 read

Both the MS-DOS and XENIX versions of the **read** routine read characters from the file given by a file handle. However, if the file has been opened in text mode, the MS-DOS version of **read** replaces each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the CR-LF pairs have been replaced. Thus, the return value may not always correspond to the actual number of bytes read. This is considered normal and has no implications for detecting an end-of-file condition.

In case of error, only the **EBADF** value may be used for **errno** on MS-DOS.

B.5.20 signal

The MS-DOS version of the **signal** routine can only handle the **SIGINT**, **SIGFPE**, **SIGABRT**, **SIGILL**, and **SIGSEGV** signals. In MS-DOS, **SIGINT** is defined to be INT 23H (the signal), **SIGFPE** corresponds to floating-point exceptions that are not masked, **SIGABRT** is the default **abort** handler, and **SIGILL** and **SIGSEGV** are undefined, but provided for ANSI compatibility.

On MS-DOS, child processes executed through the **exec** or **spawn** routines do not inherit the signal settings of the parent process. All signal settings (including signals set to be ignored) are reset to the default settings in the child process.

The MS-DOS version of **signal** uses only **EINVAL** for **errno**.

B.5.21 stat

The **stat** routine returns a structure defining the current status of the given file or directory. The structure members returned by **stat** have the following names and meanings on MS-DOS:

Value	Meaning
st_atime	Time of last modification of file (same as st_mtime and st_ctime).
st_ctime	Time of last modification of file (same as st_atime and st_mtime).
st_dev	Drive number of the disk containing the file (same as st_rdev).
st_gid	Not used.
st_ino	Not used.
st_mode	User read and write bits reflect the file's permission setting. The S_IFDIR bit is set for a device; otherwise, the S_IFREG bit is set.
st_mtime	Time of last modification of file (same as st_atime and st_ctime).
st_nlink	Always 1.
st_rdev	Drive number of the disk containing the file (same as st_dev).
st_size	Size, in bytes, of the file.
st_uid	Not used.

In case of error, only the **ENOENT** value may be returned for **errno** on MS-DOS.

B.5.22 system

The **system** routine passes the given string to the operating system for execution. For MS-DOS to execute this string, the full pathname of the directory containing it must be assigned to the environment variable. If the string is a **NULL**, the system searches for **COMMAND.COM**.



C Library Guide

The **system** call returns an error if the string cannot be found using these variables. Where a null pointer is passed, it sets **errno** to **ENOENT** and returns 0 if it cannot find **COMMAND.COM**, and 1 if it can. In case of error, only the **E2BIG**, **ENOENT**, **ENOEXEC**, and **ENOMEM** values may be returned for **errno** on MS-DOS.

B.5.23 umask

The **umask** routine can set a mask for “owner” read and write access permissions only. All other permissions are ignored. (See the discussion of the **access** routine above for details.)

B.5.24 unlink

The MS-DOS version of the **unlink** routine always deletes the given file. Since MS-DOS does not implement multiple “links” to the same file, unlinking a file is the same as deleting it.

In case of error, only the **EACCES** and **ENOENT** values may be returned for **errno** on MS-DOS.

B.5.25 utime

The MS-DOS **utime** routine sets the file modification time only; MS-DOS does not maintain a separate access time.

In case of error, the **EACCES** and **ENOENT** values may be returned for **errno** on MS-DOS. In addition, the **EMFILE** value may be used; under MS-DOS, the file must be opened to set the modification time.

B.5.26 write

Both the MS-DOS and XENIX versions of the **write** routine write a specified number of characters to the file named by the given file handle. However, in the MS-DOS version, if the file has been opened in text mode, every line-feed (LF) character in the output is replaced by a carriage-return-line-feed (CR-LF) pair before being written. This does not affect the return value.

In case of error, only the **EBADF** and **ENOSPC** values may be returned for **errno** on MS-DOS.

Appendix C

XENIX to DOS: A Cross

Development System

- C.1 Introduction C-1
- C.2 Creating Source Files C-2
- C.3 Compiling a DOS Source File C-2
 - C.3.1 DOS Floating Point Flags C-3
- C.4 Using Assembly Language Source Files C-4
- C.5 Creating and Linking Object Files C-4
- C.6 Running and Debugging a DOS Program C-5
- C.7 Transferring Programs Between Systems C-5
- C.8 Creating DOS Libraries C-7
- C.9 Common Run-Time Routines C-7
 - C.9.1 Common Routines for DOS and XENIX C-7
 - C.9.2 Common Routines for DOS and UNIX System V C-8
 - C.9.3 Routines Specific to DOS C-8
- C.10 Common System-Wide Variables C-9
 - C.10.1 Common Variables for DOS and XENIX C-9
 - C.10.2 Common Variables for DOS and UNIX System V C-10
 - C.10.3 Variables Specific to DOS C-10
- C.11 Common Include Files C-10
 - C.11.1 Common Include Files for DOS and XENIX C-10
 - C.11.2 Common Include Files for DOS and UNIX System V C-11
 - C.11.3 Include Files Specific to DOS C-11

C.12 Differences Between Common Routines C-11

- C.12.1 abort C-11
- C.12.2 access C-12
- C.12.3 chdir C-12
- C.12.4 chmod C-12
- C.12.5 creat C-13
- C.12.6 exec C-13
- C.12.7 fopen, freopen C-14
- C.12.8 fread C-14
- C.12.9 fseek C-15
- C.12.10 fstat C-15
- C.12.11 ftell C-16
- C.12.12 ftime C-16
- C.12.13 fwrite C-16
- C.12.14 getpid C-17
- C.12.15 locking C-17
- C.12.16 lseek C-17
- C.12.17 open C-18
- C.12.18 read C-18
- C.12.19 signal C-18
- C.12.20 stat C-19
- C.12.21 system C-20
- C.12.22 umask C-20
- C.12.23 unlink C-20
- C.12.24 utime C-20
- C.12.25 write C-20

C.13 Differences in Definitions C-21

C.1 Introduction

The XENIX system provides a variety of tools to create programs that can be executed under the DOS operating system. The DOS cross development system lets you create, compile, and link DOS programs on the XENIX system and transfer these programs to a DOS system for execution and debugging.

The complete development system consists of:

- The C program compiler **cc**
- The assembler **masm**
- The DOS linker **dosld**
- The DOS libraries (in */usr/lib/dos*)
- The DOS include files (in */usr/include/dos*)
- The **dos(C)** commands

The heart of the cross development system is the **cc** command. The command provides a special **-dos** option that directs the compiler to create code for execution under DOS. When **-dos** is given, **cc** uses the special DOS include files and libraries to create a program. The resulting program file has the correct format for execution on any DOS system.

The **cc** command uses the **dosld** program to carry out the last part of the compilation process, the creation of the executable program file. **cc** invokes the **masm** command only when XENIX assembly language source files are given in the command line. In most cases, **cc** invokes **masm** and **dosld** automatically. You can also invoke them directly when you need to perform special tasks.

The last step in the cross development process is to transfer the executable program files to a DOS system. Since DOS programs cannot be executed or debugged on the XENIX system, you must copy the resulting programs to DOS before attempting execution. You can do this using the XENIX **dos(C)** commands. For example, the **doscp** command lets you copy files back and forth between XENIX and DOS disks. This means you can transfer program files from the XENIX system to a DOS system, or copy source files from a DOS system to XENIX.



C Library Guide

C.2 Creating Source Files

You can create program source files using either XENIX or DOS text editors. The most convenient way is to use a XENIX editor, such as **vi**, since this means you do not have to transfer the source files from the DOS system to XENIX each time you make changes to the files.

When creating source files, you should follow these simple rules:

- Use the standard C language format for your source files. DOS C and assembler source files have the same format as XENIX source files. In fact, many DOS programs, if compiled without the **-dos** option, can be executed on the XENIX system.
- Use the DOS naming conventions when giving file and directory names within a program; e.g., use “\” instead of “/” for the path-name separator. Since the compiler does not check names, failure to follow the conventions will cause errors when the program is executed.
- Use only the DOS include files and library functions. Most DOS include files and functions are identical to their XENIX counterparts. Others have only slight differences. For a list of the available DOS include files and functions, and a description of the differences between them and the corresponding XENIX files and functions, see section A.11 of this appendix.

If you use a function that does not exist, **dosld** displays an error message and leaves the linked output file incomplete.

C.3 Compiling a DOS Source File

You can compile a DOS C source file under XENIX by using the **-dos** option of the XENIX **cc** command. The command line has the form:

```
cc -dos options filename ...
```

where *options* are other **cc** command options (as described in Chapter 2 of the *C User's Guide*), and *filename* is the name of the source file you wish to compile. You can give more than one source file if desired. Each source filename must end with the “.c” extension.

The **cc** command compiles each source file separately, creating an object file for each file. It then links all the object files together with the appropriate C libraries. The object files created by the **cc** command have

XENIX to DOS: A Cross Development System

the same base name as the corresponding source file, but end with the “.o” extension instead of the “.c” extension. The linked program file has the name *a.out* if no name is explicitly given.

For example, the command:

```
cc -dos test.c
```

compiles the source file *test.c*, and creates the object file *test.o*. It then calls **dosld** which links the object file with functions from the DOS libraries. The resulting program file is named *a.out*.

You can use any number of **cc** options in the command line. The options work as described in Chapter 2 of the *C User's Guide*. For example, you may use the **-o** option to explicitly name the resulting program file, or the **-c** option to create object files without creating a program file. In some cases, the default values for an option are different than when compiling for XENIX. In particular, the default directory for library files given with the **-l** option is */usr/lib/dos*. Note that the **-p** (for “profiling”) option cannot be used.



C.3.1 DOS Floating Point Flags

The **-FPn** options to the C compiler are used when generating programs targeted for DOS. These five flags control how the resulting program performs floating point operations:

- | | |
|--------|--|
| -FPa | Generates subroutine calls to the “alternate” floating point library. (<i>/usr/lib/dos/[SML]dlibcfa.a</i>) This library is faster than the standard math coprocessor emulation library, but not as accurate. |
| -FPc | Generates subroutine calls (as opposed to inline code) to the coprocessor emulation library. (<i>/usr/lib/dos/em.a</i>) Programs compiled with this flag will do all floating point operations in software. |
| -FPc87 | Generates subroutine calls to the floating point library, which then uses the math coprocessor. (<i>/usr/lib/dos/87.a</i>) A coprocessor must be installed to run programs compiled with this flag. |

C Library Guide

- FPi** Generates inline code that will check to see if a math coprocessor is present, use it if one is there, or call the emulation library if one is not. This is the default for DOS, and is the only method used for performing floating point operations under XENIX.
- FPi87** Generates true inline code for the math coprocessor. A coprocessor must be installed to run a program compiled with this flag.

Again, note that programs compiled with **-FPa** or **-FPc** will ignore a math coprocessor (8087, 80287) if one is installed, and programs compiled with **-FPc87** and **-FPi87** will not run if a math coprocessor is not installed.

C.4 Using Assembly Language Source Files

You can direct **cc** to assemble XENIX assembly language source files by including the files in the **cc** command line. Like C source files, assembly language source files may contain only calls to functions in the DOS libraries. Furthermore, the source files must follow the C calling conventions described in the *Macro Assembler User's Guide*. The filename of an assembly language source file must end with the “.s” extension.

When an assembly language source file is given, **cc** automatically invokes **masm**, the 8086/80286 assembler. The assembler creates an object file that can be linked with any other object file created by **cc**.

You can invoke the assembler directly by using the **masm** command. The command creates an object file just as the **cc** command does, but does not create an executable file. For a description of the command and its options, see **masm(CP)** in the *XENIX Reference*.

C.5 Creating and Linking Object Files

You can link DOS object files previously created by **cc** or **masm** by giving the names of the files in the **cc** command line. The object files must have been created with **masm**, or with **cc** using the **-dos** option. Object files created without using the **-dos** option cannot be linked to DOS programs. The object filenames must end with the “.o” extension.

When an object file is given, **cc** automatically invokes **dosld** (the DOS linker) which links the given object files with the appropriate C libraries. If there are no errors, **dosld** creates an executable program file named *a.out*.

You can use **dosld** independently of **cc**. The command creates a DOS program file just as the **cc** command does, but does not accept source files. If it is necessary to invoke **dosld**, invoke the **cc** command with the **-z** flag to see a correct **dosld** command line. For a description of the command and its options, see **dosld(CP)** in the *XENIX Programmer's Reference*.

Note

DOS programs created by **cc** and **dosld** are completely compatible with the DOS system and can be executed on any such system. DOS programs cannot be executed on the XENIX system.



C.6 Running and Debugging a DOS Program

You can debug a DOS program by transferring the program file to a DOS system and using the DOS debugger, **DEBUG**, to load and execute the program. The following section explains how to transfer program files between systems. For a description of the **DEBUG** program, see the appropriate DOS guide.

C.7 Transferring Programs Between Systems

You can transfer programs between XENIX and DOS systems by using DOS floppy disks and the XENIX **doscp** command (see **dos (C)**). The **doscp** command lets you copy files to a DOS floppy disk.

The command has the form:

```
doscp -r file.1 dev:file.2
```

where **-r** is the required “raw” option, *file.1* is the XENIX name of the DOS program file you wish to transfer, *dev* is the full pathname of a XENIX system floppy disk drive, and *file.2* is the full DOS pathname of the new program file on the DOS disk. The new filename must have the “.exe” extension. The **-r** option ensures that the program file is copied.

C Library Guide

To transfer a XENIX program file to a DOS system, follow these steps:

1. Insert a formatted DOS diskette into a XENIX system floppy disk drive.
2. Use the **dosc**p command to copy the program file to the disk. For example, to copy the program file *a.out*, to a file renamed *test.exe* on a DOS disk in floppy drive */dev/fd0*, enter:

```
doscp -r a.out /dev/fd0:test.exe
```

3. Remove the floppy disk from the drive.

You can now insert the floppy disk into the floppy disk drive of the DOS system and invoke the program just as you would any other DOS program.

Note

DOS program files that do not end with the *.EXE* or *.COM* extension cannot be loaded for execution under DOS. When transferring program files from XENIX to DOS, you must make sure you rename *a.out* files to an appropriate *.EXE* or *.COM* file.

On some XENIX systems, you may be able to create a DOS partition on the system hard disk and copy DOS program files to this partition instead of to floppy disks. To execute the program, you must reboot the system, loading the DOS operating system from the DOS partition.

The file */etc/default/msdos* is an easily configurable file that aliases default device names used by the **dos**(C) commands. For example, it now contains the lines:

```
C=/dev/hd0d  
D=/dev/hd1d
```

Users using the **dos**(C) utilities can specify “C:” or “D:” on the command line, referring to the DOS partition on the first or second hard disk. For a complete description on using */etc/default/msdos*, see the manual page **dos**(C) in the *XENIX User's Reference* and “Using XENIX and DOS On the Same Disk” in the *XENIX System Administrator's Guide*.

C.8 Creating DOS Libraries

You can create a library of your own DOS object files by using the XENIX **ar** command. The command copies object files created by the compiler to a given archive file. The command has the form..

ar archive filename ...

where *archive* is the name of an archive file, and *filename* is the name of the DOS object file you wish to add to the library.

Note

DOS libraries created on a XENIX system are not compatible with libraries created on a DOS system.



C.9 Common Run-Time Routines

The sections below list routines from the DOS C library that are compatible with XENIX and UNIX System V routines. Routines specific to the DOS environment are also listed.

C.9.1 Common Routines for DOS and XENIX

The following is a list of the common routines for DOS and XENIX.

abort*	ctime	fprintf	isascii	modf	scanf	_tolower
abs	dup	fputc	iscntrl	open*	stat*	_toupper
access*	dup2	fputs	isdigit	perror	strcat	umask*
acos†	ecvt	fread*	isgraph	pow†	strchr	ungetc
asctime	execl*	free	islower	printf	strcmp	unlink*
asin†	execle*	freopen*	isprint	putc	strcpy	utime*
assert	execlp*	frexp	ispunct	putchar	strcspn	write*
atan†	execv*	fscanf	isspace	putenv	strdup	
atan2†	execve*	fseek*	isupper	puts	strlen	
atof	execvp	fstat*	isxdigit	putw	strncat	
atoi	exit	ftell*	ldexp†	qsort	strncmp	

C Library Guide

atol	exp	ftime*	localtime	rand	strncpy
bessel†,††	fabs	fwrite*	locking*	read*	strpbrk
bsearch	fclose	gcvt	log†	realloc	strchr
cabs	fcvt	getc	log10†	rewind	strspn
calloc	fdopen	getchar	longjmp	sbrk	strtok
ceil	feof	getcwd	lseek*	scanf	swab
chdir*	error	getenv	malloc	setbuf	system*
chmod*	fflush	getpid*	matherr	setjmp	tan†
chsize	fgetc	gets	memcpy	signal*	tanh†
clearerr	fgets	getw	memchr	sin†	time
close	fileno	gmtime	memcpy	sinh†	toascii
cos†	floor	hypot	memcpy	sprintf	tolower
cosh†	fmod	isalnum	memset	sqrt†	toupper
creat*	fopen*	isalpha	mktemp	srand	tzset

* Operates differently or has a different meaning under DOS than under XENIX.

† Implements UNIX System V-style error returns.

†† Doesn't correspond to a single function, but to six functions named **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**.

C.9.2 Common Routines for DOS and UNIX System V

The XENIX-compatible routines listed in the previous section are also compatible with the routines by the same names in UNIX System V environments.

Note that most of the math functions in the DOS library implement error handling in the same manner as the UNIX System V routines by the same name. The math routines marked with a dagger (†) in the list are common routines for DOS and XENIX that implement System V-style error handling. See Section C.9.1.

C.9.3 Routines Specific to DOS

The routines listed below are only available in the DOS C library. Programmers who are writing code to be ported to XENIX systems should avoid using these routines:

bdos	flushall	isatty	segread	strnset
cgets	FP_OFF	itoa	setmode	strrev
cprintf	FP_SEG	kbhit	sopen	strset
cputs	fputc	labs	spawnl	strupr

XENIX to DOS: A Cross Development System

cscanf	getch	ltoa	spawnle	tell
dosexterr	getche	mkdir	spawnlp	ultoa
eof	inp	movedata	spawnv	ungetch
_exit	int86	outp	spawnve	
_fcloseall	int86x	putch	spawnvp	
fgetchar	intdos	rename	strcmpi	
filelength	intdosx	rmdir	strlwr	

Section (DOS) in the XENIX *Reference* contains pages describing these routines. Refer to this section for more details on specific routines.

C.10 Common System-Wide Variables



The sections below list system-wide variables that are used in the DOS C library, as well as in XENIX and UNIX environments.

These variables are set either by the super-user or the XENIX kernel (with the exception of the **environ** variable). Although they can be referenced, they cannot be altered.

The variables specific to the DOS environment are also listed.

C.10.1 Common Variables for DOS and XENIX

The following is a list of system-wide variables used in the run-time library and available in both the DOS and XENIX environments:

daylight	environ	errno	sys_errlist
sys_nerr	timezone	tzname	

Note

Not all values of **errno** available on XENIX are used by the DOS run-time library.

C Library Guide

C.10.2 Common Variables for DOS and UNIX System V

The XENIX-compatible system-wide variables listed in Section A.10.1 are also available in UNIX System V environments. There are no additional common variables for DOS and UNIX System V.

C.10.3 Variables Specific to DOS

The following global variables are available only in the DOS C library. Programmers who are writing code to be ported to XENIX systems should avoid using these variables:

`_doserrno`
`_fmode`
`_osmajor`
`_osminor`
`_psp`

C.11 Common Include Files

Structure definitions, return value types, and manifest constants used in the descriptions of some of the common routines, may vary from environment to environment and are therefore fully defined in a set of include files for each environment. Include files provided with the DOS C library are compatible with include files by the same names on XENIX and UNIX systems. Some additional include files are compatible with include files by the same name in UNIX System V environments.

Sections C.11.1 and C.11.2 list the DOS include files that are compatible with XENIX and UNIX System V. The include files that apply only to DOS environments are listed in section C.11.3.

C.11.1 Common Include Files for DOS and XENIX

The following DOS include files are compatible with the XENIX (and UNIX) include files by the same name:

<i>assert.h</i>	<i>setjmp.h</i>	<i>sys/stat.h</i>
<i>ctype.h</i>	<i>signal.h</i>	<i>sys/timeb.h</i>
<i>errno.h</i>	<i>stdio.h</i>	<i>sys/types.h</i>
<i>fcntl.h</i>	<i>time.h</i>	
<i>math.h</i>	<i>sys/locking.h</i>	

C.11.2 Common Include Files for DOS and UNIX System V

The XENIX-compatible include files listed in section C.11.1 are also compatible with the include files by the same names in UNIX System V environments. In addition, the names of the following DOS include files correspond to UNIX System V include files; however, the DOS include files may not contain all the constants and types defined in the corresponding UNIX System V include files.

malloc.h
memory.h
search.h
string.h



C.11.3 Include Files Specific to DOS

The following include files are used only in DOS environments and do not have counterparts on XENIX and UNIX systems.

<i>conio.h</i>	<i>io.h</i>	<i>stdlib.h</i>
<i>direct.h</i>	<i>process.h</i>	<i>sys/utime.h</i>
<i>dos.h</i>	<i>share.h</i>	<i>v2tov3.h</i>

C.12 Differences Between Common Routines

Sections C.12.1 through C.12.25 explain how the DOS routines in the common library for XENIX and DOS differ from their XENIX counterparts. These descriptions are intended to be used in conjunction with the more detailed descriptions of DOS and XENIX routines in the *XENIX Reference*.

C.12.1 abort

The DOS version of the **abort** routine terminates the process by a call to an exit routine rather than through a signal. Control is returned to the parent (calling) process with an exit status of 3 and the message:

Abnormal program termination

is sent to standard error. No core dump occurs on DOS.

C Library Guide

C.12.2 access

The **access** routine checks the access to a given file. Under DOS, the real and effective user IDs are non-existent. The permission (access) setting can be any combination of the following values.

Value	Meaning
04	Read
02	Write
00	Check for existence

The “Execute” access mode (01) is not implemented.

In case of error, only the **EACCES** and **ENOENT** values may be returned for **errno** on DOS.

C.12.3 chdir

In case of error, only the **ENOENT** value may be returned for **errno** on DOS.

C.12.4 chmod

The **chmod** routine can set the “owner” access permissions for a given file, but all other permission settings are ignored. The mode argument can be any one of the constant-expressions shown in the left column below; the equivalent XENIX value is shown in the right column.

Constant-Expression	Meaning	XENIX Value
S_IREAD	Read by owner	0400
S_IWRITE	Write by owner	0200
S_IREAD S_IWRITE	Read and write by owner	0000

The **S_IREAD** and **S_IWRITE** constants are defined in the *sys\stat.h* include file. Note that the OR operator (**|**) is used to combine these constants to form read and write permission.

XENIX to DOS: A Cross Development System

If write permission is not given, the file is treated as a read-only file. Giving write-only permission is allowed, but has no effect; under DOS, all files are readable.

In case of error, only the **ENOENT** value may be returned for **errno** on DOS.

C.12.5 creat

The **creat** routine creates a new file or prepares an existing file for writing. If the file is created, the access permissions are set as defined by the mode argument. Only “owner” permissions are allowed (see **chmod** above).



In case of error, only the **EACCES**, **EMFILE**, and **ENOENT** values may be returned for **errno** on DOS.

Use of the **open** routine is preferred over **creat** when creating or opening files in both DOS and XENIX environments.

C.12.6 exec

The DOS versions of the **execl**, **execle**, **execlp**, **execv**, **execve**, and **execvp** routines overlay the calling process, as in the XENIX environment. If there is not enough memory for the new process, the **exec** routine will fail and return to the calling process. Otherwise, the new process begins execution.

Under DOS, the **exec** routines *do not*:

- Use the close-on-exec flag to determine open files for the new process.
- Disable profiling for the new process (profiling is not available under DOS).
- Pass on signal settings to the child process. Under DOS, all signals (including signals set to be ignored) are reset to the default in the child process.

The combined size of all arguments (including the program name) in an **exec** routine under DOS must not exceed 128 bytes.

In case of error, the **E2BIG**, **EACCES**, **ENOENT**, **ENOEXEC**, and **ENOMEM** values may be returned for **errno** on DOS. In addition, the

C Library Guide

EMFILE value may be used; under DOS, the file must be opened to determine whether it is executable.

C.12.7 **fopen**, **freopen**

The DOS versions of the **fopen** and **freopen** routines open stream files just as they do in the XENIX environment. However, under DOS the following additional values for the *type* string are available.

Value	Meaning
t	Opens the file in text mode. Opening a file in this mode causes translation of carriage return/linefeed (CR-LF) character combinations into a single linefeed (LF) on input. Similarly, on output, linefeeds are translated into CR-LF combinations.
b	Opens the file in binary mode. This mode suppresses translation.

See the DOS reference pages **fopen(DOS)** and **freopen(DOS)** in the *XENIX Programmer's Reference* for more information on their default settings.

The DOS and XENIX versions of these routines also differ in their interpretation of append mode (“a” or “a+”). When append mode is specified in the DOS version of **fopen** or **freopen**, the file pointer is repositioned to the end of the file before any write operation. Thus, all write operations take place at the end of the file.

In the XENIX versions, all write operations take place at the current position of the file pointer. In append mode, the file pointer is initially positioned at the end of the file, but if the file pointer is later repositioned, write operations take place at the new position rather than at the end of the file.

C.12.8 **fread**

The DOS **fread** routine uses the low-level **read** function to carry out read operations. If the file has been opened in text mode, **read** replaces each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the the CR-LF pairs have been replaced. Thus, the return value may not always correspond to the actual number of bytes read. This is considered normal and has no

implications for detecting the end of the file.

C.12.9 fseek

The DOS version of the **fseek** routine moves the file pointer to the given position, just as in the XENIX environment. However, for streams opened in text mode, **fseek** has limited use because carriage return-linefeed translations can cause **fseek** to produce unexpected results. The only **fseek** operations guaranteed to work on streams opened in text mode are: seeking with an offset of zero relative to any of the origin values, or seeking from the beginning of the file with an offset value returned from a call to **ftell**.



C.12.10 fstat

DOS does not make as much information available for file handles as it does for full pathnames; thus, the DOS version of **fstat** returns less useful information than the **stat** routine. The DOS **fstat** routine can detect device files, but it must not be used with directories.

The structure returned by **fstat** contains the following members.

Member	Meaning
st_mode	User read and write bits reflect the file's permission setting. The S_IFCHR bit is set for a device; otherwise, the S_IFREG bit is set.
st_ino	Not used.
st_dev	Either drive number of the disk containing the file, or the file handle in the case of a device.
st_rdev	Either drive number of the disk containing the file, or the file handle in the case of a device.
st_nlink	Always 1.
st_uid	Not used.
st_gid	Not used.

C Library Guide

st_size	Size of the file in bytes.
st_atime	Time of last modification of file.
st_mtime	Time of last modification of file (same as st_atime).
st_ctime	Time of last modification of file (same as st_atime and st_mtime).

In case of error, only the **EBADF** value may be returned for **errno** on DOS.

C.12.11 ftell

The DOS version of the **ftell** routine gets the current file pointer position, just as in the XENIX environment. However, for streams opened in text mode, the value returned by **ftell** may not reflect the physical byte offset, since text mode causes carriage return-linefeed translation. The **ftell** routine can be used in conjunction with the **fseek** routine to remember and return to file locations correctly.

C.12.12 ftime

Unlike the system time on XENIX systems, the DOS system time does not include the concept of a default time zone. Instead, **ftime** uses the value of a DOS environment variable named **TZ** to determine the time zone. The user can set the default time zone by setting the **TZ** variable. If **TZ** is not explicitly set, the default time zone corresponds to the Pacific Time Zone. See the reference page for **ctime(S)** in the *XENIX Programmer's Reference* for details on the **TZ** variable.

C.12.13 fwrite

The DOS **fwrite** routine uses the low-level **write** function to carry out write operations. If the file was opened in text mode, every linefeed (LF) character in the output is replaced by a carriage return-linefeed (CR-LF) pair before being written. This does not affect the return value.

C.12.14 `getpid`

The `getpid` routine returns a process-unique number. Although the number may be used to uniquely identify the process, it does not have the same meaning as the process ID returned by `getpid` in the XENIX environment.

C.12.15 `locking`

The DOS and XENIX versions of the `locking` routine differ in several respects, as listed below.

1. Under DOS, it is not possible to lock a file only against write access; locking a region of a file prevents both reading and writing in that region. This means that setting `LK_RLCK` in the `locking` call is equivalent to setting `LK_LOCK`, and setting `LK_NBRLCK` is equivalent to setting `LK_NBLCK`.
2. On DOS, specifying `LK_LOCK` or `LK_RLCK` will *not* cause a program to wait until the specified region of a file is unlocked. Instead, up to ten attempts are made to lock the file (one attempt per second). If the lock is still unsuccessful after 10 seconds, the `locking` function returns an error value. On XENIX, if the first attempt at locking fails, the locking process “sleeps” (suspends execution) and periodically “wakes” to attempt the lock again. There is no limit on the number of attempts, and the process can continue indefinitely.
3. On DOS, locking of overlapping regions of a file is not allowed.
4. On DOS, if more than one region of a file is locked, only one region can be unlocked at a time, and the region must correspond to a region that was previously locked. You cannot unlock more than one region at a time, even if the regions are adjacent.

C.12.16 `lseek`

In case of error, only the `EBADF` and `EINVAL` values may be returned for `errno` on DOS.



C Library Guide

C.12.17 open

The **open** routine opens a file handle for a named file, just as in the XENIX environment. However, two additional *oflag* values (**O_BINARY** and **O_TEXT**) are available and the **O_NDELAY** and **O_SYNCW** values are not available.

The **O_BINARY** flag causes the file to be opened in binary mode, regardless of the default mode setting. Similarly, the **O_TEXT** flag causes the file to be opened in text mode.

In case of error, only the **EACCES**, **EEXIST**, **EMFILE**, and **ENOENT** values may be used for **errno** on DOS.

C.12.18 read

The DOS version of the **read** routine reads characters from the file given by a file handle, just as in the XENIX environment. However, if the file has been opened in text mode, **read** replaces each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the the CR-LF pairs have been replaced. Thus, the return value may not always correspond with the actual number of bytes read. This is considered normal and has no implications for detecting an end-of-file condition.

In case of error, only the **EBADF** value may be used for **errno** on DOS.

C.12.19 signal

The DOS version of the **signal** routine can only handle the **SIGINT** signal. In DOS, **SIGINT** is defined to be INT 23H (the CONTROL-C signal).

On DOS, child processes executed through the **exec** or **spawn** routines do not inherit the signal settings of the parent process. All signal settings (including signals set to be ignored) are reset to the default settings in the child process.

The DOS version of **signal** uses only the **EINVAL** for **errno**.

C.12.20 `stat`

The `stat` routine returns a structure defining the current status of the given file or directory. The structure members returned by `stat` have the following names and meanings on DOS.

Value	Meaning
<code>st_mode</code>	User read and write bits reflect the file's permission setting. The <code>S_IFDIR</code> bit is set for a device; otherwise, the <code>S_IFREG</code> bit is set.
<code>st_ino</code>	Not used.
<code>st_dev</code>	Drive number of the disk containing the file.
<code>st_rdev</code>	Drive number of the disk containing the file.
<code>st_nlink</code>	Always 1.
<code>st_uid</code>	Not used.
<code>st_gid</code>	Not used.
<code>st_size</code>	Size of the file in bytes.
<code>st_atime</code>	Time of last modification of file.
<code>st_mtime</code>	Time of last modification of file (same as <code>st_atime</code>).
<code>st_ctime</code>	Time of last modification of file (same as <code>st_atime</code> and <code>st_mtime</code>).

In case of error, only the `ENOENT` value may be returned for `errno` on DOS.



C Library Guide

C.12.21 system

The **system** routine passes the given string to the operating system for execution. For DOS to execute this string, the full pathname of the directory containing COMMAND.COM must be assigned to the COMSPEC or PATH environment variable. The **system** call returns an error if COMMAND.COM cannot be found using these variables.

In case of error, only the **E2BIG**, **ENOENT**, **ENOEXEC** and **ENOMEM** values may be returned for **errno** on DOS.

C.12.22 umask

The **umask** routine can set a mask for “owner” read and write access permissions only. All other permissions are ignored. (See the discussion of the **access** routine above for details.)

C.12.23 unlink

The DOS version of the **unlink** routine always deletes the given file. Since DOS does not implement multiple “links” to the same file, unlinking a file is the same as deleting it.

In case of error, only the **EACCES** and **ENOENT** values may be returned for **errno** on DOS.

C.12.24 utime

The DOS **utime** routine sets the file modification time only; DOS does not maintain a separate access time.

In case of error, the **EACCES** and **ENOENT** values may be returned for **errno** on DOS. In addition, the **EMFILE** value may be used; under DOS, the file must be opened to set the modification time.

C.12.25 write

The **write** routine writes a specified number of characters to the file named by the given file handle, just as in the XENIX environment. However, if the file has been opened in text mode, every linefeed (LF) character in the output is replaced by a carriage return-linefeed (CR-LF) pair before being written. This does not affect the return value.

XENIX to DOS: A Cross Development System

In case of error, only the **EBADF** and **ENOSPC** values may be returned for **errno** on DOS.

C.13 Differences in Definitions

Many of the special definitions given in *intro(S)* in the *XENIX Programmer's Reference* do not apply to the common routines when used in the DOS environment. The following is a list of the differences.

The *process ID* is still a unique integer, but does not have the same meaning as in the XENIX environment.



The *parent process*, *process group*, *tty group*, *real user*, *real group*, *effective user* and *effective group* IDs are not used by the common routines when run under DOS. Furthermore, there is no *super-user* or *special processes* in the DOS environment.

The *filenames* in DOS have two parts: a filename and a filename extension. Filenames may be any combination of up to eight letters or digits. Filename extensions may be any combination of up to three letters or digits, preceded by a period (.).

The *pathnames* in DOS may be any combination of directory names separated by a backslash (\). The slash (/) used in the XENIX environment is not allowed unless the user has redefined the leading character used with options in DOS command lines (this character is initially the slash). Directory names may be any combination of up to eight letters or digits. The special names “.” and “..” refer to the current directory and the parent directory, respectively.

The *drive names* may be used at the begin of a pathname to specify a specific disk drive or device. Drives names are generally a letter or combination of letters and digits followed by a colon (:).

The *access permissions* in DOS are restricted to read and write by the owner of the file. Since all users own all files in DOS, access permissions do little more than define whether or not the file is a read-only file or can be modified. Execution permission and other permissions defined for files in the XENIX environment do not apply the files in the DOS environment.

Replace this Page
with Tab Marked:

Index



Index

> (redirection symbol) 6-3
{ } (braces) 1-4
[] (brackets) 1-4
| (bar) 1-4
/ (forward slash), pathname delimiter, used as
 2-7
| (pipe symbol) 4-15, 6-3
< (redirection symbol) 6-3

A

a64l 4-3
abort
 described 4-16
 differences from XENIX version C-11
 XENIX version, differences from B-7
abs
 described 4-11
access
 described 4-5
 differences from XENIX version C-12
 XENIX version, differences from B-7
acct routine 4-26
acct.h include file 5-9
acos
 described 4-11
 floating-point support 2-8
addch 4-18
addch function 7-3, 7-11
Adding
 characters 7-11
 strings 7-12
 windows 7-22
addstr 4-18
addstr function 7-12
alarm 4-16
Allocation *See* Memory allocation
ANSI
 include files B-7
 run-time library B-4
ANSI C
 compatibility with 1-1
a.out.h include file 5-8
Archive file 5-1
Archive number 5-1

argc, argument count variable
 defining 6-4
 described 6-4
Argument type
 checking 2-5
 lists 2-5
Arguments
 macros, with side effects 2-3
 variable number 2-5
argv, argument value array
 defining 6-4
 described 6-4
ar.h include file 5-1
asctime 4-27
asin
 described 4-11
 floating-point support 2-8
assert 4-28
assert.h include file 5-1, 5-9
atan
 described 4-11
 floating-point support 2-8
atan2
 described 4-11
 floating-point support 2-8
atof 4-3
 floating-point support 2-8
atoi 4-3
atol 4-3

B

Backslash (\)
 pathname delimiter, used as 2-7
Bar (|) 1-4
Bessel functions
 described 4-11
 floating-point support 2-8
Bold font 1-4
bool flag 7-5
box 4-18
box function 7-35
Braces ({ }) 1-4
Brackets ([]) 1-4
brk.h include file 5-9
bsearch 4-22
BSIZE, buffer size value 6-1
Buffer manipulation
 include file 4-1
 memcpy 4-1

Index

Buffer manipulation (*continued*)

- memchr 4-1
- memcmp 4-1
- memcpy 4-1
- memset 4-1
- routines 5-5

Buffered I/O

- character pointer 6-34
- creating 6-26
- described 6-24
- flushing a buffer 6-27
- returning a character 6-26

buf.h include file 5-9

Bytes

- reading from a file 6-30
- reading from a pipe 11-4
- writing to a file 6-30
- writing to a pipe 11-4

C

cabs

- described 4-11
- floating-point support 2-8

cabs function A-8

calloc 4-13

calloc function 12-3

callo.h include file 5-9

Capital letters

- small 1-4
- use of 1-4

Case sensitivity

- C language 2-7
- XENIX 2-7

CBREAK mode

- automatic setting 7-13, 7-14, 7-15, 7-26
- clearing the terminal 7-41
- terminal configuration 7-41

ceil

- floating-point support 2-8

char type 7-5, 7-35

Character classification and conversion

- isalnum 4-2
- isascii 4-2
- isgraph 4-2
- islower 4-2

Character, classification and conversion

- isprint 4-2
- ispunct 4-2
- isspace 4-2
- isupper 4-2
- isxdigit 4-2

Character, classification and conversion

(*continued*)

- toascii 4-2
- tolower 4-2
- _toupper 4-2

Character functions, described 8-1

Character pointer

- described 6-34
- moving 6-34, 6-35
- moving to start 6-36
- reporting position 6-36

Characters

- adding 7-11
- alphabetic 8-3
- alphanumeric 8-3
- ASCII 8-1
- bold display 7-35
- control 8-4
- converting to
 - ASCII 8-2
 - lowercase 8-7
 - uppercase 8-7
- decimal digits 8-4
- deleting 7-28
- hexadecimal digit 8-5
- inserting 7-16, 7-27
- lowercase 8-6
- printable 8-5, 8-6
- printing 7-12
- processing, described 8-1
- punctuation 8-5
- reading 7-13, 7-15, 7-24, 7-33
- reading from
 - a file 6-15
 - standard input 6-6
- restoring normal 7-36
- uppercase 8-6
- writing to
 - a file 6-18
 - standard output 6-9

chdir 4-5

- differences from XENIX version C-12
- XENIX version, differences from B-8

Child process, described 9-7

chmod

- described 4-5
- differences from XENIX version C-12
- XENIX version, differences from B-8

chown 4-5

chroot 4-5

chsize 4-5

clear 4-18

clear flag

- clearing 7-9

- clear flag (*continued*)
 - clear-screen sequence 7-39
 - function set 7-29
 - program example 7-18
- clear function 7-3, 7-18
- clearerr 2-6, 4-8
- Clearing
 - screens 7-18, 7-19, 7-28
 - terminal modes 7-41
- clearok 4-18, 4-19
- clearok function 7-38, 7-39
- clear-screen flag 7-38
- clock 4-27
- clock_t type 3-3
- close 4-10
 - function 6-31
- closedir routine 4-4
- clrtoob 4-18, 4-19
- clrtoob function 7-19
- clrtoeol 4-18, 4-19
- clrtoeol function 7-19
- COLS variable 7-5, 7-10
- Command line
 - arguments, storage order 6-4
 - described 6-4
- Commands
 - dbm 4-3
 - notational conventions 1-4
 - termcap 7-36
- Common library
 - common routines, listed B-1
 - global variables B-5
 - include files B-6, C-10
 - listing of common routines C-7, C-8
 - run-time routine, differences B-7
 - run-time routines
 - differences C-11
 - system-wide variables C-9
- Compatibility
 - differences, listed B-7
 - differences listed C-11
 - global variables B-5
 - include files B-6, C-10
 - math routines B-1
 - run-time routines B-1, C-7
 - system-wide variables C-9
 - UNIX and XENIX B-1
 - XENIX and ANSI C 1-1
 - XENIX and MS-DOS 1-1
 - XENIX and UNIX 1-1
- conf.h include file 5-9
- Conventions, notational 1-4
- Conversion
 - time 5-7
- Core file 5-2
- core.h include file 5-2
- cos
 - described 4-11
 - floating-point support 2-8
- cosh
 - described 4-11
 - floating-point support 2-8
- cosh function A-8
- creat
 - described 4-10
 - differences from XENIX version C-13
 - XENIX version, differences from B-8
- Creating
 - subwindows 7-21
 - windows 7-20
- creatsem 4-23
- creatsem function 12-51
- crmode 4-18, 4-19
- crmode function 7-40, 7-41
- Cross development system
 - assembly language source files C-4
 - compiling a DOS source file C-2
 - creating a DOS program C-5
 - creating an DOS library C-7
 - creating object files C-4
 - creating source files C-2
 - dos option C-1
 - dosld commands C-1
 - elements of C-1
 - from XENIX to DOS C-5
 - linking object files C-4
 - running a DOS program C-5
 - DOS-specific routines C-8
 - transfer of files C-1
 - transferring programs C-5
- ctermid 4-28
- ctime 4-27
- ctype.h file 8-1
- ctype.h include file 5-2
- Current screen 7-3
- cursor pointer 7-3, 7-5, 7-31
- curses library 7-1, 7-3
- curses.h include file 5-2, 7-4
- cursor flag 7-9, 7-38
- Cursor movement 7-1, 7-42
- cuserid routine 4-26

Index

D

Data conversion

- a64l 4-3
- atof 4-3
- atoi 4-3
- atol 4-3
- ecvt 4-3
- fcvt 4-3
- gcvt 4-3
- include files 4-3
- itoa 4-3
- l3tol 4-3
- l64a 4-3
- ltoa 4-3
- ltol3 4-3
- sgetl 4-3
- sputl 4-3
- strtol 4-3
- strtoul 4-3
- ultoa 4-3

Data Stream functions

- accessing files 6-12
- accessing standard files 6-13
- described 6-12
- file pointers 6-13
- random access 6-34

Database, large 5-2

Database-manipulation routines

- dbminit 4-3
- delete 4-3
- fetch 4-3
- firstkey 4-3
- nextkey 4-3
- store 4-3

daylight variable 3-1

dbm command 4-3

dbm.h include file 5-2

dbminit 4-3

Debugging, restrictions 6-3

Declarations, function *See* Function declarations

Default

- terminal modes 7-9
- terminal size 7-10
- window flags 7-9

Definitions

- constant 5-1
- macro 5-1
- structure 5-1
- type 5-1

defopen 4-28

defread 4-29

Def_term variable 7-5, 7-8

delch 4-18, 4-19

delch function 7-17

delete 4-3

deleteln 4-18, 4-19

deleteln function 7-3, 7-18

Deleting

- characters 7-28
- lines 7-18, 7-28
- windows 7-34

Delimiters for pathname components *See*

Pathnames

delwin 4-18, 4-19

delwin function 7-34

Directory names, notational conventions 1-4

Directory operation routines

- closedir 4-4
- descriptions 4-4
- opendir 4-4
- readdir 4-4
- rewinddir 4-4
- seekdir 4-4
- telldir 4-4

dir.h include file 5-10

dmpwin function 7-30

drand48 routine 4-18

dumprestor.h include file 5-2

Dumps 5-2

dup

- described 4-10

dup function 11-6

dup2

- described 4-10

E

\(backslash)

- pathname delimiter, used as 2-7

E2BIG errno value A-2

EACCESS errno value A-3

EAGAIN errno value A-3

EBADF errno value A-2

EBUSY errno value A-3

ECHILD errno value A-3

echo 4-18, 4-19

echo function 7-40, 7-41

ECHO mode

- clearing the mode 7-41
- copying to the standard screen 7-13, 7-14, 7-15, 7-26

- terminal configuration 7-9, 7-41

ecvt 4-3

EDEADLOCK errno value A-6

- EDOM errno value A-6, A-8
- EEXIST errno value A-4
- EFAULT errno value A-3
- EBIG errno value A-5
- EIDRM errno value A-7
- EINTR errno value A-2
- EINVAL errno value A-4
- EIO errno value A-2
- EISDIR errno value A-4
- EISNAM errno value A-7
- Ellipses, use of 1-4
- EMFILE errno value A-5
- EMLINK errno value A-5
- ENAVAIL errno value A-6
- endgrent routine 4-6
- End-of-file
 - condition 2-7
 - testing 6-21
 - value, EOF 6-1
- endpwent routine 4-6
- endutent routine 4-26
- endwin 4-18, 4-19
- endwin function 7-7, 7-10, 7-43
- ENFILE errno value A-4
- ENODEV errno value A-4
- ENOENT errno value A-1
- ENOEXEC errno value A-2
- ENOLCK errno value A-7
- ENOMEM errno value A-3
- ENOMSG errno value A-7
- ENOSPC errno value A-5
- ENOTBLK errno value A-3
- ENOTDIR errno value A-4
- ENOTNAM errno value A-6
- ENOTTY errno value A-5
- environ variable 3-2
- Environment
 - variable names, notational conventions 1-4
- Environment table
 - described 4-30
- ENXIO errno value A-2
- eof 2-7, 4-10
- EOF, end-of-file value 6-1
- EPERM errno value A-1
- EPIPE errno value A-6
- erand48 routine 4-18
- ERANGE errno value A-6, A-8
- erase 4-18, 4-19
- erase function 7-18
- erf
 - described 4-11
- erfc
 - described 4-11
- EROFS errno value A-5
- ERR 7-5
- errno
 - values
 - description A-1
 - E2BIG A-2
 - EACCES A-3
 - EAGAIN A-3
 - EBADF A-2
 - EBUSY A-3
 - ECHILD A-3
 - EDEADLOCK A-6
 - EDOM A-6, A-8
 - EEXIST A-4
 - EFAULT A-3
 - EBIG A-5
 - EIDRM A-7
 - EINTR A-2
 - EINVAL A-4
 - EIO A-2
 - EISDIR A-4
 - EISNAM A-7
 - EMFILE A-5
 - EMLINK A-5
 - ENAVAIL A-6
 - ENFILE A-4
 - ENODEV A-4
 - ENOENT A-1
 - ENOEXEC A-2
 - ENOLCK A-7
 - ENOMEM A-3
 - ENOMSG A-7
 - ENOSPC A-5
 - ENOTBLK A-3
 - ENOTDIR A-4
 - ENOTNAM A-6
 - ENOTTY A-5
 - ENXIO A-2
 - EPERM A-1
 - EPIPE A-6
 - ERANGE A-6, A-8
 - EROFS A-5
 - ESPIPE A-5
 - ESRCH A-2
 - ETXTBSY A-5
 - EUCLEAN A-6
 - EXDEV A-4
 - list A-1
 - variables A-1
 - errno variable
 - described 2-6, 3-2
 - error numbers 4-30
 - using 2-7
 - errno.h include file 5-3, 5-10, A-1
 - Error messages

Index

Error messages (*continued*)

- file 5-3
- math A-8
- system A-1

Errors

- handling
 - providing for 2-6
 - stream operations 2-6
- indicator
 - error, with 2-6
- returns 2-6
- testing files 6-21

ESPIPE errno value A-5

ESRCH errno value A-2

/etc/termcap file 7-2

ETXTBSY errno value A-5

EUCLEAN errno value A-6

example program, **msgctl** 12-35

example program, **msgget** 12-31

example program, **msgop** 12-42

example program, **semctl** 12-71

example program, **semget** 12-67

example program, **semop** 12-82

example program, **shmctl** 12-107

example program, **shmget** 12-103

example program, **shmop** 12-114

EXDEV errno value A-4

exec family

- described 4-16
- differences from XENIX version C-13
- XENIX version, differences from B-9

exec function 9-3

execargs.h include file 5-3

execl

- argument-type-checking limitations 2-6
- described 4-16
- differences from XENIX version C-13
- function 9-5
- XENIX version, differences from B-9

execl

- argument-type-checking limitations 2-6
- described 4-16
- differences from XENIX version C-13
- XENIX version, differences from B-9

execlp

- argument-type-checking limitations 2-6
- described 4-16
- differences from XENIX version C-13
- XENIX version, differences from B-9

execlpe

- argument-type-checking limitations 2-6

execv

- described 4-16
- differences from XENIX version C-13

execv (*continued*)

- function 9-5
- XENIX version, differences from B-9

execvp

- described 4-16
- differences from XENIX version C-13
- XENIX version, differences from B-9

execvp

- described 4-16
- differences from XENIX version C-13
- XENIX version, differences from B-9

exit 4-16

- function 9-3

exit function

- required function 7-10

exp

- described 4-11
- floating-point support 2-8
- exp function A-8

F

fabs

- described 4-11
- floating-point support 2-8

fcntl.h include file 5-10

msgctl, using 12-34

msgget, using 12-27

msgop, using 12-40

semctl, using 12-70

semget flags 12-65

semget, using 12-63

semop, using 12-80

shmctl, using 12-106

shmget flags 12-102

shmget, using 12-100

shmop, using 12-113

fclose 4-8

- function 6-22

fclose function

- closing a file 6-2

fcntl 4-5

fcntl.h include file 5-3

fcvt 4-3

fdopen 4-8

fdopen routine

- valid pointer 6-2

feof 2-7, 4-8

- function 6-21

ferror 2-6, 4-8

- function 6-21

fetch 4-3

- fflush 4-8
 - function 6-27
- fgetc 4-8
 - function 6-15
- fgets 4-8
 - function 6-15
- FILE
 - type 3-3
- File control routines
 - group 4-6
 - password 4-6
- File descriptors
 - creating 6-29
 - described 6-28
 - freeing 6-31
 - pipes 11-1
 - predefined 6-28
- FILE, file pointer type 6-1
- File handling
 - access 4-5
 - chdir 4-5
 - chmod 4-5
 - chown 4-5
 - chsize 4-5
 - fcntl 4-5
 - fstat 4-5
 - getcwd 4-5
 - include files 4-5
 - ioctl 4-5
 - isatty 4-5
 - link 4-5
 - locking 4-5
 - mknod 4-5
 - mktemp 4-5
 - mount 4-5
 - stat 4-5
 - umask 4-5
 - umount 4-5
 - unlink 4-5
 - ustat 4-5
 - utime 4-5
- File names
 - notational conventions 1-4
- File pointers 6-2
 - creating 6-14
 - defining 6-13
 - described 6-13
 - file descriptors 6-28
 - FILE type 6-13
 - freeing 6-22
 - NULL value 6-13
 - pipes 11-1
 - predefined 6-13
 - recreating 6-25
- File routines
 - group and password control
 - endgrent 4-6
 - endpwent 4-6
 - getgrent 4-6
 - getgrgid 4-6
 - getgrnam 4-7
 - getpass 4-7
 - getpw 4-7
 - getpwent 4-7
 - getpwnam 4-7
 - getpwuid 4-7
 - putpwent 4-7
 - setgrent 4-7
 - setpwent 4-7
- FILE type 6-2
- file.h include file 5-10
- Filename conventions 2-7
- fileno 4-8
- Files
 - archive 5-1
 - buffers 6-24, 6-26, 6-27
 - closing 6-22
 - low-level access 6-31
 - core 5-2
 - /etc/termcap 7-2, 7-8
 - /etc/terminfo 7-2
 - include 5-1
 - inherited by processes 9-9
 - library
 - libcurses.a 7-4
 - libtermli.a 7-4
 - locking 12-8
 - opening 6-14
 - for low-level access 6-29
 - pipes 4-14
 - random access 6-34
 - reading
 - bytes 6-30
 - characters 6-15
 - formatted data 6-17
 - records 6-16
 - strings 6-15
 - reopening 6-25
 - standard error *See* Standard error files
 - standard input *See* Standard input files
 - standard output *See* Standard output files
 - testing
 - end-of-file condition 6-21
 - for errors 6-21
 - writing
 - bytes 6-30
 - characters 6-18
 - formatted output 6-19

Index

- Files (*continued*)
 - writing (*continued*)
 - records 6-20
 - strings 6-19
 - filsys.h include file 5-10
 - firstkey 4-3
- Flags
 - clear 7-9, 7-18, 7-29, 7-39
 - clear screen 7-38
 - cursor 7-9, 7-38
 - scroll 7-9, 7-38
 - terminal 7-43
 - window 7-38
 - window default 7-9
- Floating point
 - support 2-8
- floor
 - described 4-11
 - floating-point support 2-8
- fmod
 - described 4-11
 - floating-point support 2-8
- fopen 4-8
 - differences from XENIX version C-14
 - function 6-14
 - XENIX version, differences from B-10
- open function
 - reading and writing to a file 6-2
- open routine
 - valid pointer 6-2
- fork 4-16
 - function 9-7
- Format
 - termcap 7-9
- Formatted input
 - reading from
 - a file 6-17
 - a pipe 11-2
 - standard input 6-8
- Formatted output
 - writing to
 - a file 6-19
 - a pipe 11-2
 - standard output 6-11
- Forward slash (/), pathname delimiter, used as 2-7
- fprintf 4-8
 - argument-type-checking limitations 2-5
 - function 6-19
- fputc 4-8
 - function 6-18
- fputs 4-8
 - function 6-19
- fread 4-8
 - fread 4-8 (*continued*)
 - differences from XENIX version C-14
 - XENIX version, differences from B-10
 - fread function 6-16
 - free function 12-4
 - freopen 4-8
 - differences from XENIX version C-14
 - function 6-25
 - XENIX version, differences from B-10
 - freopen routine
 - valid pointer 6-2
 - frexp
 - described 4-11
 - floating-point support 2-8
 - fscanf 4-8
 - argument-type-checking limitations 2-5
 - function 6-17
 - fseek 4-10
 - differences from XENIX version C-15
 - function 6-35
 - XENIX version, differences from B-10
 - fstat
 - described 4-5
 - differences from XENIX version C-15
 - XENIX version, differences from B-11
 - tell 4-10
 - differences from XENIX version C-16
 - function 6-36
 - XENIX version, differences from B-12
 - time
 - described 4-27
 - differences from XENIX version C-16
 - XENIX version, differences from B-12
 - ftok 4-23
 - ftw.h include file 5-3
- Function declarations 2-5
- Functions
 - addch 4-18
 - addstr 4-18
 - box 4-18
 - clear 4-18
 - clearok 4-18
 - clrrobot 4-18
 - clrtoeol 4-18
 - crmode 4-18
 - cursor 5-2
 - declarations 5-1
 - delch 4-18
 - deleteln 4-18
 - delwin 4-18
 - echo 4-18
 - endwin 4-18
 - erase 4-18
 - exit 7-10

Functions (*continued*)

- getch 4-18
- getstr 4-18
- gettmode 4-18
- getyx 4-18
- inch 4-18
- initscr 4-18
- insch 4-18
- insertln 4-18
- leaveok 4-18
- longname 4-18
- math
 - cabs A-8
 - cosh A-8
 - exp A-8
 - hypot A-8
 - library A-8
 - log A-8
 - log10 A-8
 - pow A-8
 - sinh A-8
 - sqrt A-8
 - tanh A-8
- message control 4-14
- move 4-18
- mv 4-18
- mvcur 4-18
- mvwin 4-18
- newwin 4-18
- nl 4-18
- nocrmode 4-18
- noecho 4-18
- nonl 4-18
- noraw 4-18
- overlay 4-18
- overwrite 4-18
- pclose 4-14
- perror A-1
- pipe 4-14
 - pclose 4-15
 - pipe 4-15
 - popen 4-15
 - read 4-15
 - write 4-15
- popen 4-14
- printw 4-18
- process control
 - exec 9-3
- raw 4-18
- refresh 4-18
- resetty 4-18
- savetty 4-18
- scanw 4-18
- screen 5-2

Functions (*continued*)

- screen processing 4-18
 - addch 7-3, 7-11
 - addstr 7-12
 - box 7-35
 - clear 7-3, 7-18
 - clearok 7-38, 7-39
 - clrtobot 7-19
 - clrtoeol 7-19
 - crmode 7-40
 - delch 7-17
 - deleteln 7-3, 7-18
 - delwin 7-34
 - echo 7-40
 - endwin 7-7, 7-10, 7-43
 - erase 7-18
 - getch 7-13
 - getstr 7-14
 - gettmode 7-10, 7-43
 - getyx 7-37
 - inch 7-33
 - initscr *See* initscr function
 - insch 7-16
 - insertln 7-3, 7-17
 - leaveok 7-38
 - longname 7-44
 - move 7-3, 7-16, 7-43
 - mv prefix 7-40
 - mvcur 7-42
 - mvwin 7-32
 - newwin 7-2, 7-20
 - nl 7-40, 7-41
 - nocrmode 7-41
 - noecho 7-41
 - nonl 7-42
 - noraw 7-42
 - overlay 7-3, 7-31
 - overwrite 7-3, 7-32
 - printw 7-12
 - raw 7-40, 7-41
 - refresh *See* refresh function
 - resetty 7-43
 - savetty 7-43
 - scanw 7-15
 - scroll 7-39
 - scrollok 7-38
 - setterm 7-10, 7-43
 - standend 7-36
 - standout 7-35
 - subwin 7-2, 7-21
 - touchwin 7-34
 - waddch 7-3, 7-23
 - waddstr 7-23
 - wclear 7-28

Index

Functions (*continued*)

- screen processing 4-18 (*continued*)
 - wclrobot 7-28, 7-29
 - wclrtoool 7-28, 7-29
 - wdelch 7-28
 - wdeleteln 7-28
 - werase 7-28, 7-29
 - wgetch 7-24
 - wgetstr 7-24, 7-25
 - winch 7-33
 - winsch 7-27
 - winsertln 7-27
 - wmove 7-3, 7-26, 7-43
 - wprintw 7-23
 - wrefresh 7-3, 7-30, 7-43
 - wscanw 7-24, 7-25
 - wstandend 7-36
 - wstandout 7-36
- scroll 4-18
- scrollok 4-18
- setterm 4-18
- standend 4-18
- standout 4-18
- subwin 4-18
- touchwin 4-18
- waddch 4-18
- waddstr 4-18
- wclear 4-18
- wclrobot 4-18
- wclrtoool 4-18
- wdelch 4-18
- wdeleteln 4-18
- werase 4-18
- wgetch 4-18
- wgetstr 4-18
- winch 4-18
- window *See* Functions, screen processing
- winsch 4-18
- winsertln 4-18
- wmove 4-18
- wprintw 4-18
- wrefresh 4-18
- wscanw 4-18
- wstandend 4-18
- wstandout 4-18

Functions, advantages over macros 2-1

- fwrite 4-8
 - differences from XENIX version C-16
 - function 6-20
 - XENIX version, differences from B-12
- fxlist 4-29

G

- gamma
 - described 4-11
- gcvt 4-3
- getc 4-8
 - function 6-15
- getch 4-18, 4-19
- getch function 7-13
- getchar 4-8
- getchar function 6-6
 - reading the standard input 4-8
- getcwd 4-5
- getenv 4-29
- getgrent routine 4-6
- getgrgid routine 4-6
- getgrnam routine 4-7
- getlogin routine 4-26
- getopt 4-29
- getpass routine 4-7
- getpgrp 4-16
- getpid
 - described 4-16
 - differences from XENIX version C-16
 - XENIX version, differences from B-12
- getppid 4-16
- getpw routine 4-7
- getpwent routine 4-7
- getpwnam routine 4-7
- getpwuid routine 4-7
- gets 4-8
 - function 6-7
- gets function
 - files 4-8
- getstr 4-18, 4-19
- getstr function 7-14
- getting message queues 12-27
- gettmode 4-18, 4-19
- gettmode function 7-10, 7-43
- getuid routine 4-26
- getuline routine 4-26
- getutent routine 4-26
- getw 4-8
- getyx 4-18, 4-19
- getyx function 7-37
- Global variables
 - accessing 3-1
 - common library, used in B-5
 - daylight 3-1
 - environ 3-2
 - errno

Global variables (*continued*)

errno (*continued*)
 described 3-2
 sys_errlist
 described 3-2
 sys_nerr 3-2
 timezone 3-1
 tzname 3-1
 gmtime 4-27
 Goto, nonlocal 4-31
 Greenwich mean time 4-28
 Group files 4-6
 grp.h include file 5-3
 gsignal 4-16

H

Huge arrays, used in library functions 2-9
 Huge pointers, used in library functions 2-9
 hypot
 described 4-11
 floating-point support 2-8
 hypot function A-8

I

inch 4-18, 4-19
 inch function 7-33
 Include file
 terminfo.h 7-4
 Include files
 buffer manipulation routines, used with 4-1
 common library, used in B-6
 curses.h 7-4
 data conversion 4-3
 errno.h A-1
 file handling 4-5
 in common library C-10
 math routines 4-12
 math.h A-8
 memory allocation 4-13
 searching and sorting 4-22
 stdio.h 6-2
 string manipulation 4-24
 system
 acct.h 5-9
 a.out.h 5-8
 assert.h 5-9
 brk.h 5-9
 Include files (*continued*)
 system (*continued*)
 buf.h 5-9
 callo.h 5-9
 conf.h 5-9
 dir.h 5-10
 errno.h 5-10
 fblk.h 5-10
 file.h 5-10
 filsys.h 5-10
 inode.h 5-10
 ino.h 5-10
 iobuf.h 5-11
 ioctl.h 5-11
 ipc.h 5-11
 lock.h 5-11
 locking.h 5-11
 machdep.h 5-11
 map.h 5-11
 mmu.h 5-11
 mount.h 5-12
 msg.h 5-12
 param.h 5-12
 proc.h 5-12
 reg.h 5-12
 relysym86.h 5-13
 relysym.h 5-13
 sd.h 5-13
 sdu.h 5-13
 sem.h 5-14
 signal.h 5-14
 sites.h 5-14
 stat.h 5-14
 sysinfo.h 5-14
 sysmacros.h 5-14
 system.h 5-15
 text.h 5-15
 timeb.h 5-15
 times.h 5-15
 ttold.h 5-15
 tty.h 5-15
 types.h 5-16
 ulimit.h 5-16
 user.h 5-16
 utsname.h 5-16
 var.h 5-16
 time routines 4-27, 4-28
 user
 ar.h 5-1
 assert.h 5-1
 core.h 5-2
 ctype.h 5-2
 curses.h 5-2
 dbm.h 5-2

Index

Include files (*continued*)

- user (*continued*)
 - dumprestor.h 5-2
 - errno.h 5-3
 - execargs.h 5-3
 - fcntl.h 5-3
 - ftw.h 5-3
 - grp.h 5-3
 - macros.h 5-3
 - malloc.h 5-4
 - math.h 5-4
 - memory.h 5-5
 - mnttab.h 5-5
 - mon.h 5-5
 - pwd.h 5-5
 - regexp.h 5-5
 - sd.h 5-6
 - search.h 5-6
 - setjmp.h 5-6
 - sgtty.h 5-6
 - signal.h 5-6
 - stand.h 5-6
 - stdio.h 5-7
 - string.h 5-7
 - termio.h 5-7
 - time.h 5-7
 - unlstd.h 5-8
 - ustat.h 5-8
 - utmp.h 5-8
 - values.h 5-8
 - varargs.h 5-8
- /usr/include 5-1
- /usr/include/sys 5-8

Initializing screens 7-7

initscr 4-18, 4-19

initscr function

- clearing flags 7-9
- initializing screen processing 7-3, 7-7, 7-10
- returning TTY mode 7-43
- setting ECHO mode 7-9
- setting screen size 7-10
- terminal-capability descriptions 7-8

inode.h include file 5-10

ino.h include file 5-10

Input/Output *See* I/O

insch 4-18, 4-19

insch function 7-16

Inserting

- characters 7-16, 7-27
- lines 7-17, 7-27

insertln 4-18, 4-19

insertln function 7-3, 7-17

int 7-5

I/O

I/O (*continued*)

- low level
 - close 4-10
 - creat 4-10
 - dup 4-10
 - dup2 4-10
 - eof 4-10
 - errno, use of 2-7
 - error handling 2-7
 - fseek 4-10
 - ftell 4-10
 - lseek 4-10
 - open 4-10
 - read 4-10
 - rewind 4-10
 - write 4-10
- stream 4-8

I/O routines

- gets 4-8
- printf 4-8
- putchar 4-8
- puts 4-8
- scanf 4-8
- standard files 4-8
- stream 4-8

iobuf.h include file 5-11

ioctl 4-5

ioctl.h include file 5-11

ipc.h include file 5-11

isalnum 4-2

- function 8-3

isalpha function 8-3

isascii 4-2

isascii function 8-1

isatty 4-5

isctrl function 8-4

isdigit 4-2

isdigit function 8-4

isgraph 4-2

islower 4-2

islower function 8-6

isprint 4-2

isprint function 8-5

ispunct 4-2

ispunct function 8-5

isspace 4-2

isspace function 8-6

isupper 4-2

isupper function 8-6

isxdigit 4-2

isxdigit function 8-5

Italics 1-4

itoa 4-3

J

j0 *See* Bessel functions
 j1 *See* Bessel functions
 jmp_buf type 3-3
 jn *See* Bessel functions
 jrand48 routine 4-18

K

Key sequences, notational conventions 1-4
 kill 4-16

L

l3tol 4-3
 l64a 4-3
 lcong48 routine 4-18
 ldexp
 described 4-11
 floating-point support 2-8
 leaveok 4-18, 4-19
 leaveok function 7-38
 lfind 4-22
 libcurses.a library file 7-4
 Library
 curses 7-1, 7-3
 cursor movement 7-1
 math A-8
 screen processing 7-5
 screen updating 7-1
 terminfo 7-3
 libtermcap.a library file 7-4
 Lines
 deleting 7-18, 7-28
 inserting 7-17, 7-27
 LINES variable 7-5, 7-10
 link 4-5
 Local time corrections 3-1
 localtime 4-27
 lock 4-16
 lock.h include file 5-11
 locking
 described 4-5
 differences from XENIX version C-17
 XENIX version, differences from B-13
 Locking files
 described 12-8

Locking files (*continued*)
 preparation 12-10
 sys/locking.h file 12-9
 locking function 12-10
 locking.h include file 5-11
 log function A-8
 log10 function A-8
 Logarithmic functions
 log
 described 4-11
 floating-point support 2-8
 XENIX version, differences from B-13
 log10
 described 4-11
 floating-point support 2-8
 XENIX version, differences from B-13
 logname 4-29
 longjmp 4-29
 longname 4-18, 4-19
 longname function 7-44
 Low-level functions
 accessing files 6-28
 described 6-28
 file descriptors 6-28
 random access 6-34
 lrand48 routine 4-18
 lsearch 4-22
 lseek
 described 4-10
 differences from XENIX version C-17
 function 6-34
 XENIX version, differences from B-13
 ltoa 4-3
 ltol3 4-3

M

machdep.h include file 5-11
 Macros
 advantages over functions 2-1
 arguments with side effects 2-3
 defining 5-2, 5-3, 5-7
 notational conventions 1-4
 restrictions on use 2-2
 Macros, special I/O functions 6-3
 macros.h include file 5-3
 malloc 4-13
 UNIX System V 12-5
 XENIX 12-5
 malloc function 12-2
 malloc.h 4-13
 malloc.h include file 5-4

Index

- Manifest constants, notational conventions 1-4
- Manipulating
 - databases 4-3
- map.h include file 5-11
- Math functions A-8
- Math routines
 - routine list 5-4
- matherr 2-6, 4-11
- math.h 4-3, 4-12
- math.h include file 5-4, A-8
- memcpy 4-1
- memchr 4-1
- memcpy 4-1
- memcpy 4-1
- Memory
 - allocating arrays 12-3
 - allocating-dynamically 12-1
 - allocating variables 12-2
 - freeing allocated memory 12-4
 - reallocating 12-4
- Memory allocation 5-4
 - calloc 4-13
 - include files 4-13
 - malloc 4-13
 - sbrk 4-13
- Memory allocation functions, described 12-1
- Memory models, huge arrays and huge pointers, used with 2-9
- Memory routines *See* Shared-memory routines
- memory.h 4-1
- memory.h include file 5-5
- memset 4-1
- Message operation permissions codes 12-29
- Message-control routines
 - msgctl 4-14
 - msgget 4-14
 - msgrcv 4-14
 - msgsnd 4-14
- Miscellaneous routines
 - assert 4-28
 - ctermid 4-28
 - defopen 4-28
 - defread 4-29
 - fxlist 4-29
 - getenv 4-29
 - getopt 4-29
 - logname 4-29
 - longjmp 4-29
 - nlist 4-29
 - perror 4-29
 - putenv 4-29
 - regcmp 4-29
 - regex 4-29
 - setgid 4-29
- Miscellaneous routines (*continued*)
 - setjmp 4-29
 - setuid 4-30
 - shutdown 4-30
 - swab 4-30
 - sync 4-30
 - tmpfile 4-30
 - tmpnam 4-30
 - ttyname 4-30
 - uname 4-30
 - xlist 4-30
- mknod 4-5
- mktemp 4-5
- mmu.h include file 5-11
- mnttab.h include file 5-5
- Modes
 - CBREAK
 - automatic setting 7-13, 7-14, 7-15, 7-26
 - clearing the terminal 7-41
 - terminal configuration 7-41
 - ECHO
 - clearing the mode 7-41
 - copying to the standard screen 7-13, 7-14, 7-15, 7-26
 - terminal configuration 7-9, 7-41
 - NEWLINE 7-41, 7-42
 - NOECHO 7-13, 7-14, 7-15, 7-26
 - RAW
 - direct character input 7-41
 - direct input 7-9
 - restoring editing and signal-generating functions 7-42
 - terminal mode after reading 7-13, 7-14, 7-15, 7-26
 - terminal 7-40, 7-41, 7-43
 - terminal default 7-9
 - TTY 7-43
- modf
 - described 4-12
 - floating-point support 2-8
- mon.h include file 5-5
- monitor 4-16
- mount 4-5
- mount.h include file 5-12
- move 4-18, 4-20
- move function 7-3, 7-16, 7-43
- Moving
 - current position 7-16, 7-26
 - cursor 7-42
 - windows 7-32, 7-40
- mrnd48 routine 4-18
- MS-DOS
 - compatibility with 1-1
 - specific routines B-2

MS-DOS operating system B-1
 msgctl routine 4-14
 msgget flags 12-29
 msgget routine 4-14
 msg.h include file 5-12
 msgrcv routine 4-14
 msgsnd routine 4-14
 mv 4-18, 4-20
 mv function prefix 7-40
 mvcur 4-18, 4-20
 mvcur function 7-42
 mvwin 4-18, 4-20
 mvwin function 7-32
 My_term variable 7-5, 7-9

N

nap 4-16
 nbwaitsem 4-23
 nbwaitsem function 12-54
 NEWLINE mode 7-41, 7-42
 newwin 4-18, 4-20
 newwin function 7-2, 7-20
 nextkey 4-3
 nice 4-16
 nl 4-18, 4-20
 nl function 7-40, 7-41
 nlist 4-29
 nocrmode 4-18, 4-19
 nocrmode function 7-41
 noecho 4-18, 4-19
 noecho function 7-41
 NOECHO mode 7-13, 7-14, 7-15, 7-26
 nonl 4-18, 4-20
 nonl function 7-42
 Nonlocal goto 4-31
 noraw 4-18, 4-20
 noraw function 7-42
 Notational conventions 1-4
 nrand48 routine 4-18
 NULL, null pointer value 6-1
 Numbers
 printing 7-12
 reading 7-15, 7-24

O

OK flag 7-5
 open

open (*continued*)
 argument-type-checking limitations 2-5
 described 4-10
 differences from XENIX version C-17
 function 6-29
 XENIX version, differences from B-14
 opendir routine 4-4
 opensem 4-23
 opensem function 12-52
 Optional fields, notational conventions 1-4
 overlay 4-18, 4-20
 overlay function 7-3, 7-31
 overwrite 4-18, 4-20
 overwrite function 7-3, 7-32

P

param.h include file 5-12
 Parent process, described 9-7
 Password files 4-6
 Path names
 notational conventions 1-4
 Pathnames
 conventions 2-7
 delimiters 2-7
 pause 4-16
 pclose 4-14
 function 11-3
 pclose function 4-15
 permission data structure 12-26
 perror 2-6, 4-29
 perror function A-1
 pipe 4-14
 function 11-3, 11-8
 pipe functions
 access to a process 4-15
 standard library 4-15
 Pipe symbol (|) 4-15, 6-3
 Pipes
 closing 11-3
 closing low-level access 11-5
 described 11-1
 description 4-14
 file
 descriptor 11-3
 descriptors 11-1
 pointer 11-1
 pointers 11-1
 low-level between processes 11-6
 named pipes 11-8
 opening for low-level access 11-3
 opening to a new process 11-1

Index

Pipes (*continued*)

- pclose 4-14
- pipe 4-14
- popen 4-14
- process ID 11-1
- reading bytes 11-4
- reading from 11-2
- redefining standard input and output 6-3
- shell pipe symbol 11-1
- writing bytes 11-4
- writing to 11-2

Placeholders 1-4

Pointers

- current position 7-37
 - file 6-2
 - moving 7-16
 - stderr 6-2
 - stdin 6-2
 - stdout 6-2
- popen 4-14
- function 11-1
- popen function
- opening a pipe process 4-15

Portability 2-7

Portability *See* Compatibility.

pow

- described 4-12
 - floating-point support 2-8
- pow function A-8

Predefined

- types *See* Standard types

printf 4-8

- argument-type-checking limitations 2-5
- family, floating-point support 2-8
- function 6-11

printf function

- writing to the standard output 4-8

Printing

- characters 7-12
- numbers 7-12
- strings 7-12
- windows 7-22

printw 4-18, 4-20

printw function 7-12

Process

- termination status 9-3

Process control

- abort 4-16
- alarm 4-16
- exec family 4-16
- execl 4-16
- execle 4-16
- execlp 4-16
- execv 4-16

Process control (*continued*)

- execve 4-16
- execvp 4-16
- _exit 4-16
- fork 4-16
- getpgrp 4-16
- getpid 4-16
- getppid 4-16
- gsignal 4-16
- kill 4-16
- lock 4-16
- monitor 4-16
- nap 4-16
- nice 4-16
- pause 4-16
- procl 4-16
- profil 4-16
- ptrace 4-16
- rdchk 4-16
- sbrk 4-16
- setpgrp 4-16
- signal 4-16
- sleep 4-16
- ssignal 4-16
- system 4-16
- times 4-16
- ulimit 4-16
- wait 4-16

Process control functions, described 9-1

Process ID 4-16

- described 9-1

Processes

- calling a system program 9-2
- child 9-7
- communication by pipe 11-1
- described 9-1
- ID 9-1
- multiple copies 9-7
- parent 9-7
- splitting 9-7
- terminating 9-3
- termination status 9-9
- under shell control 9-7
- waiting 9-8

proc.h include file 5-12

procl 4-16

Product names, notational conventions 1-4

profil 4-16

Program

- starting 9-3

Programs, invoking 6-4

Prompts 1-4

ptrace 4-16

putc 4-8

putc 4-8 (*continued*)
 function 6-18
 putchar 4-8
 putchar function 6-9
 writing to the standard output 4-8
 putenv 4-29
 putpwent routine 4-7
 puts 4-8
 function 6-10
 puts function
 standard input and output files 4-8
 putuline routine 4-26
 putw 4-8
 pwd.h include file 5-5

Q

qsort 4-22
 Quotation marks, use of 1-4

R

rand
 described 4-11
 Random access functions
 character pointer 6-34
 described 6-34
 Random-number generation
 drand48 routine 4-18
 erand48 routine 4-18
 jrand48 routine 4-18
 lcong48 routine 4-18
 lrand48 routine 4-18
 mrand48 routine 4-18
 nrand48 routine 4-18
 seed48 routine 4-18
 srand48 routine 4-18
 raw 4-18, 4-20
 raw function 7-40, 7-41
 RAW mode
 direct input 7-9
 restoring editing and signal-generating
 functions 7-42
 terminal configuration 7-13, 7-14, 7-15, 7-26
 rdchk 4-16
 read
 described 4-10
 differences from XENIX version C-18
 end-of-file condition 2-7
 read (*continued*)
 function 6-30
 XENIX version, differences from B-14
 read function
 low-level pipe 4-15
 readdir routine 4-4
 Reading
 characters 7-13, 7-15, 7-24, 7-33
 numbers 7-15, 7-24
 strings 7-14, 7-15, 7-24
 terminal name 7-44
 realloc 4-13
 realloc function 12-4
 Records
 reading from a file 6-16
 writing to a file 6-20
 Redirection symbol (<) 6-3
 Redirection symbol (
) 6-3
 refresh 4-18, 4-20
 refresh function
 clearing characters on the screen 7-18, 7-29,
 7-39
 moving the cursor 7-43
 terminal cursor 7-38
 updating different windows 7-3, 7-34
 updating the terminal screen 7-19
 Refreshing
 screens 7-19
 window, from 7-30
 regcmp 4-29
 regex 4-29
 regexp.h include file 5-5
 reg.h include file 5-12
 relsym86.h include file 5-13
 relsym.h include file 5-13
 resetty 4-18, 4-20
 resetty function 7-43
 Return value on error *See* Errors
 rewind 4-10
 function 6-36
 rewinddir routine 4-4
 Routines
 buffer manipulation 5-5
 category, by 4-1
 data stream
 fdopen 6-2
 fopen 6-2
 freopen 6-2
 database manipulation 4-3
 directory operation 4-4
 group file control 4-6
 I/O
 file operations 4-7

Index

Routines (*continued*)

- math 5-4
 - abs 4-11
 - acos 4-11
 - asin 4-11
 - atan 4-11
 - atan2 4-11
 - bessel 4-11
 - cabs 4-11
 - ceil 4-11
 - cos 4-11
 - cosh 4-11
 - erf 4-11
 - erfc 4-11
 - errno, use of 2-6
 - error handling 2-6, 4-12
 - exp 4-11
 - fabs 4-11
 - floor 4-11
 - fmod 4-11
 - frexp 4-11
 - gamma 4-11
 - hypot 4-11
 - include files 4-12
 - j0,j1,jn,y0,y1,yn 4-11
 - ldexp 4-11
 - log 4-11
 - log10 4-11
 - matherr 4-11
 - modf 4-12
 - pow 4-12
 - rand 4-11
 - sin 4-12
 - sinh 4-12
 - sqrt 4-12
 - srand 4-11
 - tan 4-12
 - tanh 4-12
- message control 4-14
- MS-DOS specific B-2
- password file control 4-6
- random-number generation 4-18
- semaphore control 4-23
- shared memory 4-23
- system accounting control 4-26
- terminal control 4-27

S

- XENIX to MS-DOS cross development C-1
- savetty 4-18, 4-20
- savetty function 7-43

- Saving
 - from a window 7-30
- sbrk 4-13, 4-16
- scanf 4-8
 - argument-type-checking limitations 2-5
 - family 2-8
 - function 6-8
- scanf function
 - reading standard input 4-8
- scanw 4-18, 4-20
- scanw function 7-15
- Screen processing 7-2
- Screen processing functions 4-18
- Screen updating 7-1
- Screen-processing functions *See* Functions, screen processing
- Screen-processing library 7-5
- Screens
 - clearing 7-18, 7-19, 7-28
 - erasing 7-18
 - initializing 7-7
 - memory 7-2
 - overwriting 7-32
 - preparing 7-7
 - refreshing 7-19
- scroll 4-18, 4-20
- scroll flag 7-9, 7-38
- scroll function 7-39
- scrollok 4-18, 4-20
- scrollok function 7-38
- sdenter 4-23
- sdenter function 12-90
- sdfree 4-24
- sdfree function 12-94
- sdget 4-24
- sdget function 12-88
- sdgetv 4-24
- sdgetv function 12-92
- sd.h include file 5-6, 5-13
- sdleave 4-24
- sdleave function 12-91
- sdu.h include file 5-13
- sdwaitv 4-24
- sdwaitv function 12-93
- search.h 4-22
- search.h include file 5-6
- Searching and sorting
 - bsearch 4-22
 - include files 4-22
 - lfind 4-22
 - lsearch 4-22
 - qsort 4-22
- seed48 routine 4-18
- seekdir routine 4-4

- semaphore data structure 12-59
- Semaphore functions, described 12-50
- semaphore identifier 12-59
- semaphore IPC organization 12-59
- semaphore operation permissions codes 12-64
- semaphore set (array) 12-59
- Semaphore-control routines
 - creatsem 4-23
 - nbwaitsem 4-23
 - opensem 4-23
 - resource allocation 4-23
 - semct 4-23
 - semget 4-23
 - semop 4-23
 - sigsem 4-23
 - waitsem 4-23
- Semaphores
 - checking status 12-54
 - creating 12-51
 - described 12-50
 - opening 12-52
 - relinquishing control 12-54
 - requesting control 12-53
- semct 4-23
- semget 4-23
- sem.h include file 5-14
- semop 4-23
- setbuf 4-8
 - function 6-26
- setgid 4-29
- setgrent routine 4-7
- setgrp 4-16
- setjmp 4-29
- setjmp.h include file 5-6
- setpwent routine 4-7
- setterm 4-18, 4-20
- setterm function 7-10, 7-43
- setuid 4-30
- setutent routine 4-26
- setvbuf 4-8
- sgctl 4-3
- sgtty.h include file 5-6
- Shared data
 - attaching segments 12-88
 - creating segments 12-88
 - described 12-86
 - entering segments 12-90
 - freeing segments 12-94
 - leaving segments 12-91
 - version number 12-92
 - waiting for segments 12-93
- Shared memory
 - description 12-86
- shared memory data structure 12-97
- shared memory identifier 12-97
- Shared memory operation permissions codes
 - 12-101
- shared memory segment 12-97
- shared memory, using 12-97
- Shared-memory routines
 - ftok 4-23
 - sdenter 4-23
 - sdfree 4-24
 - sdget 4-24
 - sdgetv 4-24
 - sdleave 4-24
 - sdwaitv 4-24
 - shared segments 4-23
 - shmat 4-24
 - shmctl 4-24
 - shmdt 4-24
 - shmget 4-24
- Shell
 - called as a separate process 9-7
- Shells
 - directing 6-3
- shmat 4-24
- shmctl 4-24
- shmdt 4-24
- shmget 4-24
- shutdn 4-30
- Side effects in macro arguments 2-3
- signal
 - described 4-16
 - differences from XENIX version C-18
 - XENIX version, differences from B-14
- signal.h include file 5-6, 5-14
- sigsem 4-23
- sigsem function 12-54
- sin
 - described 4-12
 - floating-point support 2-8
- sinh
 - described 4-12
 - floating-point support 2-8
- sinh function A-8
- sites.h include file 5-14
- size_t type 3-3
- sleep 4-16
- Small capitals, use of 1-4
- Sorting *See* Searching and sorting
- sprintf 4-8
 - argument-type-checking limitations 2-5
 - function 8-13
- sputl 4-3
- sqrt
 - described 4-12
 - floating-point support 2-8

Index

- sqrt function A-8
- srand
 - described 4-11
- srand48 routine 4-18
- scanf
 - argument-type-checking limitations 2-5
 - function 8-12
- ssignal 4-16
- Stack checking 2-5
- Standard error
 - described 6-5
- Standard error file 4-8, 6-2
- Standard files
 - described 6-5
 - predefined file
 - descriptors 6-28
 - pointers 6-13
 - reading and writing 6-6
 - redirecting 6-6
- Standard input
 - described 6-5
 - reading 6-6
 - characters 6-6
 - formatted input 6-8
 - strings 6-7
- Standard input file
 - filename 6-2
 - pipes 6-3
 - redirecting 4-8, 6-3
 - screen processing 7-2
- Standard I/O
 - file 6-1
 - functions 6-1
- Standard I/O routines 4-8
- Standard I/O routines *See* I/O routines
- Standard output
 - described 6-5
 - writing 6-9
 - characters 6-9
 - formatted output 6-11
 - strings 6-10
- Standard output file
 - filename 6-2
 - pipes 6-3
 - redirecting 4-8, 6-3
 - screen processing 7-2
- Standard screen 7-3
- Standard types
 - clock_t 3-3
 - FILE 3-3
 - jmp_buf 3-3
 - listed 3-2
 - size_t 3-3
 - stat *See* stat type
- Standard types (*continued*)
 - timeb 3-3
 - time_t 3-3
 - tm 3-3
 - utimbuf 3-3
 - standend 4-18, 4-20
 - standend function 7-36
 - stand.h include file 5-6
 - standout 4-18, 4-20
 - standout function 7-35
 - Starting programs 9-3
 - stat
 - described 4-5
 - differences from XENIX version C-18
 - XENIX version, differences from B-15
 - stat type
 - described 3-3
 - stat.h include file 5-14
 - stderr file pointer 6-2
 - stderr, standard error file pointer 6-1, 6-13
 - stdin
 - redirecting 6-3
 - stdin file pointer 6-2
 - stdin, standard input file pointer 6-1, 6-13
 - stdio.h file
 - described 6-1
 - including 6-1
 - stdio.h include file
 - defining routines 5-7
 - file pointer 6-2
 - stdout
 - redirecting 6-3
 - stdout file pointer 6-2
 - stdout, standard output file pointer 6-1, 6-13
 - stdio.h file
 - described 6-1
 - including 6-1
 - stdio.h include file
 - defining routines 5-7
 - file pointer 6-2
 - stdout
 - redirecting 6-3
 - stdout file pointer 6-2
 - stdout, standard output file pointer 6-1, 6-13
 - stdscr pointer 7-3, 7-5
 - stime 4-27
 - store 4-3
 - strcat 4-25
 - function 8-8
 - strchr 4-25
 - strcmp 4-25
 - function 8-8
 - strcpy 4-25
 - strcpy function 8-9
 - strcspn 4-25
 - strdup 4-25
 - Stream I/O
 - clearerr 4-8
 - described 4-8
 - error handling 2-6
 - fclose 4-8
 - fdopen 4-8
 - feof 4-8
 - ferror 4-8

Stream I/O (*continued*)

- fflush 4-8
- fgetc 4-8
- fgets 4-8
- fileno 4-8
- fopen 4-8
- fprintf 4-8
- fputc 4-8
- fputs 4-8
- fread 4-8
- freopen 4-8
- fscanf 4-8
- fwrite 4-8
- getc 4-8
- getchar 4-8
- gets 4-8
- getw 4-8
- printf 4-8
- putc 4-8
- putchar 4-8
- puts 4-8
- putw 4-8
- scanf 4-8
- setbuf 4-8
- setvbuf 4-8
- sprintf 4-8
- sscanf 4-8
- tmpfile 4-8
- ungetc 4-8
- vfprintf 4-8
- vprintf 4-8
- vsprintf 4-8

String functions, described 8-7

String manipulation

- include files 4-24
- strcat 4-25
- strchr 4-25
- strcmp 4-25
- strcpy 4-25
- strcspn 4-25
- strdup 4-25
- strlen 4-25
- strncat 4-25
- strncmp 4-25
- strncpy 4-25
- strpbrk 4-25
- strrchr 4-25
- strspn 4-25
- strtok 4-25

- string.h 4-24

- string.h include file 5-7

String-manipulation routines 5-7

Strings

- adding 7-12

Strings (*continued*)

- comparing 8-8, 8-11
- concatenating 8-8, 8-10
- copying 8-9, 8-11
- length 8-10
- notational conventions 1-4
- printing 7-12
- printing to 8-13
- processing, described 8-1
- reading 7-14, 7-15, 7-24
- reading from a file 6-15
- reading from standard input 6-7
- scanning 8-12
- writing to a file 6-19
- writing to standard output 6-10
- strlen 4-25
 - function 8-10
- strncat 4-25
 - function 8-10
- strncmp 4-25
- strcmp function 8-11
- strcpy 4-25
- strncpy function 8-11
- strpbrk 4-25
- strrchr 4-25
- strspn 4-25
- strtod 4-3
- strtok 4-25
- strtol 4-3
- strtoul 4-3
- Subdirectory conventions 2-7
- subwin 4-18, 4-21
- subwin function 7-2, 7-21
- swab 4-30
- sync 4-30
- Syntax conventions *See* Notational conventions
- sys subdirectory 2-7
- sys_errlist
 - described 3-2
- sys\timeb.h 4-28
- sys\types.h 4-28
- sys\utime.h 4-28
- sysinfo.h include file 5-14
- sys/locking.h file 12-9
- sysmacros.h include file 5-14
- sys_nerr 3-2
- system 4-16
 - differences from XENIX version C-19
 - function 9-2
- System
 - resources 12-1
- system
 - XENIX version, differences from B-15
- System accounting-control routines

Index

System accounting-control routines (*continued*)

- accounting files 4-26
- acct 4-26
- cuserid 4-26
- endutent 4-26
- getlogin 4-26
- getuid 4-26
- getuline 4-26
- getutent 4-26
- putuline 4-26
- setutent 4-26
- ttyslot 4-26
- utmpname 4-26

System programs

- calling as a separate process 9-2

System resource functions, described 12-1

System-wide variables

- in common library C-9

- sysm.h include file 5-15

T

tan

- described 4-12
- floating-point support 2-8

tanh

- described 4-12
- floating-point support 2-8

tanh function A-8

tellmdir routine 4-4

TERM variable 7-8

termcap command 7-36

Terminal

- capability 7-8
- clearing modes 7-41
- control 7-40
- cursor 7-42
- default 7-10
- default modes 7-9
- flags 7-43
- modes 7-40, 7-43
- name 7-44
- type 7-8, 7-43

Terminal-control routines

- tgetent 4-27
- tgetflag 4-27
- tgetnum 4-27
- tgetstr 4-27
- tgoto 4-27
- tputs 4-27

termination status

termination status (*continued*)

- described 9-9

- processes 9-3

terminfo library 7-3

terminfo.h include file 7-4

termio.h include file 5-7

text.h include file 5-15

tgetent routine 4-27

tgetflag routine 4-27

tgetnum routine 4-27

tgetstr routine 4-27

tgoto routine 4-27

time 4-27

Time

routines

- asctime 4-27
- clock 4-27
- ctime 4-27
- ftime 4-27
- gmtime 4-27
- include files 4-27, 4-28
- localtime 4-27
- stime 4-27
- time 4-27
- tzset 4-27

timeb type 3-3

timeb.h include file 5-15

time.h 4-27, 4-28

time.h include file 5-7

times 4-16

times.h include file 5-15

time_t type 3-3

timezone variable 3-1

tm type 3-3

tmpfile 4-8, 4-30

tmpnam 4-30

toascii 4-2

- function 8-2

tolower 4-2

- function 8-7

touchwin 4-18, 4-21

touchwin function 7-34

toupper 4-2

- function 8-7

tputs routine 4-27

ttold.h include file 5-15

tty.h include file 5-15

ttyname 4-30

ttyslot routine 4-26

ttytype variable 7-5, 7-9, 7-44

Type

- char 7-35

- FILE 6-2

- terminal 7-43

Type (*continued*)

WINDOW(** 7-20
 types.h include file 5-16
 TZ environment variable
 described 3-1
 tzname variable 3-1
 tzset 4-27

U

ulimit 4-16
 ulimit.h include file 5-16
 ultoa 4-3
 umask
 described 4-5
 differences from XENIX version C-20
 XENIX version, differences from B-16
 umount 4-5
 uname 4-30
 Unbuffered I/O
 creating 6-26
 described 6-24
 low-level functions 6-28
 ungetc 4-8
 function 6-26
 UNIX
 compatibility with 1-1
 unlink
 described 4-5
 differences from XENIX version C-20
 XENIX version, differences from B-16
 unlst.h include file 5-8
 Uppercase letters, use of 1-4
 user.h include file 5-16
 ustat 4-5
 ustat.h include file 5-8
 utimbuf type 3-3
 utime 4-5
 described 4-27
 differences from XENIX version C-20
 XENIX version, differences from B-16
 utmp.h include file 5-8
 utmpname routine 4-26
 utsname.h include file 5-16

V

values.h include file 5-8
 varargs.h include file 5-8

var.h include file 5-16
 Variable, global *See* Global variables
 Variables
 allocating for arrays 12-3
 COLS 7-10
 Def_term 7-8
 errno A-1
 LINES 7-10
 memory allocation 12-2
 My_term 7-8
 TERM 7-8
 ttytype 7-8, 7-44
 Vertical bar (|) 1-4
 vfprintf 4-8
 vprintf 4-8
 vsprintf 4-8

W

waddch 4-18, 4-21
 waddch function 7-3, 7-23
 waddstr 4-18, 4-21
 waddstr function 7-23
 wait 4-16
 function 9-8
 waitsem 4-23
 waitsem function 12-53
 wclear 4-18, 4-21
 wclear function 7-29
 wclrtoebot 4-18, 4-21
 wclrtoebot function 7-29
 wclrtoeol 4-18, 4-21
 wclrtoeol function 7-29
 wdelch 4-18, 4-21
 wdelch function 7-28
 wdeleteln 4-18, 4-21
 wdeleteln function 7-28
 werase 4-18, 4-21
 werase function 7-29
 wgetch 4-18, 4-21
 wgetch function 7-24
 wgetstr 4-18, 4-21
 wgetstr function 7-25
 winch 4-18, 4-21
 winch function 7-33
 Window functions *See* Functions, screen
 processing
 WINDOW* pointer 7-5
 WINDOW(** type 7-20, 7-21
 Windows
 adding 7-22
 creating 7-20

Index

Windows (*continued*)

- current position 7-37
- default flags 7-9
- deleting 7-34
- dimension 7-22
- functions 7-2
- moving 7-26, 7-32, 7-40
- origins 7-22
- overlying 7-31
- printing 7-22
- reading characters 7-33
- refreshing from 7-30
- save to a file 7-30
- scrolling 7-39
- setting flags 7-38
- structure 7-22
- subwindow 7-21
- touching 7-34
- using 7-20

winsch 4-18, 4-21

winsch function 7-27

winsertln 4-18, 4-21

winsertln function 7-27

wmove 4-18, 4-21

wmove function 7-3, 7-26, 7-43

wprintw 4-18, 4-21

wprintw function 7-23

wrefresh 4-18, 4-21

wrefresh function 7-3, 7-30, 7-43

write

- described 4-10
- differences from XENIX version C-20
- function 6-30
- XENIX version, differences from B-16

write function

- writing to pipes 4-15

wscanw 4-18, 4-21

wscanw function 7-25

wstandend 4-18, 4-22

wstandend function 7-36

wstandout 4-18, 4-22

wstandout function 7-36

X

xlist 4-30

Y

y0 *See* Bessel functions

y1 *See* Bessel functions

yn *See* Bessel functions "

Replace this Page
with Tab Marked:

C LANGUAGE REFERENCE

XENIX[®] System V

Development System

C Language Reference

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.

Contents

1 Introduction

- 1.1 Overview of the C Language 1-1
- 1.2 About This Manual 1-2
- 1.3 Notational Conventions 1-4

2 Elements of C

- 2.1 Introduction 2-1
- 2.2 Character Sets 2-1
- 2.3 Constants 2-8
- 2.4 Identifiers 2-14
- 2.5 Keywords 2-15
- 2.6 Comments 2-16
- 2.7 Tokens 2-17

3 Program Structure

- 3.1 Introduction 3-1
- 3.2 Source Program 3-1
- 3.3 Source Files 3-3
- 3.4 Functions and Program Execution 3-5
- 3.5 Lifetime and Visibility 3-6
- 3.6 Naming Classes 3-11

4 Declarations

- 4.1 Introduction 4-1
- 4.2 Type Specifiers 4-2
- 4.3 Declarators 4-8
- 4.4 Variable Declarations 4-16
- 4.5 Function Declarations (Prototypes) 4-31
- 4.6 Storage Classes 4-37
- 4.7 Initialization 4-44
- 4.8 Type Declarations 4-51
- 4.9 Type Names 4-53

5 Expressions and Assignments

- 5.1 Introduction 5-1
- 5.2 C Operands 5-1
- 5.3 C Operators 5-13
- 5.4 Assignment Operators 5-30

- 5.5 Precedence and Order of Evaluation 5-35
- 5.6 Type Conversions 5-38

6 Statements

- 6.1 Introduction 6-1
- 6.2 The break Statement 6-2
- 6.3 The Compound Statement 6-3
- 6.4 The continue Statement 6-4
- 6.5 The do Statement 6-4
- 6.6 The Expression Statement 6-5
- 6.7 The for Statement 6-6
- 6.8 The goto and Labeled Statements 6-8
- 6.9 The if Statement 6-9
- 6.10 The Null Statement 6-10
- 6.11 The return Statement 6-11
- 6.12 The switch Statement 6-13
- 6.13 The while Statement 6-15

7 Functions

- 7.1 Introduction 7-1
- 7.2 Function Definitions 7-3
- 7.3 Function Prototypes (Declarations) 7-12
- 7.4 Function Calls 7-14

8 Preprocessor Directives and Pragmas

- 8.1 Introduction 8-1
- 8.2 Manifest Constants and Macros 8-2
- 8.3 Include Files 8-10
- 8.4 Conditional Compilation 8-12
- 8.5 Line Control 8-17
- 8.6 Pragmas 8-18

A Differences Between K&R C and Microsoft C

- A.1 Introduction A-1

B Syntax Summary

- B.1 Tokens B-1
- B.2 Expressions B-7
- B.3 Declarations B-9
- B.4 Statements B-13
- B.5 Definitions B-14
- B.6 Preprocessor Directives B-15
- B.7 Pragmas B-15

Chapter 1

Introduction

- 1.1 Overview of the C Language 1-1
- 1.2 About This Manual 1-2
- 1.3 Notational Conventions 1-4

1.1 Overview of the C Language

The C language is a general-purpose programming language known for its efficiency, economy, and portability. While these characteristics make it a good choice for almost any kind of programming, C has proven especially useful in systems programming because it facilitates writing fast, compact programs that are readily adaptable to other systems. Well-written C programs are often as fast as assembly-language programs, and they are typically easier for programmers to read and maintain.

C was designed to combine efficiency and power in a relatively small language. C does not include built-in functions to perform tasks such as input and output, storage allocation, screen manipulation, and process control. To perform such tasks, C programmers rely on “run-time libraries,” a set of predefined functions and macros. The run-time library functions available for use in Microsoft® C programs are discussed in a separate manual, the *XENIX® C Library Guide*.

C’s design makes it both flexible and compact. Because the language is relatively sparse, it neither assumes nor imposes a particular programming model. You can use the run-time routines supplied, or tailor your own variations for special purposes. The design also helps to isolate language features from processor-specific features in a particular C implementation, which makes it easier to write portable code. While the strict definition of the language makes it independent of any particular operating system or machine, you can easily add system-specific routines to take advantage of the most efficient features of a particular machine.

Note

Microsoft is committed to conformity with the developing standard for the C language as set forth in the Draft Proposed American National Standard — Programming Language C (hereinafter referred to as the ANSI C standard. Microsoft extensions to the ANSI C standard are noted in the text. Because the extensions are not a part of the ANSI C standard, their use may restrict portability of programs between systems. See your compiler guide for information on enabling and disabling Microsoft extensions.

C Language Reference

The C language includes the following significant features:

- A full set of loop, conditional, and transfer statements to control program flow logically and efficiently and to encourage structured programming.
- A large set of operators. Many of these operators correspond to common machine instructions, allowing a direct translation into machine code. The variety of operators allows you to specify different kinds of operations clearly and with a minimum of code.
- Several sizes of integers, as well as single- and double-precision floating-point types. You can also design more complex data types, such as arrays and data structures, to suit specific program needs.
- Declarations of “pointers” to variables and functions. A pointer to an item corresponds to the item’s machine address. Pointers can make programs more efficient, since they let you refer to items in the same way the machine does. C also supports pointer arithmetic, which lets you access and manipulate memory addresses directly.
- A C preprocessor that acts on the text of files before they are compiled. You can use the C preprocessor to define program constants, substitute fast macro definitions for function calls, and compile parts of programs based on specified conditions.

C is a flexible language that leaves many programming decisions up to you. In keeping with this philosophy, C imposes few restrictions in matters such as type conversion. Although this characteristic of the language can make your programming job easier, you must know the language well to understand how programs will behave.

1.2 About This Manual

The *C Language Reference* defines the C language as implemented by Microsoft Corporation. It is intended as a reference for programmers experienced in C or other programming languages. Thorough knowledge of programming fundamentals is assumed.

Consult your compiler guide for an explanation of how to compile and link C programs on your system; this manual also contains information specific to the implementation of C on your system.

This manual is organized as follows:

Chapter 1, “Introduction,” introduces this guide and outlines the notational conventions used in this manual.

Chapter 2, “Elements of C,” describes the letters, numbers, and symbols that can be used in C programs and the combinations of characters that have special meanings to the C compiler.

Chapter 3, “Program Structure,” discusses the components and structure of C programs and explains how C source files are organized.

Chapter 4, “Declarations,” describes how to specify the attributes of C variables, functions, and user-defined types. C provides a number of predefined data types and lets the programmer declare “aggregate” types and pointers. Function prototypes, a relatively new feature of C, are discussed in this chapter, as well as in Chapter 7, “Functions.”

Chapter 5, “Expressions and Assignments,” describes the operands and operators that form C expressions and assignments. The chapter also discusses the type conversions and side effects that may occur when expressions are evaluated.

Chapter 6, “Statements,” describes C statements, which control the flow of program execution.

Chapter 7, “Functions,” discusses C functions. In particular, this chapter explains function prototypes, formal parameters, and return values. It also describes how to define, declare, and call functions.

Chapter 8, “Preprocessor Directives and Pragmas,” describes the instructions recognized by the C preprocessor, a text processor that is automatically invoked before compilation. This chapter also introduces “pragmas,” special instructions to the compiler that you can place in source files.

Appendix A, “Differences,” lists the differences between Microsoft C and the description of the C language found in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

Appendix B, “Syntax Summary,” summarizes the syntax of the C language as implemented by Microsoft.



C Language Reference

1.3 Notational Conventions

This manual uses the following notational conventions:

keywords Bold type indicates text that must be typed exactly as shown. Text that is shown in bold type includes C keywords, such as **goto** and **char**, and operators, such as the addition operator (+) and the multiplication operator (*).

placeholders Terms in italics may appear in syntax descriptions or in the text. In these instances, the terms are being used as placeholders that you would replace with specific terms or values in an actual C program. For example, in

goto name;

name appears in italics to show that this is a general form for the **goto** statement. In an actual program statement, you must supply a particular identifier for the placeholder *name*.

Occasionally, italics are used to emphasize particular words in the text.

Examples Examples of C programs and program elements appear in a special typeface to look similar to listings on the screen or the output of commonly used computer printers:

```
int x, y;  
.  
.  
.  
swap (&x, &y);
```

Input: *output* Some examples show both program output and user input; in these cases, input is shown in a darker font.

Repeating . . . elements

Vertical ellipsis dots are used in program examples or syntax to indicate that a portion of the program is omitted.

In the following example, the vertical ellipsis dots indicate that zero or more declarations, followed by one or more statements, may appear between the braces:

```
{
  [declaration]
  .
  .
  .
  statement
  [statement]
  .
  .
  .
}
```

In the following excerpt, two program lines are shown. The ellipsis dots between the lines indicate that additional program lines appear between these two lines but are not shown:

```
int x, y;
.
.
.
swap (&x, &y);
```

Horizontal ellipsis dots following an item indicate that more items of the same form may appear. For instance,

= {*expression* [, *expression*]...}

indicates that one or more expressions separated by commas may appear between the braces ({}).

C Language Reference

[*optional items*]

Brackets enclose optional items in syntax descriptions. For example,

return [*expression*];

is a syntax description showing that *expression* is an optional item in the **return** statement.

Single brackets are used to indicate brackets used by C-language array declarations and subscript expressions. For instance, *a[10]* is an example of brackets in a C subscript expression.

“Defined terms”

Quotation marks set off terms defined in the text. For example, the term “token” appears in quotation marks when it is defined.

Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form " " rather than “ ”. The following example shows a C string:

"abc"

Quotation marks also occasionally indicate a term that is being used in a colloquial sense.

KEY+NAMES

Names of special key combinations, such as CTRL+Z, appear in small capital letters.

Chapter 2

Elements of C

- 2.1 Introduction 2-1
- 2.2 Character Sets 2-1
 - 2.2.1 Letters, Digits, and Underscore 2-2
 - 2.2.2 White-Space Characters 2-2
 - 2.2.3 Punctuation and Special Characters 2-3
 - 2.2.4 Escape Sequences 2-4
 - 2.2.5 Operators 2-6
- 2.3 Constants 2-8
 - 2.3.1 Integer Constants 2-8
 - 2.3.2 Floating-Point Constants 2-10
 - 2.3.3 Character Constants 2-11
 - 2.3.4 String Literals 2-12
- 2.4 Identifiers 2-14
- 2.5 Keywords 2-15
- 2.6 Comments 2-16
- 2.7 Tokens 2-17

2.1 Introduction

This chapter describes the elements of the C programming language, including the names, numbers, and characters used to construct a C program. The following topics are discussed in this chapter:

- Character sets
- Constants
- Identifiers
- Keywords
- Comments
- Tokens



2.2 Character Sets

Two character sets are defined for use in C programs: the “C character set” and the “representable character set.”

The C character set consists of the letters, digits, and punctuation marks having specific meanings in the C language. You construct a C program by combining the characters of the C character set into meaningful statements.

The C character set is a subset of the representable character set. The representable character set includes each letter, digit, and symbol that can be represented graphically with a single character. The extent of the representable character set depends on the type of terminal, console, or character device being used.

All characters in a C program must be part of the C character set. However, string literals, character constants, comments, and file names in **#include** directives can include any character from the representable character set.

Since each character in the C character set has an explicit meaning in the language, the compiler generates error messages when it finds inappropriate or inappropriately used characters in a program.

The sections that follow describe the characters and symbols of the C character set and explain how and when to use them.

C Language Reference

2.2.1 Letters, Digits, and Underscore

The C character set includes the uppercase and lowercase letters of the English alphabet, the 10 decimal digits of the Arabic number system, and the underscore (`_`) character.

- Uppercase English letters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Lowercase English letters

a b c d e f g h i j k l m n o p q r s t u v w x y z

- Decimal digits

0 1 2 3 4 5 6 7 8 9

- Underscore character (`_`)

These characters are used to form the constants, identifiers, and keywords described later in this chapter.

The C compiler treats uppercase and lowercase letters as distinct characters. For example, if a lowercase *a* is specified in an identifier, you cannot substitute an uppercase *A*; you must use the lowercase letter.

2.2.2 White-Space Characters

The space, tab, line-feed, carriage-return, form-feed, vertical-tab, and new-line characters are called “white-space characters” because they serve the same purpose as the spaces between words and lines on a printed page. These characters separate the items you define, such as constants and identifiers, from other items in a program.

The C compiler ignores white-space characters unless you use them as separators or as components of character constants or string literals. Therefore, you can use extra white-space characters to make a program more readable. The compiler also treats comments as white space. (Comments are described in “Comments.”)

2.2.3 Punctuation and Special Characters

The punctuation and special characters in the C character set have various uses, from organizing program text to defining the tasks that the compiler or compiled program will carry out. Table 2.1 lists the punctuation and special characters in the C character set.

Table 2.1
Punctuation and Special Characters

Character	Name	Character	Name
,	Comma	!	Exclamation mark
.	Period		Vertical bar
;	Semicolon	/	Forward slash
:	Colon	\	Backslash
?	Question mark	~	Tilde
'	Single quotation mark	+	Plus sign
"	Double quotation mark	#	Number sign
(Left parenthesis	%	Percent sign
)	Right parenthesis	&	Ampersand
[Left bracket	^	Caret
]	Right bracket	*	Asterisk
{	Left brace	-	Minus sign
}	Right brace	=	Equal sign
<	Left angle bracket	>	Right angle bracket

These characters have special meanings in C. Their uses are described throughout this manual. Any punctuation character from the representable character set that does not appear in Table 2.1 can be used only in string literals, character constants, comments, and file names in **#include** directives.

2.2.4 Escape Sequences

Strings and character constants can contain “escape sequences.” Escape sequences are character combinations representing white-space and non-graphic characters. An escape sequence consists of a backslash (\) followed by a letter or by a combination of digits.

Escape sequences are typically used to specify actions such as carriage returns and tab movements on terminals and printers and to provide literal representations of nonprinting characters and characters that normally have special meanings, such as the double-quotation-mark character ("). Table 2.2 lists the C escape sequences.

Table 2.2
Escape Sequences

Escape Sequence	Name
<code>\n</code>	New line
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\b</code>	Backspace
<code>\r</code>	Carriage return
<code>\f</code>	Form feed
<code>\a</code>	Bell (alert)
<code>\'</code>	Single quotation mark
<code>\"</code>	Double quotation mark
<code>\\</code>	Backslash
<code>\ddd</code>	ASCII character in octal notation
<code>\xdd</code>	ASCII character in hexadecimal notation

If a backslash precedes a character that does not appear in Table 2.2, the backslash is ignored and the character is represented literally. For example, the pattern `\c` represents the character `c` in a string literal or character constant. However, the use of lowercase letters in escape sequences is reserved by ANSI for future standardization. Therefore, occurrences of undefined escape sequences, though currently innocuous, could pose future portability problems.

The sequence `\ddd` lets you specify any character in the ASCII (American Standard Code for Information Interchange) character set as a three-digit octal character code. Similarly, the sequence `\xdd` lets you specify any ASCII character as a three-digit hexadecimal character code. For example, you can give the ASCII backspace character as the normal C escape sequence `(\b)`, or you can code it as `\010` (octal) or `\x008` (hexadecimal).

You can use only the digits 0 through 7 in an octal escape sequence. Though you do not need to use all three digits (as in the form shown in the previous paragraph), you must use at least one. For example, you can specify the ASCII backspace character in octal notation as `\10`. Similarly, you must use at least one digit for a hexadecimal escape sequence, but you can omit the second and third digits. Therefore you could specify the hexadecimal escape sequence for the backspace character either as `\x08` or as `\x8`.



Note

When you use octal and hexadecimal escape sequences in strings, it is safest to give all three digits of the escape sequence. If you don't specify all digits of the escape sequence, and the character immediately following the escape sequence happens to be an octal or hexadecimal digit, the compiler interprets that character as part of the sequence. For example, if you printed the string `"\x07Bell"`, the result would be `{ell` because `\x07B` is interpreted as the ASCII left-brace character (`{`). The string `\x007Bell` (note the two leading zeros) is the correct way to represent the bell character followed by the word *Bell*. The string `\x7Bell` would generate a compiler diagnostic message because `7BE` hexadecimal is too big a number to fit in one byte.

Escape sequences let you send nongraphic control characters to a display device. For example, the escape character `\033` is often used as the first character of a control command for a terminal or printer. Some escape sequences are device specific. For instance, the vertical tab and form feed (`\v` and `\f`) do not affect screen output, but they do perform appropriate operations for a printer.

You should always represent nongraphic characters by escape sequences in C programs, since using the characters directly may generate compiler diagnostic messages.

You can also use the backslash character (`\`) as a continuation character. When a new-line character immediately follows the backslash, the

C Language Reference

compiler ignores the backslash and the new line and treats the next line as part of the previous line. This is useful primarily for preprocessor definitions longer than a single line. In the past this feature was also used to create strings longer than one line. However, the string concatenation feature (see “String Literals”) is now preferred for creating long string literals.

2.2.5 Operators

“Operators” are symbols (both single characters and character combinations) that specify how values are to be manipulated. Each symbol is interpreted as a single unit, called a “token.” (Tokens are defined in “Tokens.”)

Table 2.3 lists the symbols that make up the C unary operators and names each operator. Table 2.4 lists the C binary and ternary operators and names them. You must specify operators exactly as they appear in the tables, with no white space between the characters of multicharacter operators. Note that three operator symbols (asterisk, minus sign, and ampersand) appear in both tables. Their interpretation as unary or binary depends on the context in which they appear. The **sizeof** operator is not included in these tables. It consists of a keyword (**sizeof**) rather than a symbol, and is listed in “Keywords.”

Table 2.3
Unary Operators

Operator	Name
!	Logical NOT
~	Bitwise complement
-	Arithmetic negation
*	Indirection
&	Address of
+	Unary plus ^a

^a The unary plus operator is implemented syntactically, but not semantically.

Table 2.4
Binary and Ternary Operators

Operator	Name	Operator	Name
+	Addition	&&	Logical AND
-	Subtraction		
*	Multiplication		Logical OR
/	Division	,	Sequential evaluation
%	Remainder	?:	Conditional ^a
<<	Left shift	++	Increment
>>	Right shift	--	Decrement
<	Less than	=	Simple assignment
<=	Less than or equal to	+=	Addition assignment
>	Greater than	-=	Subtraction assignment
>=	Greater than or equal to	*=	Multiplication assignment
==	Equality	/=	Division assignment
!=	Inequality	%=	Remainder assignment
&	Bitwise AND	>>=	Right-shift assignment
	Bitwise inclusive OR	<<=	Left-shift assignment
^	Bitwise exclusive OR	&=	Bitwise-AND-assignment
=	Bitwise inclusive-OR assignment	^=	Bitwise exclusive-OR assignment

C Language Reference

- ^a The conditional operator is a ternary operator, not a multicharacter operator. A conditional expression has the following form: *expression ? expression : expression*.

For a complete description of each operator, see the “Expressions and Assignments” chapter.

2.3 Constants

A *constant* is a number, character, or character string that can be used as a value in a program. A constant’s value cannot be modified.

The C language has four kinds of constants: integer constants, floating-point constants, character constants, and string literals.

2.3.1 Integer Constants

Syntax

digits

0*odigits*

0x*hdigits*

0X*hdigits*

An “integer constant” is a decimal, octal, or hexadecimal number that represents an integral value in one of the following forms:

- A “decimal constant” has the form *digits*, where *digits* represents one or more decimal digits (0 through 9), the first of which is not a zero.
- An “octal constant” has the form **0***odigits*, where *odigits* represents one or more octal digits (0 through 7). The leading zero is required.
- A “hexadecimal constant” has the form **0x***hdigits* or **0X***hdigits*, where *hdigits* represents one or more hexadecimal digits (0 through 9 and either uppercase or lowercase *a* through *f*). The leading **0x** or **0X** is required.

No white-space characters can separate the digits of an integer constant.

Table 2.5 gives examples of the three forms of integer constants.

Table 2.5
Examples of Integer Constants

<u>Decimal Constants</u>	<u>Octal Constants</u>	<u>Hexadecimal Constants</u>
<i>10</i>	<i>012</i>	<i>0xa</i> or <i>0xA</i>
<i>132</i>	<i>0204</i>	<i>0x84</i>
<i>32179</i>	<i>076663</i>	<i>0x7db3</i> or <i>0x7DB3</i>

Integer constants always specify positive values. If you need to use a negative value, place a minus sign (-) in front of a constant to form a constant expression with a negative value. (In this case, the minus sign is interpreted as the unary arithmetic negation operator.)

Every integer constant is given a type based on its value. A constant's type determines which conversions must be performed when the constant is used in an expression or when the minus sign (-) is applied, as summarized in the following rules:

- Decimal constants are considered signed quantities and are given **int** type, or **long** type if the size of the value requires it.
- Octal and hexadecimal constants are given **int**, **unsigned int**, **long**, or **unsigned long** type, depending on the size of the constant. If the constant can be represented as an **int**, it is given **int** type. If it is larger than the maximum positive value that can be represented by an **int**, but small enough to be represented in the same number of bits as an **int**, it is given **unsigned int** type. Similarly, a constant that is too large to be represented as an **unsigned int** is given **long** or **unsigned long** type, if necessary.

Table 2.6 shows the ranges of values and the corresponding types for octal and hexadecimal constants on a machine whose **int** type is 16 bits long.

Table 2.6
Types Assigned to Octal and Hexadecimal Constants

Hexadecimal Range	Octal Range	Type
0x0 - 0x7FFF	0 - 077777	int
0x8000 - 0xFFFF	0100000 - 0177777	unsigned int
0x10000 - 0x7FFFFFFF	0200000 - 0177777777	long
0x80000000 - 0xFFFFFFFF	020000000000 - 037777777777	unsigned long

The consequence of the typing rules shown in Table 2.6 is that hexadecimal and octal constants are always zero extended when converted to longer types. (For more information on type conversions, see the “Expressions and Assignments.” chapter.)

You can force any integer constant to be given **long** type by appending the letter *l* or *L* to the end of the constant. Table 2.7 illustrates some forms of **long** integer constants.

Table 2.7
Examples of Long Integer Constants

Decimal Constants	Octal Constants	Hexadecimal Constants
<i>10L</i>	<i>012L</i>	<i>0xaL</i> or <i>0xAL</i>
<i>79l</i>	<i>0115l</i>	<i>0x4fl</i> or <i>0x4Fl</i>

Types are described in the “Declarations” chapter and conversions are described in the “Expressions and Assignments” chapter.

2.3.2 Floating-Point Constants

Syntax

[digits][.digits][E|e[-|+]digits]

A “floating-point constant” is a decimal number that represents a signed real number. The value of a signed real number includes an integer portion, a fractional portion, and an exponent. The *digits* are zero or more decimal digits (0 through 9), and **E** (or **e**) is the exponent symbol. You can omit either the digits before the decimal point (the integer portion of the value) or the digits after the decimal point (the fractional portion), but not both. You can leave out the decimal point only if you include an exponent.

The exponent consists of the exponent symbol (**E** or **e**) followed by a constant integer value. The integer value may be negative. No white-space characters can separate the digits or characters of the constant.

Floating-point constants always specify positive values. However, you can place a minus sign (-) in front of the constant to form a constant floating-point expression with a negative value. In this case, the minus sign is treated as an arithmetic operator.

2

All floating-point constants have type **double**.

Examples

The following examples illustrate some forms of floating-point constants and expressions:

```
15.75
1.575E1
1575e-2
-0.0025
-2.5e-3
25E-4
```

You can omit the integer portion of the floating-point constant, as shown in the following examples:

```
-.125
-.175E-2
```

2.3.3 Character Constants

Syntax

```
'char'
```

A “character constant” is formed by enclosing a single character from the representable character set within single quotation marks (‘ ’). An escape sequence is regarded as a single character and is therefore valid in a character constant. Note that escape characters must be represented by escape sequences or diagnostic messages will be generated. The value of a character constant is the numerical value of the character.

In the syntax above, *char* can be any character from the representable character set (including any escape sequence) except a single quotation mark (’), backslash (\), or new-line character (**\n**). To use a single quotation mark or backslash character as a character constant, precede it with a backslash, as shown in Table 2.8.

Table 2.8
Examples of Character Constants

Constant	Value
' '	Single blank space
'a'	Lowercase a
'?'	Question mark
'\b'	Backspace
'\x1B'	ASCII escape character
'\''	Single quotation mark
'\\'	Backslash

Character constants have type **int**, and are therefore sign extended in type conversions. (See “Type Conversions,” for more information.)

2.3.4 String Literals

Syntax

```
"characters" ["characters"]...
```

A “string literal” is a sequence of characters from the representable character set enclosed in double quotation marks (" "). This example demonstrates a simple string literal:

```
"This is a string literal."
```

In a string literal, *characters* is a placeholder for zero or more characters from the representable character set, including any escape sequence. The double quotation mark ("), backslash (\), or new line must be represented by their escape sequences (\", \\, and \n). Non-printing characters should always be represented by a corresponding escape sequence. Each escape sequence is considered a single character.

To force a new line within a string literal, enter the new-line (\n) escape sequence at the point in the string where you want the line broken, as follows:

```
"Enter a number between 1 and 100\nOr press Return"
```

The traditional way to form string literals that take up more than one line is to type a backslash, then press RETURN. The backslash causes the compiler to ignore the following new-line character. For example, the string literal

```
"Long strings can be bro\
ken into two or more pieces."
```

is identical to the string

```
"Long strings can be broken into two or more pieces."
```

Two or more string literals separated only by white space will be concatenated into a single string. For example, long strings passed as literals to the `printf` function can now be continued in any column of a succeeding line without affecting their appearance when output, if entered as follows:

```
printf ("This is the first half of the string,"
        " this is the second half") ;
```

As long as each part of the string is enclosed in double quotation marks, the parts will be concatenated and output as a single string:

```
This is the first half of the string, this is the second half
```

You can use string concatenation anywhere you might previously have used a backslash followed by a new-line character to enter strings longer than one line. Because ensuing strings can start in any column of the source code without affecting their on-screen representation, strings can be positioned to enhance source-code readability. For example, the following pointer, initialized as two distinct string literals separated only by white space, is stored as a single string. When properly referenced, as in the following example, it produces a result identical to the previous example:

```
char *string = "This is the first half of the string,"
               " this is the second half" ;

printf("%s" , string) ;
```

To use a double quotation mark or backslash within a string literal, precede it with a backslash as shown in the following examples:

```
"First\\Second"

"\\"Yes, I do,\" she said."
```


C Language Reference

Note that an escape sequence (such as `\\` or `\"`) within a string literal counts as a single character.

The characters of a string are stored in order at contiguous memory locations. A null character (represented by the `\0` escape sequence) is automatically appended to, and marks the end of, each string literal. Each string in a program is generally considered to be distinct; however, two identical strings are not guaranteed to receive separate storage. Therefore, programs should not be designed to allow modification of string literals during execution.

String literals have type array of **char** (**char** []). This means that a string is an array with elements of type **char**. The number of elements in the array is equal to the number of characters in the string, plus one for the terminating null character.

2.4 Identifiers

Syntax

letter | *_[letter|digit|_]*...

“Identifiers” are the names you supply for variables, types, functions, and labels in your program. An identifier is a sequence of one or more letters, digits, or underscores (`_`) that begins with a letter or underscore. Identifiers can contain any number of characters, but only the first 31 characters are significant to the compiler. (Other programs that read the compiler output, such as the linker, may recognize even fewer characters.)

You create an identifier by specifying it in the declaration of a variable, type, or function. You can then use the identifier in later program statements to refer to the associated item. Although statement labels are a special kind of identifier and have their own naming class, their creation is similar to that of variables and functions. (Declarations are described in the “Declarations” chapter. Statement labels are described in the “Statements” chapter.)

Because the C compiler considers uppercase and lowercase letters distinct characters, you can create distinct identifiers that have the same spelling but different cases for one or more of the letters.

An identifier cannot have the same spelling and case as a keyword of the language. Keywords are described in “Keywords.”

You should not use leading underscores in identifiers you create: identifiers beginning with an underscore can cause conflicts with the names of system routines or variables, and produce errors. Programs containing names beginning with leading underscores are not guaranteed to be portable.

Note

Some linkers may further restrict the number and type of characters for globally visible symbols. (Visibility is defined in “Lifetime and Visibility.”) Also the linker, unlike the compiler, may not distinguish between uppercase and lowercase letters. Consult your linker documentation for information about naming restrictions imposed by the linker.

Examples

The following are examples of identifiers:

```
j
cnt
temp1
top_of_page
skip12
```

Since uppercase and lowercase letters are considered distinct characters, each of the following identifiers is unique:

```
add
ADD
Add
aDD
```

2.5 Keywords

“Keywords” are predefined identifiers that have special meanings to the C compiler. They can be used only as defined. The name of a program item cannot have the same spelling and case as a C keyword.

C Language Reference

The C language has the following keywords:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

You cannot redefine keywords. However, you can specify text to be substituted for keywords before compilation by using C preprocessor directives (see the “Functions” chapter).

The **volatile** keyword is implemented syntactically, but currently has no semantics associated with it. You cannot use **volatile** as a variable name in your programs.

The following identifiers may be keywords in some implementations. See your compiler guide for more information.

cdecl
far
fortran
huge
near
pascal

2.6 Comments

Syntax

```
/* characters */
```

A “comment” is a sequence of characters that is treated as a single white-space character by the compiler, but is otherwise ignored. In a comment, *characters* can include any combination of characters from the representable character set, including new-line characters, but excluding the “end comment” delimiter (**/*). Comments can occupy more than one line, but they cannot be nested.

Comments can appear anywhere a white-space character is allowed. Since the compiler treats a comment as a single white-space character, you cannot include comments within tokens. However, since the compiler ignores the characters of the comment, you can include keywords in comments without producing errors.

To suppress compilation of a large portion of a program or a program segment that contains comments, bracket the desired portion of code with the **#if** and **#endif** preprocessor directives, rather than “commenting out” the code (see “Conditional Compilation”).

Examples

The following examples illustrate some comments:

```
/* Comments can separate and document
   lines of a program. */

/* Comments can contain keywords such as for
   and while. */

/*****
   Comments can occupy several lines.
   *****/
```

Since comments cannot contain nested comments, the following example causes an error:

```
/* You cannot /* nest */ comments */
```

The error occurs because the compiler recognizes the first `*/`, after the word *nest*, as the end of the comment. It tries to process the remaining text and produces an error when it cannot do so.

2.7 Tokens

In a C source program, the basic element recognized by the compiler is the character group known as a “token.” A token is source-program text the compiler will not attempt to further analyze into component elements.

C Language Reference

For example, the following program fragment uses the word *elsewhere* as the name of a function. Although **else** is a keyword in C, there is no confusion between the function name token and the C keyword token it contains.

```
main()
{
    int i = 0;
    if (i)
        elsewhere() ;
}
```

However, if you were to type *elsewhere* as *else where* with a space between *else* and *where*, the preceding example would elicit a compiler diagnostic message noting the lack of a semicolon before the **else** keyword.

The operators, constants, identifiers, and keywords described in this chapter are examples of tokens. Punctuation characters such as brackets ([]), braces { }, angle brackets (< >), parentheses, and commas are also tokens.

Tokens are delimited by white-space characters and by other tokens, such as operators and punctuation characters. To prevent the compiler from breaking an item down into two or more tokens, white-space characters are not permitted within an identifier, multicharacter operator, or keyword.

When the compiler interprets tokens, it includes as many characters as possible in a single token before moving on to the next token. Because of this behavior, the compiler may not interpret tokens as you intended if they are not properly separated by white space.

Example

Consider the following expression:

```
i+++j
```

In this example, the compiler first makes the longest possible operator (++) from the three plus signs, then processes the remaining plus sign as an addition operator (+). Thus, the expression is interpreted as $(i++) + (j)$, not $(i) + (++j)$. In this and similar cases, use white space and parentheses to avoid ambiguity and ensure proper expression evaluation.

Chapter 3

Program Structure

- 3.1 Introduction 3-1
- 3.2 Source Program 3-1
- 3.3 Source Files 3-3
- 3.4 Functions and Program Execution 3-5
- 3.5 Lifetime and Visibility 3-6
 - 3.5.1 Blocks 3-6
 - 3.5.2 Lifetime 3-7
 - 3.5.3 Visibility 3-7
 - 3.5.4 Summary 3-9
- 3.6 Naming Classes 3-11

3.1 Introduction

This chapter defines terms used later in this manual to describe the C language, and discusses the structure of C source programs. It gives an overview of features of C that are described in detail in other chapters. The syntax and meaning of declarations and definitions are discussed in the “Declarations” chapter and the chapter on “Functions.” The C preprocessor and pragmas are described in “Preprocessor Directives and Pragmas.”

3.2 Source Program

A C “source program” is a collection of any number of directives, pragmas, declarations, definitions, and statements. These constructs are discussed briefly in the following paragraphs. To be valid constructs in Microsoft C, each must have the syntax described in this manual, though they can appear in any order in the program (subject to the rules outlined throughout this manual). However, order of appearance does affect how variables and functions can be used in a program. (See “Lifetime and Visibility,” for more information.)

Directives

A “directive” instructs the C preprocessor to perform a specific action on the text of the program before compilation.

Pragmas

A “pragma” instructs the compiler to perform a particular action at compile time.

Declarations and Definitions

A “declaration” establishes an association between the name and the attributes of a variable, function, or type. In C, all variables must be declared before being used.

A “definition” of a variable establishes the same associations as a declaration, but also causes storage to be allocated for the variable. Therefore, all definitions are implicitly declarations, but not all declarations are definitions. For example, variable declarations that begin with the **extern** storage-class specifier are “referencing,” rather than “defining,” declarations. Referencing declarations do not cause storage to be allocated and cannot be initialized. (For more information on storage classes, see the “Declarations” chapter.)



C Language Reference

“Function declarations” (or “prototypes”) establish the name of the function, its return type, and, optionally, its formal parameters. A function definition includes the same elements as the prototype, plus the function body. If you do not supply an explicit declaration for a function, the compiler constructs a prototype from whatever information is available in the first reference to the function, whether that is a definition or a call.

Both function and variable declarations may appear inside or outside a function definition. Any declaration within a function definition is said to appear at the “internal” (local) level. A declaration outside all function definitions is said to appear at the “external” (global) level.

Variable definitions, like declarations, can appear at the internal level or at the external level. Function definitions always occur at the external level.

Note that declarations of types (for example, structure, union, and **typedef** declarations) that do not include the name of a variable of the type being declared do not cause storage allocation.

Example

The following example illustrates a simple C source program. This source program defines the function named *main* and declares the function named *printf* with a prototype. The program uses defining declarations to initialize the global variables *x* and *y*. The local variables *z* and *w* are declared, but not initialized. Storage is allocated for all these variables, but only *x*, *y*, *u*, and *v* contain meaningful values when declared because they are initialized either explicitly or implicitly. The values in *z* and *w* are not meaningful until values are assigned to them in the executable statements.

```

int x = 1;                /* Defining declarations */
int y = 2;                /* of external variables */

extern int printf(char *,...); /* Function "prototype"
                               or declaration */

main ()                  /* Function definition
                           for main function */
{
    int z;                /* Definitions for
                           */
    int w;                /* two uninitialized
                           */
                           /* local variables */

    static int v;        /* Definition of variable
                           with global lifetime */
    extern int u;        /* Referencing declaration
                           of external variable
                           defined elsewhere */

    z = y + x;           /* Executable statements */
    w = y - x;
    printf("z= %d    w= %d", z, w);
    printf("v= %d    u= %d", v, u);
}

```

3.3 Source Files

A source program can be divided into one or more “source files.” A C source file is a text file containing all or part of a C source program. (For example, a source file may contain just a few of the functions that the program needs.) When you compile a program, you must separately compile, and then link, the individual source files composing the total program. You can also use the **#include** directive to combine separate source files into larger source files before you compile. (For information on “include” files, see the “Preprocessor Directives and Pragmas” chapter.)

A source file can contain any combination of complete directives, pragmas, declarations, and definitions. You cannot split items such as function definitions or large data structures between source files. The last character in a source file must be a new-line character.

A source file need not contain executable statements. For example, you may find it useful to place definitions of variables in one source file and then declare references to these variables in other source files that use them. This technique makes the definitions easy to find and change. For the same reason, manifest constants and macros are often organized into separate include files that may be referenced in source files as required.

C Language Reference

Directives in a source file apply only to that source file and its include files. Moreover, each directive applies only to the part of the file that follows the directive. To apply a common set of directives to a whole source program, you must include the directives in all source files that make up the program.

Pragmas usually affect a specific region of a source file. The implementation determines the specific compiler action that a pragma defines. (Your compiler guide describes the effects of particular pragmas.)

Example

The following example illustrates a C source program contained in two source files. Once you have compiled these source files, you can link and then execute them as a single program.

The *main* and *max* functions are assumed to be in separate files, and execution of the program is assumed to begin with the *main* function.

```
/******  
   Source file 1 - main function  
   *****/  
  
#define ONE      1  
#define TWO      2  
#define THREE    3  
  
extern int max(int a, int b);    /* Function prototype */  
  
main ()                          /* Function definition */  
{  
    int w = ONE, x = TWO, y = THREE;  
    int z = 0;  
    z = max(x,y);  
    w = max(z,w);  
}
```

In Source file 1 (above), a prototype of the *max* function is declared. This kind of declaration is sometimes called a “forward declaration.” The definition for the *main* function includes calls to *max*.

The lines beginning with a number sign (#) are preprocessor directives. These directives tell the preprocessor to replace the identifiers *ONE*, *TWO*, and *THREE* with the corresponding number throughout Source file 1. However, the directives do not apply to Source file 2 (follows), which will be separately compiled and then linked with Source file 1.

```

/*****
    Source file 2 - definition of max function
    *****/

int max (int a, int b)      /* Note formal parameters are
                           included in function header */
{
    if ( a > b )
        return (a);
    else
        return (b);
}

```

Source file 2 contains the function definition for *max*. This definition satisfies the calls to *max* in Source file 1. Note that the definition for *max* follows the form specified in the ANSI C standard. For more information on this new form and function prototyping, see the “Functions” chapter.

3.4 Functions and Program Execution

Every C program has a primary function that must be named **main**. The **main** function serves as the starting point for program execution. It usually controls program execution by directing the calls to other functions in the program. A program usually stops executing at the end of **main**, although it can terminate at other points in the program for a variety of reasons depending on the execution environment.

The source program usually has more than one function, with each function designed to perform one or more specific tasks. The **main** function can call these functions to perform their respective tasks. When **main** calls another function, it passes execution control to the function so that execution begins at the first statement in the function. The function returns control when a **return** statement is executed or when the end of the function is reached.

You can declare any function, including **main**, to have parameters. When one function calls another, the called function receives values for its parameters from the calling function. These values are called “arguments.” You can declare formal parameters to **main** so that it can receive values from outside the program. (Most commonly, these arguments are passed from the command line when the program is executed.)

Traditionally, the first three parameters of the **main** function are declared to have the names *argc*, *argv*, and *envp*. The *argc* parameter is declared to hold the total number of arguments passed to **main**. The *argv* parameter is declared as an array of pointers, each element of which points to a

C Language Reference

string representation of an argument passed to **main**. The *envp* parameter is a pointer to a table of string values that set up the environment in which the program executes.

The operating system supplies values for the *argc*, *argv*, and *envp* parameters, and the user supplies the actual arguments to **main**. The operating system, not the C language, determines the argument-passing convention used on a particular system. For more information, see your compiler guide.

If you declare formal parameters to a function, you must declare them when you define the function. Function declarations are described in the “Declarations,” and “Functions” chapters, including function definitions.

3.5 Lifetime and Visibility

To understand how a C program works, you must understand the rules that determine how variables and functions can be used in the program. Three concepts are crucial to understanding these rules: the “block” (or compound statement), “lifetime” (sometimes called extent), and “visibility” (sometimes called scope).

3.5.1 Blocks

A block is a sequence of declarations, definitions, and statements enclosed within braces. There are two types of blocks in C. The “compound statement” (discussed more fully in the “Statements” chapter) is one type of block. The other, the “function definition,” consists of a compound statement comprising the function body plus the function’s associated “header” (the function name, return type, and formal parameters). A block may encompass other blocks, with the exception that no block can contain a function definition. A block within other blocks is said to be “nested” within the encompassing blocks.

Note that, while all compound statements are enclosed within braces, not everything enclosed within braces constitutes a compound statement. For example, though the specifications of array, structure, or enumeration elements may appear within braces, they are not considered compound statements.

3.5.2 Lifetime

“Lifetime” is the period, during execution of a program, in which a variable or function exists. All functions in a program exist at all times during its execution.

Lifetime of a variable may be internal (local) or external (global). An item with a local lifetime (a “local item”) has storage and a defined value only within the block where the item is defined or declared. A local item is allocated new storage each time the program enters that block, and it loses its storage (and hence its value) when the program exits the block. If the lifetime of the variable is global (a “global item”), it has storage and a defined value for the entire duration of a program.



The following rules specify whether a variable has local or global lifetime:

- Variables declared at the internal level (that is, within a block) usually have local lifetimes. You can ensure global lifetime for a variable within a block by including the **static** storage class specifier in its declaration. Once declared **static**, the variable will retain its value from one entry of the block to the next. However, it will still be “visible” only within its own block and blocks nested within its own block. (Visibility of objects is discussed in the next section. For information on storage-class specifiers, see the “Declarations” chapter.)
- Variables declared at the external level (that is, outside all blocks in the program) always have global lifetimes.

3.5.3 Visibility

An item’s “visibility” determines the portions of the program in which it can be referenced by name. An item is visible only in portions of a program encompassed by its “scope,” which may be limited (in order of increasing restrictiveness) to the file, function, block, or function prototype in which it appears.

In C, only a label name is always confined to function scope. (For more information on labels and label names, see the chapter on “Statements.”) The scope of any other item is determined by the level at which its declaration occurs. An item declared at the external level has file scope and is visible everywhere within the file. If its declaration occurs within a block (including the list of formal parameters in a function definition), the item’s scope is limited to that block and blocks nested within that block. Formal parameter names declared in the parameter list of a function

C Language Reference

prototype have scope only from the completion of the parameter declaration to the end of the function declarator.

Note

Although an item with a global lifetime *exists* throughout the execution of the source program (for example, an externally declared variable or a local variable declared with the **static** keyword), it may not be visible in all parts of the program.

An item is said to be “globally visible” if it is visible, or if you can use appropriate declarations to make it visible, in all the source files making up the program. (For more information on visibility between source files, also known as “linkage,” see the chapter on “Declarations.”)

The following rules govern the visibility of variables and functions within a program:

- Variables declared or defined at the global level (that is, outside all blocks in the program) are visible from their point of definition or declaration to the end of the source file. You can use appropriate declarations to make such variables visible in other source files, as described in “Storage Classes.” However, variables declared at the global level with the **static** storage-class specifier are visible only within the source file in which they are defined.
- In general, variables declared or defined at the local level (that is, within a block) are visible only from their point of declaration or definition to the end of the block actually containing the definition or declaration.
- Variables from outer blocks (including those declared at the global level) are visible in all inner blocks. However, the visibility of variables is said to “nest” within blocks. For instance, a block within another block can contain declarations for variables whose identifiers (names) are the same as variables in enclosing blocks. Such redefinitions prevail only within the inner block, however. Outer-block definitions are restored as the inner blocks are exited.
- Functions with **static** storage class are visible only in the source file in which they are defined. All other functions are globally visible. (For more information on function declarations, see “Function Definitions (Prototypes).”)

3.5.4 Summary

Table 3.1 summarizes the main factors determining lifetime and visibility of variables and functions. However, the table does not cover all possible cases. For more information, see the “Declarations” chapter.

Note

A Microsoft extension to the ANSI C standard provides that functions declared at an internal level may have global visibility. This feature should not be relied upon where portability of source code is a consideration. See your compiler guide for information on enabling Microsoft extensions.



C Language Reference

Table 3.1
Summary of Lifetime and Visibility

Level	Item	Storage Class Specifier	Lifetime	Visibility
External	Variable definition	static	Global	Restricted to source file in which it occurs
	Variable declaration	extern	Global	Remainder of source file
	Function prototype or definition	static	Global	Restricted to single source file
	Function prototype	extern	Global	Remainder of source file
Internal	Variable declaration	extern	Global	Block
	Variable definition	static	Global	Block
	Variable definition	auto or register	Local	Block

Example

The following program example illustrates blocks, nesting, and visibility of variables. In the example, there are four levels of visibility: the external level and three block levels. Assuming that the function *printf* is defined elsewhere in the program, the values will be printed to the screen as noted in the comments preceding each statement.

```

#include <stdio.h>

/* i defined at external level: */
int i = 1;

/* main function defined at external level: */
main ()
{

    /* prints 1 (value of external level i): */
    printf("%d\n", i);

    /* begin first nested block: */
    {

        /* i and j defined at internal level: */
        int i = 2, j = 3;

        /* prints 2, 3: */
        printf("%d\n%d\n", i, j);

        /* begin second nested block: */
        {

            /* i is redefined: */
            int i = 0;

            /* prints 0, 3: */
            printf("%d\n%d\n", i, j);

        /* end of second nested block: */
        }

        /* prints 2 (outer definition restored): */
        printf("%d\n", i);

    /* end of first nested block: */
    }

    /* prints 1 (external level definition restored): */
    printf("%d\n", i);
}

```

3.6 Naming Classes

In any C program, identifiers are used to refer to many different kinds of items. When you write a C program, you provide identifiers for the functions, variables, formal parameters, union members, and other items the program uses. C lets you use the same identifier for more than one program item, as long as you follow the rules outlined in this section. (For a definition of an identifier, see the “Elements of C” chapter.)

C Language Reference

The compiler sets up “naming classes” to distinguish between the identifiers used for different kinds of items. The names within each class must be unique to avoid conflict, but an identical name can appear in more than one naming class. This means that you can use the same identifier for two or more different items, provided that the items are in different naming classes. The compiler can resolve references based on the context of the identifier in the program.


The following list describes the kinds of items you can name in C programs and the rules for naming them:

Variables and functions The names of variables and functions are in a naming class with formal parameters, **typedef** names and enumeration constants. Therefore, variable and function names must be distinct from other names in this class that have the same visibility.

However, you can redefine variable and function names within program blocks, as described in “Lifetime and Visibility.”

Formal parameters The names of formal parameters to a function are grouped with the names of the function’s variables, so the formal parameter names should be distinct from the variable names. You cannot redeclare the formal parameters at the top level of the function. However, the names of the formal parameters may be redefined (that is, used to refer to different items) within subsequent blocks nested within the function body.

Enumeration constants Enumeration constants are in the same naming class as variable and function names. This means that the names of enumeration constants must be distinct from all variable and function names with the same visibility, and distinct from the names of other enumeration constants with the same visibility. However, like variable names, the names of enumeration constants have nested visibility, so you can redefine them within blocks. (Nested visibility is discussed in “Lifetime and Visibility.”)

typedef names	The names of types defined with the typedef keyword are in a naming class with variable and function names. Therefore, typedef names must be distinct from all variable and function names with the same visibility, as well as from the names of formal parameters and enumeration constants. Like variable names, names used for typedef types can be redefined within program blocks. See “Lifetime and Visibility.”
Tags	Enumeration, structure, and union tags are grouped in a single naming class. These tags must be distinct from other tags with the same visibility. Tags do not conflict with any other names. 
Members	The members of each structure and union form a naming class. The name of a member must, therefore, be unique within the structure or union, but it does not have to be distinct from other names in the program, including the names of members of different structures and unions.
Statement labels	Statement labels form a separate naming class. Each statement label must be distinct from all other statement labels in the same function. Statement labels do not have to be distinct from other names or from label names in other functions.

Example

Since structure tags, structure members, and variable names are in three different naming classes, the three items named *student* in the following example do not conflict. The context of each item allows correct interpretation of each occurrence of *student* in the program.

C Language Reference

For example, when *student* appears after the **struct** keyword, the compiler recognizes it as a structure tag. When *student* appears after a member-selection operator (-> or .), the name refers to the structure member. In other contexts, *student* refers to the structure variable.

```
struct student {
    char student[20];
    int class;
    int id;
} student;
```

Chapter 4

Declarations

- 4.1 Introduction 4-1
- 4.2 Type Specifiers 4-2
 - 4.2.1 Storage for Fundamental Types 4-5
 - 4.2.2 Range of Values 4-7
 - 4.2.3 Data-Type Categories 4-7
- 4.3 Declarators 4-8
 - 4.3.1 Array, Pointer, and Function Declarators 4-9
 - 4.3.2 Complex Declarators 4-9
 - 4.3.3 Declarators with Special Keywords 4-13
- 4.4 Variable Declarations 4-16
 - 4.4.1 Simple Variable Declarations 4-17
 - 4.4.2 Enumeration Declarations 4-18
 - 4.4.3 Structure Declarations 4-20
 - 4.4.4 Union Declarations 4-24
 - 4.4.5 Array Declarations 4-25
 - 4.4.6 Pointer Declarations 4-27
- 4.5 Function Declarations (Prototypes) 4-31
 - 4.5.1 Formal Parameters 4-32
 - 4.5.2 Return Type 4-32
 - 4.5.3 The List of Formal Parameters 4-33
 - 4.5.4 Summary 4-34
- 4.6 Storage Classes 4-37
 - 4.6.1 Variable Declarations at the Global Level 4-38
 - 4.6.2 Variable Declarations at the Local Level 4-41
 - 4.6.3 Function Declarations at the Global and Local Levels 4-44
- 4.7 Initialization 4-44
 - 4.7.1 Fundamental and Pointer Types 4-45
 - 4.7.2 Aggregate Types 4-47
 - 4.7.3 String Initializers 4-50

- 4.8 Type Declarations 4-51
 - 4.8.1 Structure, Union, and Enumeration Types 4-51
 - 4.8.2 Using typedef Declarations 4-52
- 4.9 Type Names 4-53

4.1 Introduction

This chapter describes the form and constituents of C declarations for variables, functions, and types. C declarations have the form

```
[sc-specifier] [type-specifier] declarator[=initializer] [,declarator[=initializer]]...
```

where *sc-specifier* is a storage-class specifier; *type-specifier* is the name of a defined type; and *initializer* gives the value or sequence of values to be assigned to the variable being declared. The *declarator* is an identifier that can be modified with brackets ([]), asterisks (*), or parentheses (()).

You must explicitly declare all C variables before using them. You can declare a C function explicitly with a function prototype. If you do not provide a prototype, one is created automatically from whatever information is included in the first reference to the function, whether that reference is a definition or a call.

The C language includes a standard set of data types. You can add your own data types by declaring new ones based on types already defined. You can declare arrays, data structures, and pointers to both variables and functions.

C declarations require one or more “declarators.” A declarator is an identifier that can be modified with brackets ([]), asterisks (*), or parentheses (()) to declare an array, pointer, or function type, respectively. When you declare simple variables (such as character, integer, and floating-point items), or structures and unions of simple variables, the declarator is just an identifier.

Four storage-class specifiers are defined in C: **auto**, **extern**, **register**, and **static**. The storage-class specifier of a declaration affects how the declared item is stored and initialized and which parts of a program can reference the item. Location of the declaration within the source program and the presence or absence of other declarations of the variable are also important factors in determining the visibility of variables.

Function prototype declarations are presented in “Function Declarations (Prototypes)” in this chapter and in the “Functions” chapter of this guide. For information on function definitions, see the “Functions” chapter.



C Language Reference

4.2 Type Specifiers

The C language provides definitions for a set of basic data types, called “fundamental” types. Their names are listed in Table 4.1.

Table 4.1
Fundamental Types

Integral Types^a	Floating-Point Types	Other
char	float	void^c
int	double	const
short	long double^b	volatile^d
long		
signed		
unsigned		
enum		

^a The optional keywords **signed** and **unsigned** can precede any of the integral types, except **enum**, and can also be used alone as type specifiers, in which case they are understood as **signed int** and **unsigned int**, respectively. When used alone, the keyword **int** is assumed to be **signed**. When used alone, the keywords **long** and **short** are understood as **long int** and **short int**.

^b The **long double** type is semantically equivalent to **double**, but is syntactically distinct.

^c The keyword **void** has three uses: as a function return type, as an argument-type list for a function that will take no arguments, and to modify a pointer.

^d The **volatile** keyword is implemented syntactically, but not semantically.

Enumeration types are considered fundamental types.

Note

The **long float** type is no longer supported, and occurrences of it in old code should be changed to **double**.

The **signed char**, **signed int**, **signed short int**, and **signed long int** types, together with their **unsigned** counterparts and **enum**, are called “integral” types. The **float**, **double**, and **long double** type specifiers are referred to as “floating” or “floating-point” types. You can use any integral or floating-point type specifier in a variable or function declaration.

You can use the **void** type to declare functions that return no value or to declare a pointer to an unspecified type. When the keyword **void** occurs alone within the parentheses following a function name, it is not interpreted as a type specifier. In that context **void** indicates only that the function accepts no arguments. Function types are discussed in “Function Declarations (Prototypes).”

The **const** type specifier declares an object as nonmodifiable. The **const** keyword can be a modifier for any fundamental or aggregate type, or to modify a pointer to an object of any type. A **const** type specifier can modify a **typedef**. A declaration that includes the keyword **const** as a modifier of an aggregate type declarator indicates that each element of the aggregate type is unmodifiable. If an item is declared with only the **const** type specifier, its type is taken to be **const int**. A **const** object may be placed in a read-only region of storage.

The **volatile** type specifier declares an item whose value may legitimately be changed by something beyond the control of the program in which it appears. The **volatile** keyword can be used in the same circumstances as **const** (previously described). An item can be both **const** and **volatile**, in which case the item could not be legitimately modified by its own program, but could be modified by some asynchronous process. The **volatile** keyword is implemented syntactically, but not semantically.

You can create additional type specifiers with **typedef** declarations (see “Type Declarations”). When used in a declaration, such specifiers may only be modified by the **const** and **volatile** modifiers.

Type specifiers are commonly abbreviated, as shown in Table 4.2. Integral types are signed by default. Thus, if you omit the **unsigned** keyword from the type specifier, the integral type is signed, even if you do not specify the **signed** keyword.

C Language Reference

In some implementations, you can specify a compiler option that changes the default **char** type from signed to unsigned. When this option is in effect, the abbreviation **char** means the same as **unsigned char**, and you must use the **signed** keyword to declare a signed character value. Compiler options are described in your compiler guide.

Note

This manual generally uses the abbreviated forms of the type specifiers listed in Table 4.2 rather than the long forms, and it assumes that the **char** type is signed by default. Therefore, throughout this manual, **char** stands for **signed char**.

Table 4.2
Type Specifiers and Abbreviations

Type Specifier	Abbreviations
signed char ^a	char
signed int	signed, int
signed short int	short, signed short
signed long int	long, signed long
unsigned char ^b	--
unsigned int	unsigned
unsigned short int	unsigned short
unsigned long int	unsigned long
float	--
const int	const
volatile int	volatile
const volatile int	const volatile

^a When you make the **char** type unsigned by default (by specifying the appropriate compiler option), you cannot abbreviate **signed char**.

^b When you make the **char** type unsigned by default (by specifying the appropriate compiler option), you can abbreviate **unsigned char** as **char**.

4.2.1 Storage for Fundamental Types

Table 4.3 summarizes the storage associated with each fundamental type and gives the range of values that can be stored in a variable of each type. Since the **void** type specifier is only used to denote a function with no return value or a pointer to an unspecified type, it is not included in the table. Similarly, the table does not include **const** or **volatile** because a variable type modified by **const** or **volatile** retains its storage size and can contain any value within range for its fundamental type.

Table 4.3
Storage and Range of Values for Fundamental Types

Type	Storage	Range of Values (Internal)
char	1 byte	-128 to 127
int	implementation defined	
short	2 bytes	-32,768 to 32,767
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned char	1 byte	0 to 255
unsigned	implementation defined	
unsigned short	2 bytes	0 to 65,535
unsigned long	4 bytes	0 to 4,294,967,295
float	4 bytes	IEEE-standard notation; discussed below
double	8 bytes	IEEE-standard notation; discussed below
long double	8 bytes	IEEE-standard notation; discussed below

The **char** type stores the integer value of a member of the representable character set. That integer value is the ASCII code corresponding to the specified character. Since the **char** type is interpreted as a signed, 1-byte integer, a **char** variable can store values in the range -128 to 127, although only the values from 0 to 127 have character equivalents. Similarly, an **unsigned char** variable can store values in the range 0-255.

Note that the C language does not define the storage and range associated with the **int** and **unsigned int** types. Instead, the size of a signed or

C Language Reference

unsigned **int** item is the standard size of an integer on a particular machine. For example, on a 16-bit machine the **int** type is usually 16 bits, or 2 bytes. On a 32-bit machine the **int** type is usually 32 bits, or 4 bytes. Thus, the **int** type is equivalent to either the **short int** or the **long int** type, and the **unsigned int** type is equivalent to either the **unsigned short** or the **unsigned long** type, depending on the implementation.

The type specifiers **int** and **unsigned int** (or simply **unsigned**) define certain features of the C language (for instance, the **enum** type discussed later in “Type Declarations”). In these cases, the definitions of **int** and **unsigned int** for a particular implementation determine the actual storage.

Note

The **int** and **unsigned int** type specifiers are widely used in C programs because they let a particular machine handle integer values in the most efficient way for that machine. However, since the sizes of the **int** and **unsigned int** types vary, programs that depend on a specific **int** size may not be portable to other machines. To make programs more portable, you can use expressions with the **sizeof** operator instead of hard-coded data sizes. The actual sizes of **int** and **unsigned int** are discussed in your compiler guide.

Floating-point numbers use the IEEE (Institute of Electrical and Electronics Engineers, Inc.) format. Values with **float** type have 4 bytes, consisting of a sign bit, an 8-bit excess-127 binary exponent, and a 23-bit mantissa. The mantissa represents a number between 1.0 and 2.0. Since the high-order bit of the mantissa is always 1, it is not stored in the number. This representation gives a range of approximately 3.4E-38 to 3.4E+38 for type **float**.

Values with **double** type have 8 bytes. The format is similar to the **float** format except that it has an 11-bit excess-1023 exponent and a 52-bit mantissa, plus the implied high-order 1 bit. This format gives a range of approximately 1.7E-308 to 1.7E+308 for type **double**.

4.2.2 Range of Values

The range of values for a variable is bounded by the minimum and maximum values that can be represented *internally* in a given number of bits. However, because of C's conversion rules (discussed in detail in the "Expressions and Assignments" chapter), you cannot always use the maximum or minimum value for a constant of a particular type in an expression.

For example, the constant expression `-32768` consists of the arithmetic negation operator (`-`) applied to the constant value `32,768`. Since `32,768` is too large to represent as a **short int**, it is given the **long** type. Consequently, the constant expression `-32768` has **long** type. You can only represent `-32,768` as a **short int** by type-casting it to the **short** type. No information is lost in the type cast, since `-32,768` can be represented internally in 2 bytes.

Similarly, a value such as `65,000` can only be represented as an **unsigned short** by type-casting the value to **unsigned short** type or by giving the value in octal or hexadecimal notation. The value `65,000` in decimal notation is considered a signed constant. It is given the **long** type because `65,000` does not fit into a **short**. You can cast this **long** value to the **unsigned short** type without loss of information, since `65,000` can fit in 2 bytes when it is stored as an unsigned number.

Octal and hexadecimal constants may have either **signed** or **unsigned** type, depending on their size (see "Integer Constants," for more information). However, the method used to assign types to octal and hexadecimal constants ensures that they always behave like unsigned integers in type conversions.

4.2.3 Data-Type Categories

The C data types fall into two general categories, called scalar and aggregate. Scalar types include pointers and arithmetic types. Arithmetic types include all floating and integral types, as described in this section. Aggregate types include arrays and structures. Table 4.4 illustrates the categories of C data types.

4

Table 4.4
C Data-Type Categories

Categories	Data Types
Integral Types	char int short long signed
Arithmetic Types	unsigned enum
Scalar Types	enum
Floating Types	float double long double
Aggregate Types	Pointers Arrays Structures

4.3 Declarators

Syntax

identifier
declarator[[*constant-expression*]]
**declarator*
(declarator)

The C language lets you declare “arrays” of values, “pointers” to values, and “functions returning” values of specified types. You must use a “declarator” to declare these items.

A “declarator” is an identifier that may be modified by brackets ([]), asterisks (*), or parentheses (()) to declare an array, pointer, or function type, respectively. Declarators appear in the array, pointer, and function declarations described later in this chapter. The following section discusses the rules for forming and interpreting declarators.

4.3.1 Array, Pointer, and Function Declarators

When a declarator consists of an unmodified identifier, the item being declared has a base type. If the identifier is followed by brackets ([]), the type is modified to an *array* type. If asterisks (*) appear to the left of an identifier, the type is modified to a *pointer* type. If the identifier is followed by parentheses, the type is modified to a *function returning* type.

A declarator must include a type specifier to be a complete declaration. The type specifier gives the type of the elements of an array type, the type of object addressed by a pointer type, or the return type of a function.

The sections on array, pointer, and function declarations later in this chapter discuss each type of declaration in detail.

The following examples illustrate the simplest forms of declarators:

Example 1

This example declares an array of **int** values named *list*:

```
int list[20];
```

Example 2

The following example declares a pointer named *cp* to a **char** value:

```
char *cp;
```

Example 3

The following declares a function named *func*, with no arguments, that returns a **double** value:

```
double func(void);
```

4.3.2 Complex Declarators

You can enclose any declarator in parentheses to specify a particular interpretation of a complex declarator.

A “complex” declarator is an identifier qualified by more than one array, pointer, or function modifier. You can apply various combinations of



C Language Reference

array, pointer, and function modifiers to a single identifier. However, a declarator may not have the following illegal combinations:

- An array cannot have functions as its elements.
- A function cannot return an array or a function.

In interpreting complex declarators, brackets and parentheses (that is, modifiers to the right of the identifier) take precedence over asterisks (that is, modifiers to the left of the identifier). Brackets and parentheses have the same precedence and associate from left to right. After the declarator has been fully interpreted, the type specifier is applied as the last step. By using parentheses you can override the default association order and force a particular interpretation.

A simple way to interpret complex declarators is to read them from the inside out, using the following four steps:

1. Start with the identifier and look to the right for brackets or parentheses (if any).
2. Interpret these brackets or parentheses, then look to the left for asterisks.
3. For each right parenthesis you encounter, apply rules 1 and 2 to everything within the parentheses.
4. Apply the type specifier.

Example 1

In the following example, the steps are labeled in order and can be interpreted as follows:

1. The identifier *var* is declared as
2. a pointer to
3. a function returning
4. a pointer to
5. an array of 10 elements, which are
6. pointers to

7. **char** values.

```
char  * (* (*var) ()) [10];
  ^   ^ ^ ^ ^ ^ ^
  7   6 4 2 1 3 5
```

Examples 2 through 9 illustrate complex declarations further and show how parentheses can affect the meaning of a declaration.

Example 2

In the following example, the array modifier has higher priority than the pointer modifier, so *var* is declared to be an array. The pointer modifier applies to the type of the array elements; therefore, the array elements are pointers to **int** values.

```
/* array of pointers to int values */
int *var[5];
```

**Example 3**

In the following example, parentheses give the pointer modifier higher priority than the array modifier, and *var* is declared to be a pointer to an array of five **int** values.

```
/* pointer to array of int values */
int (*var)[5];
```

Example 4

Function modifiers also have higher priority than pointer modifiers, so this example declares *var* to be a function returning a pointer to a **long** value. The function is declared to take two **long** values as arguments.

```
/* function returning pointer to a long */
long *var(long, long);
```

Example 5

This example is similar to Example 3. Parentheses give the pointer modifier higher priority than the function modifier, and *var* is declared to

C Language Reference

be a pointer to a function that returns a **long** value. Again, the function takes two **long** arguments.

```
/* pointer to function returning long */
long (*var)(long, long);
```

Example 6

The elements of an array cannot be functions, but this example demonstrates how to declare an array of pointers to functions instead. In this example, *var* is declared to be an array of five pointers to functions that return structures with two members. The arguments to the functions are declared to be two structures with the same structure type, *both*. Note that the parentheses surrounding **var[5]* are required. Without them, the declaration is an illegal attempt to declare an array of functions, as shown here:

```
/* ILLEGAL */
struct both *var[5]( struct both, struct both );

/* array of pointers to functions
   returning structures */
struct both {
    int a;
    char b;
} (*var[5])( struct both, struct both );
```

Example 7

This example shows how to declare a function returning a pointer to an array, since functions returning arrays are illegal. Here *var* is declared to be a function returning a pointer to an array of three **double** values. The function *var* takes one argument. The argument, like the return value, is a pointer to an array of three **double** values. The argument type is given by a complex abstract declarator. The parentheses around the asterisk in the argument type are required; without them, the argument type would be an array of three pointers to **double** values. For a discussion and examples of abstract declarators, see “Type Names.”

```
/* function returning pointer
   to an array of 3 double values */
double ( *var( double (*)[3] ) )[3];
```

Example 8

As this example shows, a pointer can point to another pointer, and an array can contain arrays as elements. Here *var* is an array of five elements. Each element is a five-element array of pointers that point to unions, each of which have two members.

```

        /* array of arrays of pointers
           to pointers to unions */

union sign {
    int x;
    unsigned y;
} **var[5][5];

```

Example 9

This example shows how the placement of parentheses changes the meaning of the declaration. In this example, *var* is a five-element array of pointers to five-element arrays of pointers to unions.

```

        /* array of pointers to arrays
           of pointers to unions */

union sign *(*var[5])[5];

```

4

4.3.3 Declarators with Special Keywords

Your implementation of Microsoft C may include the following special keywords:

1. **cdecl**
2. **far**
3. **fortran**
4. **huge**
5. **near**
6. **pascal**

C Language Reference

These keywords modify the meaning of variable and function declarations. See your compiler guide for a full discussion of the effects of these special keywords. Note that the keywords *near*, *far*, and *huge*, apply only to 80286 programs.

When a special keyword appears in a declarator, it modifies the item immediately to the right of the keyword. You can apply more than one special keyword to the same item. For example, you might modify a function identifier with both the **far** keyword and the **pascal** keyword. In this case, the order of the keywords does not matter (that is, **far pascal** and **pascal far** have the same effect). Thus the “binding” characteristics of the special keywords are the same as those of the type specifiers **const** and **volatile**. (The section “Type Specifiers,” contains descriptions of the **const** and **volatile** keywords.)

You can also use two or more special keywords in different parts of a declaration to modify the meaning of the declaration. For example, the following declaration contains two occurrences of the **far** keyword:

```
int far * pascal far func(void);
```

In this example, the **pascal** and **far** keywords modify the function identifier *func*. The return value of *func* is declared to be a **far** pointer to an **int** value.

As in any C declaration, you can use parentheses to override the default interpretation of the declaration. The rules governing complex declarators also apply to declarations that use the special keywords.

The following examples show the use of special keywords in declarations.

Example 1

This example declares a **huge** array named *database* with 65,000 **int** elements. The **huge** keyword modifies the array declarator.

```
int huge database[65000];
```

Example 2

In this example, the **far** keyword modifies the asterisk to its right, making *x* a **far** pointer to a pointer to **char**.

```
char * far * x;
```

This declaration is equivalent to the following declaration:

```
char * (far *x);
```

Example 3

This example shows two equivalent declarations. Both declare *calc* as a function with the **near** and **cdecl** attributes.

```
double near cdecl calc(double,double);
double cdecl near calc(double,double);
```

Example 4

Example 4 also shows two declarations. The first declares a **far fortran** array of characters named *initlist*, and the second declares three **far** pointers named *nextchar*, *prevchar*, and *currentchar*. These pointers might be used to store the addresses of characters in the *initlist* array. Note that the **far** keyword must be repeated before each declarator.

```
char far fortran initlist[INITSIZE];
char far *nextchar, far *prevchar, far *currentchar;
```

Example 5

Example 5 shows a more complex declaration with several occurrences of the **far** keyword.

```
char  far  *(far *getint)(int far *);
 6      5      2      1  3      4
```

The following procedure would be used to interpret this declaration:

1. The identifier *getint* is declared as a
2. **far** pointer to
3. a function taking
4. a single argument that is a **far** pointer to an **int** value

C Language Reference

5. and returning a **far** pointer to a
6. **char** value.

Note that the **far** keyword always modifies the item immediately to its right.

4.4 Variable Declarations

Syntax

[sc-specifier] type-specifier declarator [, declarator]...

This section describes the form and meaning of variable declarations. In particular, it explains how to declare the following:

Simple variables	Single-value variables with integral or floating-point type
Enumeration variables	Simple variables with integral type that hold one value from a set of named integer constants
Structures	Variables composed of a collection of values that may have different types
Unions	Variables composed of several values of different types, which occupy the same storage space
Arrays	Variables composed of a collection of elements with the same type
Pointers	Variables that point to other variables and contain variable locations (in the form of addresses) instead of values

In the general form of a variable declaration, *type-specifier* gives the data type of the variable and *declarator* gives the name of the variable, possibly modified to declare an array or a pointer type. The *type-specifier* can be a compound, as when the type is modified by **const**, **volatile**, or one of the special keywords described in “Declarators with Special Keywords.” You can define more than one variable in a declaration by using multiple declarators, separated by commas. For example, *int const far *fp* declares a variable named *fp* as a far pointer to a nonmodifiable **int** value.

The *sc-specifier* gives the storage class of the variable. In some contexts, you can initialize variables at the time you declare them. For information about storage classes and initialization, see the sections on “Storage Classes” and “Initialization,” respectively.

4.4.1 Simple Variable Declarations

Syntax

```
[sc-specifier] type-specifier identifier [, identifier]...;
```

The declaration of a simple variable specifies the variable’s name and type. It can also specify the variable’s storage class, as described in “Storage Classes.” The *identifier* in the declaration is the variable’s name. The *type-specifier* is the name of a defined data type.

You can use a list of identifiers separated by commas (,) to specify several variables in the same declaration. Each identifier in the list names a variable. All variables defined in the declaration have the same type.



Example 1

Example 1 declares a simple variable named *x*. This variable can hold any value in the set defined by the **int** type for a particular implementation. The simple object *y* is declared as a constant value of type **int**. It is initialized to the value 1, and is not modifiable. If the declaration of *y* was for an uninitialized external, it would receive an initial value of 0, and that value would be unmodifiable.

```
int x;
int const y=1;
```

Example 2

This example declares two variables named *reply* and *flag*. Both variables have **unsigned long** type and hold unsigned integral values.

```
unsigned long reply, flag;
```

Example 3

The following example declares a variable named *order* that has **double** type and can hold floating-point values.

```
double order;
```


4.4.2 Enumeration Declarations

Syntax

```
enum [tag] { enum-list } [declarator [, declarator]....];
```

```
enum tag [identifier [, declarator]....];
```

An “enumeration declaration” gives the name of an enumeration variable and defines a set of named integer constants (the “enumeration set”). A variable with enumeration type stores one of the values of the enumeration set defined by that type. The integer constants of the enumeration set have **int** type; thus, the storage associated with an enumeration variable is the storage required for a single **int** value.

Variables of **enum** type are treated as if they are of type **int** in all cases. They may be used in indexing expressions and as operands of all arithmetic and relational operators.

Enumeration declarations begin with the **enum** keyword and have the two forms shown at the beginning of this section and described below:

- In the first form, *enum-list* specifies the values and names of the enumeration set. (The *enum-list* is described in detail below.) The optional *tag* is an identifier that names the enumeration type defined by *enum-list*. The *declarator* names the enumeration variable. You can specify zero or more enumeration variables in a single enumeration declaration.
- The second form of the enumeration declaration uses a previously defined enumeration *tag* to refer to an enumeration type defined elsewhere. The *tag* must refer to a defined enumeration type, and that enumeration type must be currently visible. Since the enumeration type is defined elsewhere, *enum-list* does not appear in this type of declaration. Declarations of pointers to enumerations and **typedef** declarations for enumeration types can use the enumeration *tag* before the enumeration type is defined. However, the enumeration definition must be encountered prior to any actual use of the **typedef** declaration or pointer.

If a *tag* argument appears, but no *declarator* is given, the declaration constitutes a declaration for an enumeration tag.

An *enum-list* has the following form:

```
identifier [= constant-expression]  
[, identifier [= constant-expression] ... ]
```

Each *identifier* in an enumeration list names a value of the enumeration set. By default, the first identifier is associated with the value 0, the next identifier is associated with the value 1, and so on through the last identifier in the declaration. The name of an enumeration constant is equivalent to its value.

The optional phrase = *constant-expression* overrides the default sequence of values. Thus, if *identifier* = *constant-expression* appears in *enum-list*, the identifier is associated with the value given by *constant-expression*. The *constant-expression* must have `int` type and can be negative. The next identifier in the list is associated with the value of *constant-expression* + 1, unless you explicitly associate it with another value.

The following rules apply to the members of an enumeration set:

- An enumeration set can contain duplicate constant values. For example, you could associate the value 0 with two different identifiers named *null* and *zero* in the same set.
- The identifiers in the enumeration list must be distinct from other identifiers with the same visibility, including ordinary variable names and identifiers in other enumeration lists.
- Enumeration tags must be distinct from other enumeration, structure, and union tags with the same visibility.
- A comma is allowed following the last item in the enumeration list.

Example 1

This example defines an enumeration type named *day* and declares a variable named *workday* with that enumeration type. The value 0 is associ-



C Language Reference

ated with *saturday* by default. The identifier *sunday* is explicitly set to 0. The remaining identifiers are given the values 1 through 5 by default.

```
enum day {
    saturday,
    sunday = 0,
    monday,
    tuesday,
    wednesday,
    thursday,
    friday
} workday;
```

Example 2

In this example, a value from the set defined in Example 1 is assigned to the variable *today*. Note that the name of the enumeration constant is used to assign the value. Since the *day* enumeration type was previously declared, only the enumeration tag is necessary.

```
enum day today = wednesday;
```

4.4.3 Structure Declarations

Syntax

```
struct [tag] {member-declaration-list} [declarator [, declarator]...];
```

```
struct tag[declarator [, declarator]...];
```

A “structure declaration” names a structure variable and specifies a sequence of variable values (called “members” of the structure) that can have different types. A variable of that structure type holds the entire sequence defined by that type.

Structure declarations begin with the **struct** keyword and have two forms:

- In the first form, a *member-declaration-list* specifies the types and names of the structure members. The optional *tag* is an identifier that names the structure type defined by *member-declaration-list*.
- The second form uses a previously defined structure *tag* to refer to a structure type defined elsewhere. Thus, *member-declaration-list* is not needed as long as the definition is visible. Declarations of pointers to structures and **typedefs** for structure types can use the structure tag before the structure type is defined. However, the structure definition must be encountered prior to any actual use of the **typedef** or pointer.

In both forms, each *declarator* specifies a structure variable. A *declarator* can also modify the type of the variable to a pointer to the structure type, an array of structures, or a function returning a structure. If *tag* is given, but *declarator* does not appear, the declaration constitutes a type declaration for a structure tag.

Structure tags must be distinct from other structure, union, and enumeration tags with the same visibility.

A *member-declaration-list* argument contains one or more variable or bitfield declarations.

Each variable declared in the member-declaration list is defined as a member of the structure type. Variable declarations within the member-declaration list have the same form as other variable declarations discussed in this chapter, except that the declarations cannot contain storage-class specifiers or initializers. The structure members can have any variable type: fundamental, array, pointer, union, or structure.

4

A member cannot be declared to have the type of the structure in which it appears. However, a member can be declared as a pointer to the structure type in which it appears as long as the structure type has a tag. This lets you create linked lists of structures.

A bitfield declaration has the following form:

type-specifier [*identifier*] : *constant-expression*;

The *constant-expression* specifies the number of bits in the bitfield. The *type-specifier* has type **int** (**signed** or **unsigned**) and *constant-expression* must be a non-negative integer value. Arrays of bitfields, pointers to bitfields, and functions returning bitfields are not allowed. The optional *identifier* names the bitfield. Unnamed bitfields can be used as “dummy” fields, for alignment purposes. An unnamed bitfield whose width is specified as 0 guarantees that storage for the member following it in the member-declaration list begins on an **int** boundary.

Each *identifier* in a member-declaration list must be unique within the list. However, they do not have to be distinct from ordinary variable names or from identifiers in other member-declaration lists.

Note

A Microsoft extension to the ANSI C standard allows **char** and **long** types (both **signed** and **unsigned**) for bitfields. Unnamed bitfields with base type **long** or **char** (**signed** or **unsigned**) force alignment to a boundary appropriate to the base type.

Microsoft C does not implement **signed** bitfields. The syntax is allowed, but a bitfield specified as **signed** is treated as **unsigned** in all conversions.

Storage

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest. Storage for each member begins on a memory boundary appropriate to its type. Therefore, unnamed spaces (“holes”) may appear between structure members in memory.

Bitfields are not stored across boundaries of their declared type. For example, a bitfield declared with **unsigned int** type is packed into the space remaining (if any) if the previous bitfield was of type **unsigned int**. Otherwise, it begins a new object on an **int** boundary.

Example 1

This example defines a structure variable named *complex*. This structure has two members with **float** type, *x* and *y*. The structure type has no tag and is therefore unnamed.

```
struct {
    float x,y;
} complex;
```

Example 2

This example defines a structure variable named *temp*. The structure has three members: *name*, *id*, and *class*. The *name* member is a 20-element

array, and *id* and *class* are simple members with **int** and **long** type, respectively. The identifier *employee* is the structure tag.

```
struct employee {
    char name[20];
    int id;
    long class;
} temp;
```

Example 3

This example defines three structure variables: *student*, *faculty*, and *staff*. Each structure has the same list of three members. The members are declared to have the structure type *employee*, defined in Example 2.

```
struct employee student, faculty, staff;
```

Example 4

This example defines a structure variable named *x*. The first two members of the structure are a **char** variable and a pointer to a **float** value. The third member, *next*, is declared as a pointer to the structure type being defined (*sample*).

```
struct sample {
    char c;
    float *pf;
    struct sample *next;
} x;
```

Example 5

This example defines a two-dimensional array of structures named *screen*. The array contains 2000 elements. Each element is an individual structure containing four bitfield members: *icon*, *color*, *underline*, and *blink*.

```
struct {
    unsigned icon : 8;
    unsigned color : 4;
    unsigned underline : 1;
    unsigned blink : 1;
} screen[25][80];
```



4.4.4 Union Declarations

Syntax

```
union [tag] {member-declaration-list} [declarator [, declarator].*];
```

```
union tag[declarator [, declarator].*];
```

A “union declaration” names a union variable and specifies a set of variable values, called “members” of the union, that can have different types. A variable with **union** type stores one of the values defined by that type.

Union declarations have the same form as structure declarations, except that they begin with the **union** keyword instead of the **struct** keyword. The same rules govern structure and union declarations, except that bitfield members are not allowed in unions.

Storage

The storage associated with a union variable is the storage required for the largest member of the union. When a smaller member is stored, the union variable may contain unused memory space. All members are stored in the same memory space and start at the same address. The stored value is overwritten each time a value is assigned to a different member.

Example 1

This example defines a union variable with *sign* type and declares a variable named *number* that has two members: *svar*, a signed integer, and *uvar*, an unsigned integer. This declaration allows the current value of *number* to be stored as either a signed or an unsigned value. The tag associated with this union type is *sign*.

```
union sign {
    int svar;
    unsigned uvar;
} number;
```

Example 2

This example defines a union variable named *jack*. The members of the union are, in order of their declaration, a pointer to a **char** value, a **char** value, and an array of **float** values. The storage allocated for *jack* is the

storage required for the 20-element array *f*, since *f* is the longest member of the union. Because there is no tag associated with the union, its type is unnamed.

```
union {
    char *a, b;
    float f[20];
} jack;
```

Example 3

This example defines a two-dimensional array of unions named *screen*. The array contains 2000 elements. Each element of the array is an individual union with two members: *window1* and *screenval*. The *window1* member is a structure with two bitfield members, *icon* and *color*. The *screenval* member is an **int**. At any given time, each union element holds either the **int** represented by *screenval* or the structure represented by *window1*.

```
union {
    struct {
        unsigned int icon : 8;
        unsigned color : 4;
    } window1;
    int screenval;
} screen[25][80];
```

4

4.4.5 Array Declarations

Syntax

```
type-specifier declarator [constant-expression];
type-specifier declarator [ ];
```

An “array declaration” names the array and specifies the type of its elements. It may also define the number of elements in the array. A variable with array type is considered a pointer to the type of the array elements, as described in the section on “Identifiers.”

Array declarations have the two forms shown at the beginning of this section. Their syntax differs as follows:

- In the first form, the *constant-expression* argument within the brackets specifies the number of elements in the array. Each element has the type given by *type-specifier*, which can be any type except **void**. An array element cannot be a function type.

C Language Reference

- The second form omits the *constant-expression* argument in brackets. You can use this form only if you have initialized the array, declared it as a formal parameter, or declared it as a reference to an array explicitly defined elsewhere in the program.

In both forms, *declarator* names the variable and may modify the variable's type. The brackets ([]) following *declarator* modify the declarator to array type. You can declare an array of arrays (a "multidimensional" array) by following the array declarator with a list of bracketed constant expressions, as shown:

```
type-specifier declarator[constant-expression] [constant-expression] ...
```

Each *constant-expression* in brackets defines the number of elements in a given dimension: two-dimensional arrays have two bracketed expressions, three-dimensional arrays have three, and so on. When you declare a multidimensional array within a function, you can omit the first constant expression if you have initialized the array, declared it as a formal parameter, or declared it as a reference to an array explicitly defined elsewhere in the program.

You can define arrays of pointers to various types of objects by using complex declarators, as described in "Complex Declarators."

Storage

The storage associated with an array type is the storage required for all of its elements. The elements of an array are stored in contiguous and increasing memory locations, from the first element to the last. No blanks separate the array elements in storage.

Arrays are stored by row. For example, the following array consists of two rows with three columns each:

```
char A[2][3];
```

The three columns of the first row are stored first, followed by the three columns of the second row. This means that the last subscript varies most quickly.

To refer to an individual element of an array, use a subscript expression, as described in "Subscript Expressions."

Example 1

This example declares an array variable named *scores* with 10 elements, each of which has **int** type. The variable named *game* is declared as a simple variable with **int** type.

```
int scores[10], game;
```

Example 2

This example declares a two-dimensional array named *matrix*. The array has 150 elements, each having **float** type.

```
float matrix[10][15];
```

Example 3

This example declares an array of structures. This array has 100 elements; each element is a structure containing two members.

```
struct {
    float x,y;
} complex[100];
```

Example 4

This example declares the type and name of an array of pointers to **char**. The actual definition of *name* occurs elsewhere.

```
extern char *name[];
```

4.4.6 Pointer Declarations**Syntax**

```
type-specifier * [modification-spec] declarator;  
extern char *name [ ];
```

A “pointer declaration” names a pointer variable and specifies the type of the object to which the variable points. A variable declared as a pointer holds a memory address.

The *type-specifier* gives the type of the object, which can be any fundamental, structure, or union type. Pointer variables can also point to functions, arrays, and other pointers. (For information on declaring more complex pointer types, refer to the section on “Complex Declarators.”)



C Language Reference

By making *type-specifier* **void**, you can delay specification of the type to which the pointer refers. Such an item is referred to as a “pointer to **void**” (**void ***). A variable declared as a pointer to **void** can be used to point to an object of any type. However, in order to perform operations on the pointer or on the object to which it points, the type to which it points must be explicitly specified for each operation. Such conversion can be accomplished with a type cast.

The *modification-spec* can be either **const** or **volatile**, or both. These specify, respectively, that the pointer will not be modified by the program itself (**const**), or that the pointer may legitimately be modified by some process beyond the control of the program (**volatile**). (For more information on **const** and **volatile**, see “Type Specifiers.”)

The *declarator* names the variable and can include a type modifier. For example, if *declarator* represents an array, the type of the pointer is modified to pointer to array.

You can declare a pointer to a structure, union, or enumeration type before you define the structure, union, or enumeration type. However, the definition must appear before the pointer can be used as an operand in an expression. You declare the pointer by using the structure or union tag (see Example 7 in this section). Such declarations are allowed because the compiler does not need to know the size of the structure or union to allocate space for the pointer variable.

Storage

The amount of storage required for an address and the meaning of the address depend on the implementation of the compiler. Pointers to different types are not guaranteed to have the same length.

In some implementations, you can use the special keywords **near**, **far**, and **huge** to modify the size of a pointer. Declarations using special keywords are described in “Declarators with Special Keywords.” For more information on the meaning and use of these keywords, see your compiler guide.

Example 1

This example declares a pointer variable named *message*. It points to a variable with **char** type.

```
char *message;
```

Example 2

Example 2 declares an array of pointers named *pointers*. The array has 10 elements; each element is a pointer to a variable with **int** type.

```
int *pointers[10];
```

Example 3

This example declares a pointer variable named *pointer*; it points to an array with 10 elements. Each element in this array has **int** type.

```
int (*pointer)[10];
```

Example 4

This example declares a pointer variable, *x*, to a constant value. The pointer may be modified to point to a different **int** value, but the value to which it points may not be modified.

```
int const *x;
```

Example 5

The variable *y* in Example 5 is declared as a constant pointer to an **int** value. The value it points to may be modified, but the pointer itself must always point to the same location: the address of *fixed_object*. Similarly, *z* is a constant pointer, but it is also declared to point to an **int** whose value will not be modified by the program. The additional specifier *volatile* indicates that although the value of the **const int** pointed to by *z* cannot be modified by the program, it could legitimately be modified by a process outside the program. The declaration of *w* specifies that the value pointed to will not be changed and that the program itself will not modify the pointer. However, some outside process could legitimately modify the pointer.

```
const int some_object = 5 ;
int other_object = 37;
int *const y = &fixed_object;
const volatile *const z = &some_object;
*const volatile w = &some_object;
```

4

C Language Reference

Example 6

This example declares two pointer variables that point to the structure type *list*. This declaration can appear before the definition of the *list* structure type (see Example 7), as long as the *list* type definition has the same visibility as the declaration.

```
struct list *next, *previous;
```

Example 7

This example defines the variable *line* to have the structure type named *list*. The *list* structure type has three members: the first member is a pointer to a **char** value, the second is an **int** value, and the third is a pointer to another *list* structure.

```
struct list {
    char *token;
    int count;
    struct list *next;
} line;
```

Example 8

This example declares the variable *record* to have the structure type *id*. Note that *pname* is declared as a pointer to another structure type named *name*. This declaration can appear before the *name* type is defined.

```
struct id {
    unsigned int id_no;
    struct name *pname;
} record;
```

Example 9

In this example, the pointer variable *p* is declared, but the **void *** preceding the identifier *p* in the declaration means that *p* can be used later to point to any type object. The address of an **int** value is assigned to *p*, but no operations on the pointer itself are permitted unless it is explicitly converted to the type to which it points. Similarly, indirect operations on the object pointed to by *p* are not permitted unless *p* is converted to a specific type. Finally, a cast is used to convert *p* to a pointer to **int**, and *p* is then incremented.

```

int i;
void *p;          /* p declared as pointer to an object
                  whose type is not specified */
p = &i;          /* address of integer i assigned to p
                  but type of p itself is still not
                  specified. An operation like p++
                  would not be permitted yet */
(int *)p++;      /* incrementing p permitted when the
                  cast converts it to pointer to int */

```

4.5 Function Declarations (Prototypes)

Syntax

[sc-spec] [type-spec] declarator([formal-parameter-list]) [, declarator-list]...;

A “function declaration,” also called a “function prototype,” establishes the name and return type of a function and may specify the types, formal parameter names, and number of arguments to the function. A function declaration does not define the function body. It simply makes information about the function known to the compiler. This information enables the compiler to check the types of the actual arguments in ensuing calls to the function.

If you do not provide a function prototype, the compiler constructs one from the first reference to the function it encounters, whether a call or a function definition. Whether such a prototype reflects the correct parameter types can only be assured if the function definition occurs in the same source file. If the definition occurs in a different module, argument mismatch errors may not be detected. Function definitions are described in detail in “Function Prototypes (Declarations).”

The *sc-spec* represents a storage-class specifier; it can be either **extern** or **static**. Storage-class specifiers are discussed in “Storage Classes.”

The *type-spec* gives the function’s return type, and *declarator* names the function. If you omit *type-spec* from a function declaration, the function is assumed to return a value of type **int**.

The *formal-parameter-list* is described in the next section.

The final *declarator-list* in the syntax line represents further declarations on the same line. These may be other functions returning values of the

C Language Reference

same type as the first function, or declarations of any variables whose type is the same as the first function's return type. Each such declaration must be separated from its predecessors and successors by a comma.

4.5.1 Formal Parameters

“Formal parameters” describe the actual arguments that can be passed to a function. In a function declaration, the parameter declarations establish the number and types of the actual arguments. They may also include identifiers of the formal parameters. Though the parameters may be omitted from a function declaration, their inclusion is recommended, and they are mandatory in a true prototype. The extent of the information in the declaration influences the argument checking done on function calls that appear before the compiler has processed the function definition.

Note

Identifiers used to name the formal parameters in the prototype declaration are descriptive only. They go out of scope at the end of the declaration. Therefore, they need not be identical to the identifiers used in the declaration portion of the function definition. Using the same names may enhance readability, but this use has no other significance.

4.5.2 Return Type

Functions can return values of any type except arrays and functions. Therefore, the *type-specifier* argument of a function declaration can specify any fundamental, structure, or union type. You can modify the function identifier with one or more asterisks (*) to declare a pointer return type.

Although functions cannot return arrays and functions, they can return pointers to arrays and functions. You can declare a function that returns a pointer to an array or function type by modifying the function identifier with asterisks (*), brackets ([]), and parentheses (()). Such a function identifier is known as a “complex declarator.” Rules for forming and interpreting complex declarators are discussed in “Complex Declarators.”

4.5.3 The List of Formal Parameters

All elements of the *formal-parameter-list* argument appearing within the parentheses following the function declarator are optional. The two following syntax variations illustrate the possibilities:

```
[void]
[register] [type-spec] [declarator[[, ...][, ...]]]
```

If formal parameters are omitted from the function declaration, the parentheses should contain the keyword **void** to specify that no arguments will ever be passed to the function. If the parentheses are left entirely empty, no information is conveyed about whether arguments will be passed to the function and no checking of argument types is performed.

Note

Empty parentheses in a function declaration or definition represent an obsolete form not recommended for new code. Functions accepting no arguments should be declared with the **void** keyword replacing the list of formal parameters. This use of **void** is interpreted by context, and is distinct from uses of **void** as a type specifier.

A declaration in the list of formal parameters can contain the **register** storage-class specifier, either alone or combined with a type specifier and an identifier. If **register** is not specified, the storage class is **auto**. The only explicit storage-class specifier permitted is **register**. If the parentheses contain only the **register** keyword, the formal parameter is considered to represent an unnamed **int** for which **register** storage is being requested.

If *type-spec* is included, it can specify the type name for any fundamental, structure, or union type (such as **int**). A *declarator* for a fundamental, structure, or union type is simply an identifier of a variable having that type.

The *declarator* for a pointer, array, or function can be formed by combining a type specifier, plus the appropriate modifier, with an identifier. Alternatively, an “abstract declarator” (that is, a declarator without a specified identifier) can be used. The section “Type Names” explains how to form and interpret abstract declarators.

A full, partial, or empty list of formal parameters can be declared. If the list contains at least one declarator, a variable number of parameters can



C Language Reference

be specified by ending the list with a comma followed by three periods (,...), referred to as the “ellipsis notation.” A function is expected to have at least as many arguments as there are declarators or type specifiers preceding the last comma.

Note

To maintain compatibility with previous versions, the compiler accepts a comma without trailing periods at the end of a declarator list to indicate a variable number of arguments. However, this is a Microsoft extension to the ANSI C standard. New code should use the comma followed by three periods. For information on enabling and disabling extensions, see your compiler guide.

One other special construction is permitted as a formal parameter: **void *** represents a pointer to an object of unspecified type. Thus, in a call, the pointer can refer to any type of object after you convert the pointer (for example, with a cast) to a pointer to the desired type. Note that before operations can be performed on the pointer or the object it addresses, the pointer must be explicitly converted. For more information on **void ***, see “Pointer Declarations.”

4.5.4 Summary

Function prototypes are optional, but strongly recommended. If included, the only elements absolutely required are the name of the function, the opening and closing parentheses following the name, and the final semicolon. If no return type is included, as in the following example, the function is assumed to return an **int**:

```
/***** Obsolete form of function declaration *****/  
  
minimal_declaration();      /* may or may not  
                             accept arguments */
```

A full function prototype is the same as a function definition, except that instead of having a function “body,” it is terminated by a semicolon (;) immediately following the closing parenthesis.

Any appropriate combination of elements is permitted among the parameter declarations, from no information (as in the obsolete form in the example above) to a full prototype of the function. If no prototype at all is

given, a *de facto* prototype is constructed from information in the first reference to the function encountered in the source file.

Example 1

In this example, any information included in the formal parameter list is used to check actual arguments appearing in calls to the function that occur before the compiler has processed the function definition.

```
double func(void);      /* returns a double, but
                        * accepts no arguments
                        */
fun (void *);          /* takes a pointer to an
                        * unspecified type;
                        * returns an int
                        */
char *true(long, long); /* takes two longs;
                        * returns pointer to char
                        */
new (register a, char *); /* takes an int with request
                        * for register storage, and
                        * a pointer to char;
                        * returns an int
                        */
void go(int *[], char *b); /* takes an array of pointers
                        * to int using an abstract
                        * declarator, and a pointer
                        * to char; there is no return
                        */
void *tu(double v,...); /* takes at least one double;
                        * other arguments may also be
                        * given; returns a pointer
                        * to an unspecified type
                        */
```

4

Example 2

This example is a prototype for a function named *add* that takes two *int* arguments, represented by the identifiers *num1* and *num2*, and returns an *int* value.

```
int add(int num1, int num2);
```

C Language Reference

Example 3

This example declares a function named *calc* that returns a **double** value. The obsolete empty parentheses leave the issue of possible arguments to the function undefined.

```
double calc();
```

Example 4

This example is a prototype for a function named *strfind* that returns a pointer to **char**. The function accepts at least one argument, declared by the formal parameter *char *ptr*, to be a pointer to a **char** value. The formal parameter list has one entry and ends with a comma followed by three periods, indicating that the function may take more arguments.

```
char *strfind(char *ptr, ...);
```

Example 5

This example declares a function with **void** return type (returning no value). The **void** keyword also replaces the list of formal parameters, so no arguments are expected for this function.

```
void draw(void);
```

Example 6

In this example, *sum* is declared as a function returning a pointer to an array of three **double** values. The *sum* function takes two **double** values as arguments.

```
double (*sum(double, double))[3];
```

Example 7

In this example, the function named *select* is declared to take no arguments and to return a pointer to a function. The pointer return value points to a function taking one **int** argument, represented by the identifier *number*, and returning an **int** value.

```
int (*select(void))(int number);
```

Example 8

In this example, the function *prt* is declared to take a pointer argument of any type and return an **int** value. A pointer to any type could be passed as an argument to *prt* without producing a type-mismatch warning.

```
int prt(void *);
```

Example 9

This example shows the declaration of an array, named *rainbow*, of an unspecified number of constant pointers to functions. Each of these takes at least one parameter of type **int**, as well as an unspecified number of other parameters. Each of the functions pointed to returns a **long** value.

```
long (*const rainbow[]) (int, ...) ;
```

**4.6 Storage Classes**

The “storage class” of a variable determines whether the item has a “local” or “global” lifetime. Variables with local lifetimes are allocated new storage each time execution control passes to the block in which they are defined. When execution control passes out of the block, the variables no longer have meaningful values.

An item with a global lifetime exists and has a value throughout the execution of the program. All functions have global lifetimes.

Although C defines only two types of storage classes, it provides the following four storage-class specifiers:

Table 4.5
Storage-Class Specifiers

<u>Items declared with</u>	<u>Have a</u>
auto	local lifetime
register	local lifetime
static	global lifetime
extern	global lifetime

The four storage-class specifiers have distinct meanings because they affect the visibility of functions and variables, as well as their storage class. The term “visibility” refers to the portion of the source program in which the variable or function can be referenced by name. An item with a global lifetime exists throughout the execution of the source program, but it may not be “visible” in all parts of the program. (Visibility and the related concept of lifetime are discussed in the chapter on “Program Structure.”)

The placement of variable and function declarations within source files also affects storage class and visibility. Declarations outside all function definitions are said to appear at the “external level”; declarations within function definitions appear at the “internal level.”

The exact meaning of each storage-class specifier depends on two factors:

- Whether the declaration appears at the external or internal level
- Whether the item being declared is a variable or a function

The sections that follow describe the meanings of storage-class specifiers in each kind of declaration and explain the default behavior when the storage-class specifier is omitted from a variable or function declaration.

4.6.1 Variable Declarations at the Global Level

In variable declarations at the global level (that is, outside all functions), you can use the **static** or **extern** storage-class specifier or omit the storage-class specifier entirely. You cannot use the **auto** and **register** storage-class specifiers at the global level.

Variable declarations at the global level are either *definitions* of variables (“defining declarations”), or *references* to variables defined elsewhere (“referencing declarations”).

A global variable declaration that also initializes the variable (implicitly or explicitly) is a defining declaration of the variable. A definition at the global level can take several forms:

- A variable that you declare with the **static** storage-class specifier. You can explicitly initialize the **static** variable with a constant expression, as described in “Initialization.” If you omit the initializer, the variable is initialized to 0 by default. For example, *static int k = 16;* and *static int k;* are both considered definitions of the variable *k*.
- A variable that you explicitly initialize at the global level. For example, *int j = 3;* is a definition of the variable *j*.

Once a variable is defined at the global level, it is visible throughout the rest of the source file in which it appears. The variable is not visible prior to its definition in the same source file. Also, it is not visible in other source files of the program, unless a referencing declaration makes it visible, as described later in this section.

You can define a variable at the global level only once within a source file. If you give the **static** storage-class specifier, you can define another variable with the same name and the **static** storage-class specifier in a different source file. Since each **static** definition is visible only within its own source file, no conflict occurs.

The **extern** storage-class specifier declares a *reference* to a variable defined elsewhere. You can use an **extern** declaration to make a definition in another source file visible, or to make a variable visible above its definition in the same source file. Once you have declared a reference to the variable at the global level, the variable is visible throughout the remainder of the source file in which the declared reference occurs.

Declarations that use the **extern** storage-class specifier cannot contain initializers, since these declarations refer to variables whose values are defined elsewhere.

For an **extern** reference to be valid, the variable it refers to must be defined once, and only once, at the global level. The definition can be in any of the source files that form the program.

C Language Reference

One special case is not covered by the rules outlined above. You can omit both the storage-class specifier and the initializer from a variable declaration at the global level; for example, the declaration `int n;` is a valid global declaration. This declaration can have one of two different meanings, depending on the context:

1. If there is a global defining declaration of a variable with the same name elsewhere in the program, the current declaration is assumed to be a reference to the variable in the defining declaration, exactly as if the **extern** storage-class specifier had been used in the declaration.
2. If there is no global declaration of a variable with the same name elsewhere in the program, the declared variable is allocated storage at link time and initialized to 0. This kind of variable is known as a “communal” variable. If more than one such declaration appears in the program, storage is allocated for the largest size declared for the variable. For example, if a program contains two uninitialized declarations of *i* at the global level, `int i;` and `char i;`, storage space for an **int** value is allocated for *i* at link time.

Uninitialized variable declarations at the global level are not recommended for any file that might be placed in a library.

Example

The two source files in this example contain a total of three global declarations of *i*. Only one declaration contains an initialization; that declaration, `int i = 3;`, defines the global variable *i* with initial value 3. The **extern** declaration of *i* at the top of the first source file makes the global variable visible above its definition in the file. Without the **extern** declaration, the *main* function could not reference the global variable *i*. The **extern** declaration of *i* in the second source file also makes the global variable visible in that source file.

Assuming that the *printf* function is defined elsewhere in the program, all three functions perform the same task: they increase *i* and print it. The values 4, 5, and 6 are printed.

If the variable *i* had not been initialized, it would have been set to 0 automatically at link time. In this case, the values 1, 2, and 3 would have been printed.

Source File One

```

extern int i;                /* reference to i,
                             defined below */

main()
{
    i++;
    printf("%d\n", i); /* i equals 4 */
    next();
}

int i = 3;                  /* definition of i */

next()
{
    i++;
    printf("%d\n", i); /* i equals 5 */
    other();
}

```

4

Source File Two

```

extern int i;                /* reference to i in
                             first source file */

other()
{
    i++;
    printf("%d\n", i); /* i equals 6 */
}

```

4.6.2 Variable Declarations at the Local Level

You can use any of the four storage-class specifiers for variable declarations at the local level. When you omit the storage-class specifier from such a declaration, the default storage class is **auto**.

The **auto** storage-class specifier declares a variable with a local lifetime. An **auto** variable is visible only in the block in which it is declared. Declarations of **auto** variables can include initializers, as discussed in “Initialization.” Since variables with **auto** storage class are not initialized automatically, you should either explicitly initialize them when you

C Language Reference

declare them, or assign them initial values in statements within the block. The values of uninitialized **auto** variables are undefined.

A **static auto** variable can be initialized with the address of any global or **static** item, but not with the address of another **auto** item, because the address of an **auto** item is not a constant.

The **register** storage-class specifier tells the compiler to give the variable storage in a register, if possible. Register storage usually speeds access time and reduces code size. Variables declared with **register** storage class have the same visibility as **auto** variables. The number of registers that can be used for variable storage is machine-dependent. If no registers are available when the compiler encounters a **register** declaration, the variable is given **auto** storage class and stored in memory. The compiler assigns register storage to variables in the order in which the declarations appear in the source file. Register storage, if available, is only guaranteed for **int** and pointer types that are the same size as an **int**.

A variable declared at the local level with the **static** storage-class specifier has a global lifetime but is visible only within the block in which it is declared. Unlike **auto** variables, **static** variables keep their values when the block is exited. You can initialize a **static** variable with a constant expression. A **static** variable is initialized only once, when program execution begins; it is *not* reinitialized each time the block is entered. If you do not explicitly initialize a **static** variable, it is initialized to 0 by default.

A variable declared with the **extern** storage-class specifier is a reference to a variable with the same name defined at the global level in any of the source files of the program. The local **extern** declaration is used to make the global-level variable definition visible within the block. Unless otherwise declared at the global level, a variable declared with the **extern** keyword is visible only in the block in which it is declared.

Example

In the following example, the variable *i* is defined at the global level with initial value 1. An **extern** declaration in the *main* function is used to declare a reference to the global-level *i*. The **static** variable *a* is initialized to 0 by default, since the initializer is omitted. The call to *printf* (assuming the *printf* function is defined elsewhere in the source program) prints the values 1, 0, 0, and 0.

In the *other* function, the address of the global variable *i* is used to initialize the **static** pointer variable *external_i*. This works because the global variable has **static** lifetime, meaning its address will always be the same. Next, the variable *i* is redefined as a local variable with initial value 16.

This redefinition does not affect the value of the global-level *i*, which is hidden by the use of its name for the local variable. The value of the global *i* is now accessible only indirectly within this block, through the pointer *external_i*. Attempting to assign the address of the **auto** variable *i* to a pointer would not work, since it may be different each time the block is entered. The variable *a* is declared as a **static** variable and initialized to 2. This *a* does not conflict with the *a* in *main*, since **static** variables at the local level are visible only within the block in which they are declared.

The variable *a* is increased by 2, giving 4 as the result. If the *other* function were called again in the same program, the initial value of *a* would be 4, since local **static** variables keep their values when the program exits and then re-enters the block in which they are declared.

```

int i = 1;

main()
{
    /* reference to i, defined above: */
    extern int i;

    /* initial value is zero; a is
       visible only within main: */
    static int a;

    /* b is stored in a register, if possible: */
    register int b = 0;

    /* default storage class is auto: */
    int c = 0;

    /* values printed are 1, 0, 0, 0: */
    printf("%d\n%d\n%d\n%d\n", i, a, b, c);
    other();
}

other()
{
    /* address of global i assigned to pointer variable */
    static int *external_i = &i;

    /* i is redefined; global i no longer visible: */
    int i = 16;

    /* this a is visible only within other: */
    static int a = 2;

    a += 2;
    /* values printed are 16, 4, and 1: */
    printf("%d\n%d\n%d\n", i, a, *external_i);
}

```

C Language Reference

4.6.3 Function Declarations at the Global and Local Levels

You can use either the **static** or the **extern** storage-class specifier in function declarations. Functions always have global lifetimes.

The visibility rules for functions vary slightly from the rules for variables, as follows:

- A function declared to be **static** is visible only within the source file in which it is defined. Functions in the same source file can call the **static** function, but functions in other source files cannot. You can declare another **static** function with the same name in a different source file without conflict.
- Functions declared as **extern** are visible throughout all the source files that make up the program (unless you later redeclare such a function as **static**). Any function can call an **extern** function.
- Function declarations that omit the storage-class specifier are **extern** by default.

Note

A Microsoft extension to the ANSI C standard provides that function declarations at the local level have the same meaning as function declarations at the global level. This means that a function is visible from its point of declaration through the rest of the source file.

4.7 Initialization

Syntax

= initializer

You can set a variable to an initial value by applying an initializer to the declarator in the variable declaration. The value or values of the initializer are assigned to the variable. An equal sign (=) precedes the initializer.

You can initialize variables of any type, provided that you obey the following rules:

- Declarations that use the **extern** storage-class specifier cannot include initializers.
- Variables declared at the global level can be initialized. If you do not explicitly initialize a variable at the global level, it is initialized to 0 by default.
- A constant expression can be used to initialize any variable declared with the **static** storage-class specifier. Variables declared to be **static** are initialized when program execution begins. If you do not explicitly initialize a **static** variable, it is initialized to 0 by default.
- Variables declared with the **auto** and **register** storage-class specifiers are initialized each time execution control passes to the block in which they are declared. If you omit an initializer from the declaration of an **auto** or **register** variable, the initial value of the variable is undefined.
- Aggregate types with **auto** storage class (arrays, structures, and unions) cannot be initialized. Only **static** aggregates and aggregates declared at the global level can be initialized.
- The initial values for global variable declarations and for all **static** variables, whether global or local, must be constant expressions. You can use either constant or variable values to initialize **auto** and **register** variables.

The sections that follow describe how to initialize variables of fundamental, pointer, and aggregate types.

4.7.1 Fundamental and Pointer Types

Syntax

= *expression*

The value of *expression* is assigned to the variable. The conversion rules for assignment apply.

A locally declared static variable can only be initialized with a constant value. Since the address of any globally declared or static variable is constant, it may be used to initialize an local declared **static** pointer variable.

C Language Reference

However, the address of an **auto** variable cannot be used as an initializer because it may be different for each execution of the block.

Example 1

In this example, *x* is initialized to the constant expression 10.

```
int x = 10;
```

Example 2

In this example, the pointer *px* is initialized to 0, producing a “null” pointer.

```
register int *px = 0;
```

Example 3

This example uses a constant expression to initialize *c* to a constant value that cannot be modified.

```
const int c = (3 * 1024);
```

Example 4

This example initializes the pointer *b* with the address of another variable, *x*. The pointer *a* is initialized with the address of a variable named *z*. However, since it is specified to be a **const**, the variable *a* can only be initialized, never modified. It always points to the same location.

```
int *b = &x;  
int *const a = &z;
```

Example 5

The global variable *GLOBAL* is declared in Example 5 at the global level, so it has global lifetime. The local variable *LOCAL* has **auto** storage class and only has an address during the execution of the function in which it is declared. Therefore, attempting to initialize the **static** pointer variable *lp* with the address of *LOCAL* is not permitted. The **static** pointer variable *gp* can be initialized to the address of *GLOBAL* because that address is always the same. Similarly, **rp* can be initialized because *rp* is a local variable and can have a nonconstant initializer. Each time the block is entered, *LOCAL* will have a new address, which will then be assigned to *rp*.

```

int GLOBAL ;

int function(void)
{
    int LOCAL ;
    static int *lp = &LOCAL; /* Illegal declaration */
    static int *gp = &GLOBAL; /* Legal declaration */
    register int *rp = &LOCAL; /* Legal declaration */
}

```

4.7.2 Aggregate Types

Syntax

= {*initializer-list*}

The *initializer-list* is a list of initializers separated by commas. Each initializer in the list is either a constant expression or an initializer list. Therefore, an initializer list enclosed in braces can appear within another initializer list. This form is useful for initializing aggregate members of an aggregate type, as shown in the in this section.

For each *initializer-list*, the values of the constant expressions are assigned, in order, to the corresponding members of the aggregate variable. When a union is initialized, *initializer-list* must be a single constant expression. The value of the constant expression is assigned to the first member of the union.

If *initializer-list* has fewer values than an aggregate type, the remaining members or elements of the aggregate type are initialized to 0. If *initializer-list* has more values than an aggregate type, an error results. These rules apply to each embedded initializer list, as well as to the aggregate as a whole.

The following example, declares *P* as a 4-by-3 array and initializes the elements of its first row to 1, the elements of its second row to 2, and so on through the fourth row:

```

int P[4][3] = {
    { 1, 1, 1 },
    { 2, 2, 2 },
    { 3, 3, 3 },
    { 4, 4, 4 },
};

```

C Language Reference

Note that the initializer list for the third and fourth rows contains commas after the last constant expression. The last initializer list (`{4, 4, 4,}`) is also followed by a comma. These extra commas are permitted but not required; only commas that separate constant expressions from one another, and those that separate one initializer list from another, are required.

If there is no embedded initializer list for an aggregate member, values are simply assigned, in order, to each member of the subaggregate. Therefore, the initialization in the previous example is equivalent to the following:

```
int P[4][3] = {
    1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4
};
```

Braces can also appear around individual initializers in the list.

When you initialize an aggregate variable, you must be careful to use braces and initializer lists properly. The following example illustrates the compiler's interpretation of braces in more detail:

```
typedef struct {
    int n1, n2, n3;
} triplet;

triplet nlist[2][3] = {
    { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } }, /* Line 1 */
    { { 10,11,12 }, { 13,14,15 }, { 16,17,18 } } /* Line 2 */
};
```

In this example, *nlist* is declared as a 2-by-3 array of structures, each structure having three members. Line 1 of the initialization assigns values to the first row of *nlist*, as follows:

1. The first left brace on Line 1 signals the compiler that the first aggregate member of *nlist* (that is, *nlist[0]*) is initializing.
2. The second left brace indicates that the first aggregate member of *nlist[0]* (that is, the structure at *nlist[0][0]*) is initializing.
3. The first right brace ends initialization of the structure *nlist[0][0]*; the next left brace starts initializing *nlist[0][1]*.
4. The process continues until the end of the line, where the closing right brace ends initialization of *nlist[0]*.

Line 2 assigns values to the second row of *nlist* in a similar way.

Note that the outer sets of braces enclosing the initializers on lines 1 and 2 are required. The following construction, which omits the outer braces, would cause an error:

```
triplet nlist[2][3] = {
    { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, /* Line 1 */
    { 10,11,12 }, { 13,14,15 }, { 16,17,18 } /* Line 2 */
};
```

In this construction, the first left brace on line 1 initializes *nlist[0]*, which is an array of three structures. The values 1, 2, and 3 are assigned to the three members of the first structure. When the next right brace is encountered (after the value 3), initialization of *nlist[0]* is complete, and the two remaining structures in the three-structure array are automatically initialized to 0. Similarly, { 4,5,6 } initializes the first structure in the second row of *nlist*. The remaining two structures of *nlist[1]* are set to 0. When the compiler encounters the next initializer list ({ 7,8,9 }), it tries to initialize *nlist[2]*. Since *nlist* has only two rows, this attempt causes an error.

4

Example 1

In this example, the three **int** members of *x* are initialized to 1, 2, and 3, respectively. The three elements in the first row of *m* are initialized to 4.0; the elements of the remaining row of *m* are initialized to 0.0 by default.

```
struct list {
    int i, j, k;
    float m[2][3];
} x = {
    1,
    2,
    3,
    {4.0, 4.0, 4.0}
};
```

Example 2

In this example, the union variable *y* is initialized. The first element of the union is an array, so the initializer is an aggregate initializer. The initializer list {'I'} assigns values to the first row of the array. Since only one value appears in the list, the element in the first column is initialized

C Language Reference

to the character *I*, and the remaining two elements in the row are initialized to the value zero by default. Similarly, the first element of the second row of *x* is initialized to the character *4*, and the remaining two elements in the row are initialized to the value 0.

```
union
{
    char x[2][3];
    int i, j, k;
} y = { {
        {'1'},
        {'4'} }
};
```

4.7.3 String Initializers

Syntax

```
= "characters"
```

You can initialize an array of characters with a string literal. The following example, initializes *code* as a four-element array of characters. The fourth element is the null character, which terminates all string literals.

```
char code[ ] = "abc";
```

If you specify the array size and the string is longer than the specified array size, the extra characters are simply ignored. For example, the following declaration initializes *code* as a three-element character array:

```
char code[3] = "abcd";
```

Only the first three characters of the initializer are assigned to *code*. The character *d* and the string-terminating null character are discarded. Beware that this creates an unterminated string (that is, one without a 0 value to mark its end) and generates a diagnostic message indicating the condition.

If the string is shorter than the specified array size, the remaining elements of the array are initialized to 0 values.

4.8 Type Declarations

A type declaration defines the name and members of a structure or union type, or the name and enumeration set of an enumeration type. You can use the name of a declared type in variable or function declarations to refer to that type. This is useful if many variables and functions have the same type.

A **typedef** declaration defines a type specifier for a type. You can use **typedef** declarations to construct shorter or more meaningful names for types already defined by C or for types that you have declared.

4.8.1 Structure, Union, and Enumeration Types

Declarations of structure, union, and enumeration types have the same general form as variable declarations of those types. However, type declarations and variable declarations differ in the following ways:

- In type declarations the variable identifier is omitted, since no variable is declared.
- In type declarations *tag* is required; it names the structure, union, or enumeration type.
- The *member-declaration-list* or *enum-list* defining the type must appear in the type declaration; the abbreviated form of variable declarations, in which *tag* refers to a type defined elsewhere, is not legal for type declarations.

Example 1

This example declares an enumeration type named *status*. The name of the type can be used in declarations of enumeration variables. The identifier *loss* is explicitly set to -1. Both *bye* and *tie* are associated with the value 0, and *win* is given the value 1.

```
enum status {
    loss = -1,
    bye,
    tie = 0,
    win
};
```

C Language Reference

Example 2

This example declares a structure type named *student*. A declaration such as *struct student employee;* can be used to define a structure variable with *student* type.

```
struct student {
    char name[20];
    int id, class;
};
```

4.8.2 Using typedef Declarations

Syntax

typedef *type-specifier declarator* [, *declarator*]....

A **typedef** declaration is analogous to a variable declaration except that the **typedef** keyword replaces a storage-class specifier. A **typedef** declaration is interpreted in the same way as a variable or function declaration, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type.

Note that a **typedef** declaration does not create types. It creates synonyms for existing types, or names for types that could be specified in other ways. When a **typedef** name is used as a type specifier, it can be combined with certain type specifiers, but not others. Acceptable modifiers include **const** and **volatile**. In some implementations there are additional special keywords that can be used to modify a **typedef**.

You can declare any type with **typedef**, including pointer, function, and array types. You can declare a **typedef** name for a pointer to a structure or union type before you define the structure or union type, as long as the definition has the same visibility as the declaration.

Example 1

This example declares *WHOLE* to be a synonym for **int**. Note that *WHOLE* could now be used in a variable declaration such as *WHOLE i;* or *const WHOLE i;*. However, the declaration *long WHOLE i;* would be illegal.

```
typedef int WHOLE;
```

Example 2

This example declares *GROUP* as a structure type with three members. Since a structure tag, *club*, is also specified, either the **typedef** name (*GROUP*) or the structure tag can be used in declarations.

```
typedef struct club {
    char name[30];
    int size, year;
} GROUP ;
```

Example 3

This example uses the previous **typedef** name to declare a pointer type. The type *PG* is declared as a pointer to the *GROUP* type, which in turn is defined as a structure type.

```
typedef GROUP *PG;
```

4

Example 4

Example 4 provides the type *DRAWF* for a function returning no value and taking two **int** arguments. This means, for example, that the declaration *DRAWF box*; is equivalent to the declaration *void box(int, int)*;

```
typedef void DRAWF(int, int);
```

4.9 Type Names

A “type name” specifies a particular data type. In addition to ordinary variable declarations and defined-type declarations, type names are used in three other contexts: in the formal-parameter lists of function declarations, in type casts, and in **sizeof** operations. Formal-parameter lists are discussed in “Function Declarations.” Type casts and **sizeof** operations are discussed in Sections 5.6.2 and 5.3.4, respectively.

The type names for fundamental, enumeration, structure, and union types are simply the type specifiers for those types.

C Language Reference

A type name for a pointer, array, or function type has the following form:

type-specifier abstract-declarator

An *abstract-declarator* is a declarator without an identifier, consisting of one or more pointer, array, or function modifiers. The pointer modifier (*) always precedes the identifier in a declarator; array ([]) and function (()) modifiers follow the identifier. Knowing this, you can determine where the identifier would appear in an abstract declarator and interpret the declarator accordingly. For more information and examples of complex declarators, see Section 4.3.2.

Abstract declarators can be complex. Parentheses in a complex abstract declarator specify a particular interpretation, just as they do for the complex declarators in declarations.

Note

The abstract declarator consisting of a set of empty parentheses, (), is not allowed because it is ambiguous. It is impossible to determine whether the implied identifier belongs inside the parentheses (in which case it is an unmodified type) or before the parentheses (in which case it is a function type).

The type specifiers established by **typedef** declarations also qualify as type names.

Example 1

This example gives the type name for “pointer to **long**” type.

```
long *
```

Example 2

Examples 2 and 3 show how parentheses modify complex abstract declarators. Example 2 gives the type name for a pointer to an array of five **int** values.

```
int (*) [5]
```

Example 3

Example 3 specifies a pointer to a function taking no arguments and returning an `int` value.

```
int (*)(void)
```


Chapter 5

Expressions and Assignments

- 5.1 Introduction 5-1
- 5.2 C Operands 5-1
 - 5.2.1 Constants 5-2
 - 5.2.2 Identifiers 5-2
 - 5.2.3 Strings 5-3
 - 5.2.4 Function Calls 5-3
 - 5.2.5 Subscript Expressions 5-4
 - 5.2.6 Member-Selection Expressions 5-7
 - 5.2.7 Expressions with Operators 5-8
 - 5.2.8 Expressions in Parentheses 5-9
 - 5.2.9 Type-Cast Expressions 5-10
 - 5.2.10 Constant Expressions 5-10
 - 5.2.11 Side Effects 5-11
 - 5.2.12 Sequence Points 5-12
- 5.3 C Operators 5-13
 - 5.3.1 Usual Arithmetic Conversions 5-13
 - 5.3.2 Complement and Unary Plus Operators 5-15
 - 5.3.3 Indirection and Address-of Operators 5-16
 - 5.3.4 The sizeof Operator 5-18
 - 5.3.5 Multiplicative Operators 5-19
 - 5.3.6 Additive Operators 5-21
 - 5.3.7 Shift Operators 5-23
 - 5.3.8 Relational Operators 5-24
 - 5.3.9 Bitwise Operators 5-26
 - 5.3.10 Logical Operators 5-27
 - 5.3.11 Sequential-Evaluation Operator 5-28
 - 5.3.12 Conditional Operator 5-29
- 5.4 Assignment Operators 5-30
 - 5.4.1 Lvalue Expressions 5-31
 - 5.4.2 Unary Increment and Decrement 5-32
 - 5.4.3 Simple Assignment 5-33
 - 5.4.4 Compound Assignment 5-34

5.5	Precedence and Order of Evaluation	5-35
5.6	Type Conversions	5-38
5.6.1	Assignment Conversions	5-38
5.6.2	Type-Cast Conversions	5-46
5.6.3	Operator Conversions	5-46
5.6.4	Function-Call Conversions	5-47

5.1 Introduction

This chapter describes how to form expressions and make assignments in the C language. An “expression” is a combination of operands and operators that yields (“expresses”) a single value.

An “operand” is a constant or variable value that is manipulated in the expression. Each operand of an expression is also an expression, since it represents a single value. When an expression is evaluated, the resulting value depends on the relative precedence of operators in the expression and on “sequence points” and “side effects,” if any. The precedence of operators determines how operands are grouped for evaluation. Side effects are changes caused by the evaluation of an expression. In an expression with side effects, the evaluation of one operand can affect the value of another. With some operators, the order in which operands are evaluated also affects the result of the expression. Section 5.2 describes the formats and evaluation rules for C operands, including discussions of side effects and sequence points.

“Operators” specify how the operand or operands of the expression are manipulated. C operators are described in Section 5.3.

In C, assignments are considered expressions because an assignment yields a value. Its value is the value being assigned. In addition to the simple-assignment operator (=), C offers complex-assignment operators that both transform and assign their operands. Assignment operators are described in Section 5.4.

5

The value represented by each operand in an expression has a type that may be converted to a different type in certain contexts. Type conversions occur in assignments, type casts, function calls, and operations. (Section 5.5 gives the precedence rules for C operators; side effects are discussed in Section 5.2.11 and type conversions in Section 5.6.)

5.2 C Operands

Operands in C include constants, identifiers, strings, function calls, subscript expressions, member-selection expressions, or more complex expressions formed by combining operands with operators or enclosing operands in parentheses. Any operand that yields a constant value is called a “constant expression.”

Every operand has a type. The following sections discuss the type of value each kind of operand represents. An operand can be “cast” (or temporarily converted) from its original type to another type by means of a “type-cast” operation. A type-cast expression can also form an operand of an expression.

C Language Reference

5.2.1 Constants

A constant operand has the value and type of the constant value it represents. A character constant has **int** type. An integer constant has **int**, **long**, **unsigned int**, or **unsigned long** type, depending on the integer's size and how the value is specified. Floating-point constants always have **double** type. String literals are considered arrays of characters and are discussed in Section 5.2.3.

5.2.2 Identifiers

An “identifier” names a variable or function. Every identifier has a type that is established when the identifier is declared. The value of an identifier depends on its type, as follows:

- Identifiers of integral and floating types represent values of the corresponding type.
- An identifier of **enum** type represents one constant value among a set of constant values. The value of the identifier is the constant value. Its type is **int**, by definition of the **enum** type.
- An identifier of **struct** or **union** type represents a value of the specified **struct** or **union** type.
- An identifier declared as a pointer represents a pointer to a value of the type specified in the pointer's declaration.
- An identifier declared as an array represents a pointer whose value is the address of the first array element. The pointer addresses the type of the array elements. For if *series* is declared to be a 10-element integer array, the identifier *series* represents the address of the array, and the subscript expression *series[5]* refers to an integer value which is the sixth element of *series*. Subscript expressions are discussed in Section 5.2.5. The address of an array does not change during program execution, although the values of the individual elements can change. The pointer value represented by an array identifier is not a variable, so an array identifier cannot form the left-hand operand of an assignment operation.
- An identifier declared as a function represents a pointer whose value is the address of the function. The pointer addresses a function returning a value of a specified type. The address of a function does not change during program execution; only the return value varies. Thus, function identifiers cannot be left-hand operands in assignment operations.

5.2.3 Strings

Syntax

```
"string" ["string"]
```

A “string literal” is a character or sequence of adjacent characters enclosed in double quotation marks. Two or more adjacent string literals separated only by white space are concatenated into a single string literal. A string literal is stored as an array of elements with **char** type and initialized with the quoted sequence of characters. The string literal is represented by a pointer whose value is the address of the first array element. The address of the string’s first element is a constant, so the value represented by a string expression is a constant.

Since string literals are effectively pointers, they can be used in the same contexts as pointers, and have the same restrictions as pointers. However, since they are not variables, neither string literals nor any of their elements can be the left-hand operand in an assignment operation.

The last character of a string is always the null character. Though the null character is not visible in the string expression, it is added automatically as the last element when the string is stored. For example, the string “abc” actually has four characters rather than three.

5

5.2.4 Function Calls

Syntax

```
expression ([expression-list])
```

A “function call” consists of an *expression* followed by an optional *expression-list* in parentheses, where

- *expression* must evaluate to a function address (for example, a function identifier), and
- *expression-list* is a list of expressions (separated by commas) whose values (the “actual arguments”) are passed to the function. The *expression-list* argument can be empty.

A function-call expression has the value and type of the function’s return value. If the function’s return type is **void** (that is, the function has been declared never to return a value), the function-call expression also has **void** type. If the called function returns control without executing a **return** statement, the value of the function-call expression is undefined.

C Language Reference

(See the chapter on “Functions,” for more information about function calls.)

5.2.5 Subscript Expressions

Syntax

expression1 [*expression2*]

A subscript expression represents the value at the address that is *expression2* positions beyond *expression1*. Usually, the value represented by *expression1* is a pointer value, such as an array identifier, and *expression2* is an integral value. However, all that is required syntactically is that one of the expressions be of pointer type and the other be of integral type. Thus the integral value could be in the *expression1* position and the pointer value could be in the brackets in the *expression2*, or “subscript,” position. Whatever the order of values, *expression2* must be enclosed in brackets ([]).

Subscript expressions are generally used to refer to array elements, but you can apply a subscript to any pointer.

Unidimensional-Array References

The subscript expression is evaluated by adding the integral value to the pointer value, then applying the indirection operator (*) to the result. (See Section 5.3.3 for a discussion of the indirection operator.) In effect, for a one-dimensional array, the following four expressions are equivalent, assuming that *a* is a pointer and *b* is an integer:

```
a[b]
*(a + b)
*(b + a)
b[a]
```

According to the conversion rules for the addition operator (given in Section 5.3.6), the integral value is converted to an address offset by multiplying it by the length of the type addressed by the pointer.

For example, suppose the identifier *line* refers to an array of **int** values. The following procedure is used to evaluate the subscript expression *line*[*i*]:

1. The integer value *i* is multiplied by the number of bytes defined as the length of an **int** item. The converted value of *i* represents *i* **int** positions.

2. This converted value is added to the original pointer value (*line*) to yield an address that is offset *i* `int` positions from *line*.
3. The indirection operator is applied to the new address. The result is the value of the array element at that position (intuitively, *line*[*i*]).

Note

The following subscript expression represents the value of the first element of *line*, since the offset from the address represented by *line* is 0:

```
line[0]
```

Similarly, an expression such as the following refers to the element offset five positions from *line* or the sixth element of the array:

```
line[5]
```

Multidimensional-Array Reference

A subscript expression can be subscripted, as follows:

```
expression1 [expression2] [expression3]...
```

Subscript expressions associate from left to right. The left-most subscript expression, *expression1*[*expression2*], is evaluated first. The address that results from adding *expression1* and *expression2* forms a pointer expression; then *expression3* is added to this pointer expression to form a new pointer expression, and so on until the last subscript expression has been added. The indirection operator (*) is applied after the last subscripted expression is evaluated, unless the final pointer value addresses an array type (see Example 3).

Expressions with multiple subscripts refer to elements of “multidimensional arrays.” A multidimensional array is an array whose elements are arrays. For example, the first element of a three-dimensional array is an array with two dimensions.

C Language Reference

For the following examples, an array named *prop* is declared with three elements, each of which is a 4-by-6 array of **int** values.

```
int prop[3][4][6];
int i, *ip, (*ipp)[6];
```

Example 1

This example shows how to refer to the second individual **int** element of *prop*. Arrays are stored by row, so the last subscript varies the most quickly; the expression *prop*[0][0][2] refers to the next (third) element of the array, and so on.

```
i = prop[0][0][1];
```

Example 2

This example shows a more complex reference to an individual element of *prop*. The expression is evaluated as follows:

1. The first subscript, 2, is multiplied by the size of a 4-by-6 **int** array and added to the pointer value *prop*. The result points to the third 4-by-6 array of *prop*.
2. The second subscript, 1, is multiplied by the size of the 6-element **int** array and added to the address represented by *prop*[2].
3. Each element of the 6-element array is an **int** value, so the final subscript, 3, is multiplied by the size of an **int** before it is added to *prop*[2][1]. The resulting pointer addresses the fourth element of the 6-element array.
4. The indirection operator is applied to the pointer value. The result is the **int** element at that address.

```
i = prop[2][1][3];
```

Example 3

Examples 3 and 4 show cases where the indirection operator is not applied.

In Example 3, the expression *prop*[2][1] is a valid reference to the three-dimensional array *prop*; it refers to a 6-element array (declared above

Example 1). Since the pointer value addresses an array, the indirection operator is not applied.

```
ip = prop[2][1];
```

Example 4

As in example 3, the result of the expression *prop[2]* in Example 4 is a pointer value addressing a two-dimensional array.

```
ipp = prop[2];
```

5.2.6 Member-Selection Expressions

Syntax

expression.identifier
expression->identifier

A “member-selection expression” refers to members of structures and unions. Such an expression has the value and type of the selected member. As shown in the syntax line, a member-selection expression can have one of the two following forms:

1. In the first form, *expression.identifier*, *expression* represents a value of **struct** or **union** type, and *identifier* names a member of the specified structure or union.
2. In the second form, *expression->identifier*, *expression* represents a pointer to a structure or union, and *identifier* names a member of the specified structure or union.

The two forms of member-selection expressions have similar effects. In fact, an expression involving the pointer selection operator (\rightarrow) is a shorthand version of an expression using the period (.) if the expression before the period consists of the indirection operator (*) applied to a pointer value. (Section 5.3.3 discusses the indirection operator.) Therefore,

expression->identifier



C Language Reference

is equivalent to

*(*expression).identifier*

when *expression* is a pointer value.

Examples 1 through 3 refer to the following structure declaration:

```
struct pair {
    int a;
    int b;
    struct pair *sp;
} item, list[10];
```

Example 1

In this example, the address of the *item* structure is assigned to the *sp* member of the structure. This means that *item* contains a pointer to itself.

```
item.sp = &item;
```

Example 2

In this example, the pointer expression *item.sp* is used with the pointer selection operator (\rightarrow) to assign a value to the member *a*.

```
(item.sp)→a = 24;
```

Example 3

This example shows how to select an individual structure member from an array of structures.

```
list[8].b = 12;
```

5.2.7 Expressions with Operators

Expressions with operators can be “unary,” “binary,” or “ternary” expressions. A unary expression consists of either a unary operator (“unop”) prepended to an operand, or the **sizeof** keyword followed by an *expression*. The *expression* can be either the name of a variable or a cast expression. If *expression* is a cast expression it must be enclosed in parentheses.

unop operand
sizeof *expression*

A binary expression consists of two operands joined by a binary operator (“binop”):

operand binop operand

A ternary expression consists of three operands joined by the ternary operator (?:):

operand ? operand : operand

Sections 5.3.1-5.3.12, describe the operators used in unary, binary, and ternary expressions.

Expressions with operators also include assignment expressions that use unary or binary assignment operators. The unary assignment operators are the increment (++) and decrement (--) operators; the binary assignment operators are the simple-assignment operator (=) and the compound-assignment operators (referred to as “compound-assign-ops”). Each compound-assignment operator is a combination of another binary operator with the simple-assignment operator. Assignment expressions have the following forms:

operand++
operand--
++operand
--operand
operand = operand
operand compound-assign-op operand

5

Sections 5.4.1 - 5.4.4 describe the assignment operators in detail.

5.2.8 Expressions in Parentheses

You can enclose any operand in parentheses without changing the type or value of the enclosed expression. For example, in the following expression, the parentheses around $10 + 5$ mean that the value of $10 + 5$ is the left operand of the division (/) operator.

$(10 + 5) / 5$

The result of $(10 + 5) / 5$ is 3. Without the parentheses, $10 + 5 / 5$ would evaluate to 11.

Although parentheses affect the way operands are grouped in an expression, they cannot guarantee a particular order of evaluation in all cases. Exceptions resulting from “side effects” are discussed in Section 5.2.11.

C Language Reference

5.2.9 Type-Cast Expressions

A type cast provides a method for explicit conversion of the type of an object in a specific situation. Type-cast expressions have the following form:

(type-name) operand

Casts can be used to convert objects of any scalar type to or from any other scalar type. Explicit type casts are constrained by the same rules that determine the effects of implicit conversions, discussed in “Assignment Conversions.” Additional restraints on casts may result from the actual sizes or representation of specific types on specific implementations. Representation is discussed in the “Declarations” chapter. For information on actual sizes of integral types and pointers, see your compiler guide.

Any object may be cast to **void** type. However, if the *type-name* in a type-cast expression is not **void**, then *operand* cannot be a **void** expression. Any expression can be cast to **void**, but an expression of type **void** cannot be cast to any other type. For example, a function with **void** return type cannot have its return cast to another type. Note that a **void *** expression has a type pointer to **void**, not type **void**. If an object is cast to **void** type, the resulting expression cannot be assigned to any item. Similarly, a type-cast object is not an acceptable lvalue, so no assignment can be made to a type-cast object. Section 4.9 discusses type names. Section 5.4.1 discusses Lvalues. Section 5.6 discusses type-cast conversions.

5.2.10 Constant Expressions

A constant expression is any expression that evaluates to a constant. The operands of a constant expression can be integer constants, character constants, floating type constants, enumeration constants, type casts, **sizeof** expressions, and other constant expressions. You can use operators to combine and modify operands as described in Section 5.2.7, with the following restrictions:

- You cannot use assignment operators (see Section 5.4) or the binary sequential-evaluation operator (**,**) in constant expressions.
- You can use the unary address-of operator (**&**) only in certain initializations (as described in the last paragraph of this section).

Constant expressions used in preprocessor directives are subject to additional restrictions. Consequently, they are known as “restricted constant expressions.” A restricted constant expression cannot contain **sizeof**

expressions, enumeration constants, type casts to any type, or floating-type constants. It can, however, contain the special constant expression **defined**(*identifier*). (For more information about this expression, see Section 8.2.2, “The #define Directive.”)

Constant expressions involving floating constants, casts to nonarithmetic types, and address-of expressions can only appear in initializers. The unary address-of operator (&) can only be applied to variables with fundamental, structure, or union types that are declared at the global level, or to subscripted array references. In these expressions, a constant expression that does not include the address-of operator can be added to or subtracted from the address expression.

5.2.11 Side Effects

“Side effects” occur whenever the value of a variable is changed by expression evaluation. All assignment operations have side effects. Function calls may also have side effects if they change the value of an externally visible item, either by direct assignment or by indirect assignment through a pointer.

The order of evaluation of expressions is defined by the specific implementation, except when the language guarantees a particular order of evaluation (as outlined in Section 5.5).



For example, side effects occur in the following function call:

```
add (i + 1, i = j + 2)
```

The arguments of a function call can be evaluated in any order. The expression $i + 1$ can be evaluated before $i = j + 2$, or $i = j + 2$ can be evaluated before $i + 1$. The result is different in each case.

Since unary increment and decrement operations involve assignments, such operations can cause side effects, as shown in the following example:

```
d = 0;
a = b++ = c++ = d++;
```

In this example, the value of a is unpredictable. The value of d (initially 0) could be assigned to c , then to b , and then to a before any of the variables are incremented. In this case, a would be equal to 0.

A second way to evaluate this expression begins by evaluating the operand $c++ = d++$. The value of d (initially 0) is assigned to c , and then

C Language Reference

both *d* and *c* are incremented. Next, the value of *c*, now 1, is assigned to *b*, and *b* is incremented. Finally, the incremented value of *b* is assigned to *a*; in this case, the final value of *a* is 2.

Since C does not define the order of evaluation of side effects, both evaluation methods discussed above are correct and either may be implemented. To make sure that your code is portable and clear, avoid statements that depend on a particular order of evaluation for side effects.

5.2.12 Sequence Points

Expressions involving assignment, unary “increment,” unary “decrement,” or calling a function may have consequences incidental to their evaluation (side effects). When a “sequence point” is reached, everything preceding the sequence point, including any side effects, is guaranteed to have been evaluated before evaluation begins on anything following the sequence point.

Certain operators act as sequence points, including the following:

- The logical-AND operator (&&)
- The logical-OR operator (||)
- The ternary operator (?:)
- The sequential-evaluation operator (,)
- The function-call operator (that is, the parentheses following a function name)

Other sequence points include

- the end of a full expression (that is, an expression that is not part of another expression)
- any initializer
- an expression in an expression statement
- the control expressions in selection statements (**if** or **switch**) and iteration statements (**do**, **while**, or **for**)
- the expression in a **return** statement

5.3 C Operators

C operators take one operand (unary operators), two operands (binary operators), or three operands (the ternary operator). Assignment operators include both unary or binary operators; Section 5.4 describes the assignment operators.

Unary operators appear before their operand and associate from right to left. C includes the following unary operators:

- ~ !	Negation and complement operators
* &	Indirection and address-of operators
sizeof	Size operator
+	Unary plus operator

Binary operators associate from left to right. C provides the following binary operators:

* / %	Multiplicative operators
+ -	Additive operators
<< >>	Shift operators
< > <= >= == !=	Relational operators
& ^	Bitwise operators
&&	Logical operators
,	Sequential-evaluation operator

C has one ternary operator: the conditional operator (? :). It associates from right to left.

5.3.1 Usual Arithmetic Conversions

Most C operators perform type conversions to bring the operands of an expression to a common type or to extend short values to the integer size used in machine operations. The conversions performed by C operators depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integral and floating types. These conversions are known as “arithmetic

C Language Reference

conversions” because they apply to the types of values ordinarily used in arithmetic.

The arithmetic conversions summarized in this section are called “usual arithmetic conversions.” The discussion of each operator in the following sections specifies whether or not the operator performs the usual arithmetic conversions. It also specifies the additional conversions, if any, the operator performs. This is not a precedence order. It is an outline of an algorithm that is applied to each binary operator in the expression.

Section 5.6 outlines the specific path of each type of conversion. In determining which conversions will actually take place, the following algorithm is applied to each binary operation in the expression:

1. Any operands of **float** type are converted to **double** type.
2. If one operand has **long double** type, the other operand is converted to **long double** type.
3. If one operand has **double** type, the other operand is converted to **double** type.
4. Any operands of **char** or **short** type are converted to **int** type.
5. Any operands of **unsigned char** or **unsigned short** type are converted to **unsigned int** type.
6. If one operand is of **unsigned long** type, the other operand is converted to **unsigned long** type.
7. If one operand is of **long** type, the other operand is converted to **long** type.
8. If one operand is of **unsigned int** type, the other operand is converted to **unsigned int** type.

The following example illustrates the application of the preceding algorithm:

```
long l;  
unsigned char uc;  
int i;  
f( l + uc * i);
```

The preceding example would be converted as follows:

1. *uc* is converted to an **unsigned int** (step 5).
2. *i* is converted to an **unsigned int** (step 8). The multiplication is performed and the result is an **unsigned int**.
3. *uc * i* is converted to a **long** (step 7).

The addition is performed and the result is type **long**.

5.3.2 Complement and Unary Plus Operators

The C complement operators are discussed in the following list:

- The arithmetic-negation operator produces the negative (two's complement) of its operand. The operand must be an integral or floating value. This operator performs the usual arithmetic conversions.
- The bitwise-complement operator produces the bitwise complement of its operand. The operand must be of integral type. This operator performs usual arithmetic conversions; the result has the type of the operand after conversion.
- ! The logical-NOT operator produces the value 0 if its operand is true (nonzero) and the value 1 if its operand is false (0). The result has **int** type. The operand must be an integral, floating, or pointer value.
- + The unary plus operator preceding a parenthesized expression forces the grouping of the enclosed operations. It is used with expressions involving more than one associative or commutative binary operator.

5

Note

The unary plus operator (+) is implemented syntactically in Microsoft C, but has no semantics of any type associated with it.

C Language Reference

Example 1

In this example, the new value of x is the negative of 987, or -987 .

```
short x = 987;  
x = -x;
```

Example 2

In this example, the new value assigned to y is the one's complement of the unsigned value 0xaaaa, or 0x5555.

```
unsigned short y = 0xaaaa;  
y = ~y;
```

Example 3

In this example, if x is greater than or equal to y , the result of the expression is 1 (true). If x is less than y , the result is 0 (false).

```
if ( !(x < y) );
```

5.3.3 Indirection and Address-of Operators

The C indirection and address-of operators are discussed in the following list:

- The indirection operator accesses a value indirectly, through a pointer. The operand must be a pointer value. The result of the operation is the value addressed by the operand; that is, the value at the address specified by the operand. The type of the result is the type that the operand addresses. If the pointer value is invalid, the result is undefined. The specific conditions that invalidate a pointer value are implementation-defined. The following list includes some of the most common:
 - The pointer is a null pointer.
 - The pointer specifies the address of a local item that is not active at the time of the reference.
 - The pointer specifies an address that is inappropriately aligned for the type of the object pointed to.

- The pointer specifies an address not used by the executing program.
- The address-of operator gives the address of its operand. The operand can be any value that is a valid left-hand value of an assignment operation. A function designator or array name can also be the operand of the address-of operator, although in these cases the operator is superfluous since function designators and array names are addresses. (Assignment operations are discussed in Section 5.4.) The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.

You cannot apply the address-of operator to a bitfield member of a structure (described in Section 4.4.3, “Structure Declarations”) or to an identifier declared with the **register** storage-class specifier (described in Section 4.6).

Examples 1 through 4 use the following declarations:

```
int *pa, x;  
int a[20];  
double d;
```

5

Example 1

In this example, the address-of operator (&) takes the address of the sixth element of the array *a*. The result is stored in the pointer variable *pa*.

```
pa = &a[5];
```

Example 2

In this example the indirection operator (*) is used to access the **int** value at the address stored in *pa*. The value is assigned to the integer variable *x*.

```
x = *pa;
```

Example 3

In this example, the word *True* would be printed. This example demonstrates that the result of applying the indirection operator to the address of *x* is the same as *x*.

```
if (x == *x)  
    printf("True\n");
```

C Language Reference

Example 4

This example demonstrates an appropriate application of the rule shown in Example 3. First the address of *x* is converted by a type cast to a pointer to a **double** type; then the indirection operator is applied to give a result of type **double**.

```
d = *(double *)(&x);
```

Example 5

In this example, the function *roundup* is declared, and then two pointers to *roundup* are declared and initialized. The first pointer *proundup* is initialized using only the name of the function, while the second, *pround*, uses the address-of operator in the initialization. The initializations are equal.

```
int roundup() ;  
  
int (*proundup) = roundup;  
int (*pround) = &roundup;
```

5.3.4 The sizeof Operator

The **sizeof** operator gives the amount of storage, in bytes, associated with an identifier or a type. This operator lets you avoid specifying machine-dependent data sizes in your programs.

A **sizeof** expression has the form

sizeof *expression*

An *expression* is either an identifier or a type-cast expression (that is, a type specifier enclosed in parentheses). If *expression* is a type-cast expression, it cannot be **void**. If it is an identifier, it cannot represent a bitfield object or a function designator.

When you apply the **sizeof** operator to an array identifier, the result is the size of the entire array rather than the size of the pointer represented by the array identifier.

When you apply the **sizeof** operator to a structure or union type name, or to an identifier of structure or union type, the result is the actual size of the structure or union. This size may include internal and trailing padding used to align the members of the structure or union on memory

boundaries. Thus, the result may not correspond to the size calculated by adding up the storage requirements of the individual members.

Example 1

This example uses the **sizeof** operator to pass the size of an **int**, which varies among machines, as an argument to a function named *calloc*. The *buffer* stores the value returned by the function.

```
buffer = calloc(100, sizeof (int) );
```

Example 2

In this example, *strings* is an array of pointers to **char**. The number of pointers is the number of elements in the array, but is not specified. It is easy to determine the number of pointers by using the **sizeof** operator to calculate the number of elements in the array. The **const** integer value *string_no* is initialized to this number. Because it is a **const** value, *string_no* cannot be modified.

```
static char *strings[] ={
    "this is string one",
    "this is string two",
    "this is string three",
};
const int string_no = (sizeof strings)/(sizeof strings[0]);
```

5

5.3.5 Multiplicative Operators

The multiplicative operators perform multiplication (*), division (/), and remainder (%) operations. The operands of the remainder operator (%) must be integral. The multiplication (*) and division (/) operators can take integral- or floating-type operands; the types of the operands can be different.

The multiplicative operators perform the usual arithmetic conversions on the operands. The type of the result is the type of the operands after conversion.

Note

Since the conversions performed by the multiplicative operators do not provide for overflow or underflow conditions, information may be lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

The C multiplicative operators are described as follows:

* The multiplication operator causes its two operands to be multiplied.

/ The division operator causes the first operand to be divided by the second. If two integer operands are divided and the result is not an integer, it is truncated according to the following rules:

- If both operands are positive or unsigned, the result is truncated toward 0.
- If either operand is negative, the direction of truncation of the result (either toward 0 or away from 0) is defined by the implementation. For more information, see your compiler guide.

The result of division by 0 is undefined.

% The result of the remainder operator is the remainder when the first operand is divided by the second. If either or both operands are positive or unsigned, the result is positive. If either operand is negative the sign of the result is defined by the implementation. (For more information, see your compiler guide.) If the right operand is zero, the result is undefined.

These declarations are used for all of the following examples:

```
int i = 10, j = 3, n;  
double x = 2.0, y;
```

Example 1

In this example, x is multiplied by i to give the value 20.0. The result has **double** type.

```
y = x * i;
```

Example 2

In this example, 10 is divided by 3. The result is truncated toward 0, yielding the integer value 3.

```
n = i / j;
```

Example 3

In this example, n is assigned the integer remainder, 1, when 10 is divided by 3.

```
n = i % j;
```

5.3.6 Additive Operators

5

The additive operators perform addition (+) and subtraction (-). The operands can be integral or floating values. Some additive operations can also be performed on pointer values, as outlined under the discussion of each operator.

The additive operators perform the usual arithmetic conversions on integral and floating operands. The type of the result is the type of the operands after conversion. Since the conversions performed by the additive operators do not provide for overflow or underflow conditions, information may be lost if the result of an additive operation cannot be represented in the type of the operands after conversion.

Addition (+)

The addition operator (+) causes its two operands to be added. Both operands can have integral or floating types, or one operand can be a pointer and the other an integer.

When an integer is added to a pointer, the integer value (i) is converted by multiplying it by the size of the value that the pointer addresses. After conversion, the integer value represents i memory positions, where each position has the length specified by the pointer type. When the converted

C Language Reference

integer value is added to the pointer value, the result is a new pointer value representing the address i positions from the original address. The new pointer value addresses a value of the same type as the original pointer value.

Subtraction (-)

The subtraction operator (-) subtracts the second operand from the first. The following combinations of operands can be used with this operator:

- Both operands integral or floating type values
- Both operands pointer values to the same type
- The first operand a pointer value and the second operand an integer

When two pointers are subtracted, the difference is converted to a signed integral value by dividing the difference by the size of a value of the type that the pointers address. The result represents the number of memory positions of that type between the two addresses. The result is only guaranteed to be meaningful for two elements of the same array, as discussed in “Pointer Arithmetic,” later in this section.

When an integer value is subtracted from a pointer value, the subtraction operator converts the integer value (i) by multiplying it by the size of the value that the pointer addresses. After conversion, the integer value represents i memory positions, where each position has the length specified by the pointer type. When the converted integer value is subtracted from the pointer value, the result is the memory address i positions before the original address. The new pointer points to a value of the type addressed by the original pointer value.

Pointer Arithmetic

Additive operations involving a pointer and an integer give meaningful results only if the pointer operand addresses an array member and the integer value produces an offset within the bounds of the same array. When the integer value is converted to an address offset, the compiler assumes that only memory positions of the same size lie between the original address and the address plus the offset.

This assumption is valid for array members. By definition, an array is a series of values of the same type; its elements reside in contiguous memory locations. However, storage for any types except array elements is not guaranteed to be completely filled. That is, blanks may appear between memory positions, even positions of the same type. Therefore,

the results of adding to or subtracting from the addresses of any values but array elements are undefined.

Similarly, when two pointer values are subtracted, the conversion assumes that only values of the same type, with no blanks, lie between the addresses given by the operands.

On machines with segmented architecture (such as the 8086/8088), additive operations between pointer and integer values may not be valid in some cases. For example, an operation may result in an address that is outside the bounds of an array. See your compiler guide for more information on memory models.

The following declarations are used for both examples:

```
int i = 4, j;
float x[10];
float *px;
```

Example 1

In this example, the value of i is multiplied by the length of a **float** and added to $\&x[4]$. The resulting pointer value is the address of $x[8]$.

```
px = &x[4] + i; /* equivalent to px = &x[4+i]; */
```

Example 2

In this example, the address of the third element of x (given by $x[i-2]$) is subtracted from the address of the fifth element of x (given by $x[i]$). The difference is divided by the length of a **float**; the result is the integer value 2.

```
j = &x[i] - &x[i-2];
```

5.3.7 Shift Operators

The shift operators shift their first operand left (<<) or right (>>) by the number of positions the second operand specifies. Both operands must be integral values. These operators perform the usual arithmetic conversions; the type of the result is the type of the left operand after conversion.

For leftward shifts, the vacated right bits are set to 0. For rightward shifts, the vacated left bits are filled based on the type of the first operand after conversion. If the type is **unsigned**, they are set to 0. Otherwise, they are filled with copies of the sign bit.



C Language Reference

The result of a shift operation is undefined if the second operand is negative.

Since the conversions performed by the shift operators do not provide for overflow or underflow conditions, information may be lost if the result of a shift operation cannot be represented in the type of the first operand after conversion.

Example

```
unsigned int x, y, z;

x = 0x00aa;
y = 0x5500;

z = (x << 8) + (y >> 8);
```

In this example, *x* is shifted left eight positions and *y* is shifted right eight positions. The shifted values are added, giving *0xaa55*, and assigned to *z*.

5.3.8 Relational Operators

The binary relational operators compare their first operand to their second operand to test the validity of the specified relationship. The result of a relational expression is 1 if the tested relationship is true and 0 if it is false. The type of the result is **int**.

The relational operators test the following relationships:

- < First operand less than second operand
- > First operand greater than second operand
- <= First operand less than or equal to second operand
- >= First operand greater than or equal to second operand
- = First operand equal to second operand
- != First operand not equal to second operand

The operands can have integral, floating, or pointer type. The types of the operands can be different. Relational operators perform the usual arithmetic conversions on integral and floating type operands. In addition, you

can use the following combinations of operand types with relational operators:

- Both operands of any relational operator can be pointers to the same type. For the equality (`==`) and inequality (`!=`) operators, the result of the comparison indicates whether or not the two pointers address the same memory location. For the other relational operators (`<`, `>`, `<=`, and `>=`), the result of the comparison indicates the relative position of two memory addresses.

Since the address of a given value is arbitrary, comparisons between the addresses of two unrelated values are generally meaningless. However, comparisons between the addresses of different elements of the same array can be useful, since array elements are guaranteed to be stored in order from the first element to the last. The address of the first array element is “less than” the address of the last element.

- A pointer value can be compared to the constant value 0 for equality (`==`) or inequality (`!=`). A pointer with a value of 0, called a “null” pointer, does not point to a memory location.

Example 1

Because x and y are equal, the expression in Example 1 yields the value 0.

```
int x = 0, y = 0;
x < y
```

Example 2

The fragment in Example 2 initializes each element of *array* to a null character constant.

```
char array[10] ;
char *p ;

for (p = array; p < &array[10]; p++)
    *p = '\0' ;
```

Example 3

Example 3 declares an enumeration variable named *col* with the tag *color*. At any time, the variable may contain an integer value of 0, 1, or 2, which represents one of the elements of the enumeration set *color*: the color red, white, or green, respectively. If *col* contains 0 when the `if` statement is executed, any statements depending on the `if` will be executed.

C Language Reference

```
enum color {red, white, green} col;  
.  
.  
.  
if (col == red)  
.  
.  
.
```

5.3.9 Bitwise Operators

The bitwise operators perform bitwise-AND (&), inclusive-OR (|), and exclusive-OR (^) operations. The operands of bitwise operators must have integral types, but their types can be different. These operators perform the usual arithmetic conversions; the type of the result is the type of the operands after conversion.

The C bitwise operators are described as follows:

- & The bitwise-AND operator compares each bit of its first operand to the corresponding bit of its second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

- | The bitwise-inclusive-OR operator compares each bit of its first operand to the corresponding bit of its second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

- ^ The bitwise-exclusive-OR operator compares each bit of its first operand to the corresponding bit of its second operand. If one bit is 0 and the other bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

The following declarations are used for these examples:

```
short i = 0xab00;  
short j = 0xabcd;  
short n;
```

Example 1

The result assigned to *n* in Example 1 is the same as *i* (0xab00 hexadecimal).

```
n = i & j;
```

Example 2

The bitwise-inclusive OR in Example 2 results in the value 0xabcd (hexadecimal).

```
n = i | j;
```

Example 3

The bitwise-exclusive OR in Example 3 produces 0xcd (hexadecimal).

```
n = i ^ j;
```

5.3.10 Logical Operators

The logical operators perform logical-AND (&&) and logical-OR (||) operations. The operands of the logical operators must have integral, floating, or pointer type. The types of the operands can be different.

The operands of logical-AND and logical-OR expressions are evaluated from left to right. If the value of the first operand is sufficient to determine the result of the operation, the second operand is not evaluated. There is a sequence point after the first operand.



Logical operators do not perform the usual arithmetic conversions. Instead, they evaluate each operand in terms of its equivalence to 0.

The result of a logical operation is either 0 or 1. The result's type is **int**.

The C logical operators are described as follows:

&& The logical-AND operator produces the value 1 if both operands have nonzero values. If either operand is equal to 0, the result is 0. If the first operand of a logical-AND operation is equal to 0, the second operand is not evaluated.

|| The logical-OR operator performs an inclusive-OR operation on its operands. The result is 0 if both operands have 0 values. If either operand has a nonzero value, the result is 1. If the first operand of a logical-OR operation has a nonzero value, the second operand is not evaluated.

C Language Reference

The following examples use these declarations:

```
int w, x, y, z;
```

Example 1

In this example, the *printf* function is called to print a message if *x* is less than *y* and *y* is less than *z*. If *x* is greater than *y*, the second operand (*y* < *z*) is not evaluated and nothing is printed. Note that this could cause problems in cases where the second operand has side effects that are being relied on for some other reason.

```
if (x < y && y < z)
    printf ("x is less than z\n");
```

Example 2

In this example, if *x* is equal to either *w*, *y*, or *z*, the second argument to the *printf* function evaluates to true and the value 1 is printed. Otherwise, it evaluates to false and the value 0 is printed. As soon as one of the conditions evaluates to true, evaluation ceases.

```
printf ("%d" , (x==w || x==y || x==z));
```

5.3.11 Sequential-Evaluation Operator

The sequential-evaluation operator evaluates its two operands sequentially from left to right. There is a sequence point after the first operand. The result of the operation has the same value and type as the right operand. Each operand can be of any type. The sequential-evaluation operator does not perform type conversions between its operands.

The sequential-evaluation operator, also called the “comma operator,” is typically used to evaluate two or more expressions in contexts where only one expression is allowed.

Commas can be used as separators in some contexts. However, you must be careful not to confuse the use of the comma as a separator with its use as an operator; the two uses are completely different.

Example 1

In this example, each operand of the **for** statement’s third expression is evaluated independently. The left operand, *i* += *i*, is evaluated first; then the right operand, *j*– –, is evaluated.

```
for ( i = j = 1; i + j < 20; i += i, j-- );
```

Example 2

In the function call to *func_one*, three arguments, separated by commas, are passed: *x*, *y + 2*, and *z*.

In the function call to *func_two*, parentheses force the compiler to interpret the first comma as the sequential-evaluation operator. This function call passes two arguments to *func_two*. The first argument is the result of the sequential-evaluation operation (*x--*, *y + 2*), which has the value and type of the expression *y + 2*; the second argument is *z*.

```
func_one(x, y + 2, z);
func_two((x--, y + 2), z);
```

5.3.12 Conditional Operator

C has one ternary operator: the conditional operator (*? :*). It has the following form:

$$\textit{operand1} \textit{ ? operand2 : operand3}$$

The expression *operand1* must have integral, floating, or pointer type. It is evaluated in terms of its equivalence to 0. A sequence point follows *operand1*. Evaluation proceeds as follows:

- If *operand1* does not evaluate to 0, *operand2* is evaluated, and the result of the expression is the value of *operand2*.
- If *operand1* evaluates to 0, *operand3* is evaluated, and the result of the expression is the value of *operand3*.

Note that either *operand2* or *operand3* is evaluated, but not both.

The type of the result of a conditional operation depends on the type of *operand2* or *operand3*, as follows:

- If *operand2* or *operand3* has integral or floating type (their types can be different), the operator performs the usual arithmetic conversions. The type of the result is the type of the operands after conversion.
- If both *operand2* and *operand3* have the same structure, union, or pointer type, the type of the result is the same structure, union, or pointer type.



C Language Reference

- If both operands have type **void**, the result has type **void**.
- If either operand is a pointer to an object of any type, and the other operand is a pointer to **void**, the pointer to the object is converted to a pointer to **void** and the result is a pointer to **void**.
- If either *operand2* or *operand3* is a pointer and the other operand is a constant expression with the value 0, the type of the result is the pointer type.

Example 1

This example assigns the absolute value of *i* to *j*. If *i* is less than 0, *-i* is assigned to *j*. If *i* is greater than or equal to 0, *i* is assigned to *j*.

```
j = (i < 0) ? (-i) : (i);
```

Example 2

In this example, two functions, *f1* and *f2*, and two variables, *x* and *y*, are declared. Later in the program, if the two variables have the same value, the function *f1* is called. Otherwise, *f2* is called.

```
void f1(void) ;  
void f2(void) ;  
int x ;  
int y ;  
.  
.  
.  
(x==y) ? (f1()) : (f2()) ;
```

5.4 Assignment Operators

The assignment operators in C can both transform and assign values in a single operation. Using a compound-assignment operator to replace two separate operations can make your programs smaller and more efficient.

C provides the following assignment operators:

++ Unary increment

— Unary decrement

=	Simple assignment
*=	Multiplication assignment
/=	Division assignment
%=	Remainder assignment
+=	Addition assignment
-=	Subtraction assignment
<<=	Left-shift assignment
>>=	Right-shift assignment
&=	Bitwise-AND assignment
=	Bitwise-inclusive-OR assignment
^=	Bitwise-exclusive-OR assignment

In assignment, the type of the right-hand value is converted to the type of the left-hand value. The specific conversion path, which depends on the two types, is outlined in detail in Section 5.6.



5.4.1 Lvalue Expressions

An assignment operation assigns the value of the right-hand operand to the storage location named by the left-hand operand. Therefore, the left-hand operand of an assignment operation (or the single operand of a unary assignment expression) must be an expression that refers to a modifiable memory location.

Expressions that refer to memory locations are called “lvalue expressions.” Expressions referring to modifiable locations are “modifiable lvalues.” One example of a modifiable lvalue expression is a variable name declared without the **const** specifier (**non-const**). The name of the variable denotes a storage location, while the value of the variable is the value stored at that location.

The following C expressions may be lvalue expressions:

- An identifier of integral, floating, pointer, structure, or union type

C Language Reference

- A subscript ([]) expression that does not evaluate to an array or a function
- A member-selection expression (-> or .), if the selected member is one of the aforementioned expressions
- A unary-indirection (*) expression that does not refer to an array or function
- An lvalue expression in parentheses
- A **const** object (a nonmodifiable lvalue)

Note

Microsoft C includes an extension to the ANSI C standard allowing a type cast to a pointer type as an lvalue expression, as long as the size of the object does not change. The following example illustrates this feature:

```
char *p ;
int i;
long l;

(long *) p = &l ;      /* legal cast */
(long) i = l ;        /* illegal cast */
```

See your compiler guide for information on enabling and disabling the Microsoft extensions.

5.4.2 Unary Increment and Decrement

The unary assignment operators (++ and --) increment and decrement their operand, respectively. The operand must have integral, floating, or pointer type and must be a modifiable (non-**const**) lvalue expression.

An operand of integral or floating type is incremented or decremented by the integer value 1. The type of the result is the same as the operand type. An operand of pointer type is incremented or decremented by the size of the object it addresses.

An incremented pointer points to the next object; a decremented pointer points to the previous object.

An increment (`++`) or decrement (`--`) operator can appear either before or after its operand, with the following results:

- When the operator appears before its operand, the operand is incremented or decremented and its new value is the result of the expression.
- When the operator appears after its operand, the immediate result of the expression is the value of the operand *before* it is incremented or decremented. After that result is applied in context, the operand is incremented or decremented.

Example 1

In this example, the variable `pos` is compared to 0, then incremented. If `pos` was positive before being incremented, the next statement is executed. First, the value of `q` is assigned to `p`. Then `q` and `p` are incremented.

```
if (pos++ > 0)
    *p++ = *q++;
```



Example 2

In this example, the variable `i` is decremented before it is used as a subscript to `line`.

```
if (line[--i] != '\n')
    return;
```

5.4.3 Simple Assignment

The simple-assignment operator assigns its right operand to its left operand. The conversion rules for assignment apply (see Section 5.6.1).

Example

In this example, the value of `y` is converted to **double** type and assigned to `x`:

```
double x;
int y;

x = y;
```

5.4.4 Compound Assignment

The compound-assignment operators combine the simple-assignment operator with another binary operator. Compound-assignment operators perform the operation specified by the additional operator, then assign the result to the left operand. For example, a compound-assignment expression such as

$$\textit{expression1} \text{ += } \textit{expression2}$$

can be understood as

$$\textit{expression1} = \textit{expression1} + \textit{expression2}$$

However, the compound-assignment expression is not equivalent to the expanded version because the compound-assignment expression evaluates *expression1* only once, while the expanded version evaluates *expression1* twice: in the addition operation and in the assignment operation.

The operands of a compound-assignment operator must be of integral or floating type. Each compound-assignment operator performs the conversions that the corresponding binary operator performs and restricts the types of its operands accordingly. The addition-assignment (+) and subtraction-assignment (-) operators may also have a left operand of pointer type, in which case the right-hand operand must be of integral type. The result of a compound-assignment operation has the value and type of the left operand.

Example

In this example, a bitwise-inclusive-AND operation is performed on *n* and *MASK*, and the result is assigned to *n*. The manifest constant *MASK* is defined with a **#define** preprocessor directive (this directive is discussed in Section 8.2.2.).

```
#define MASK    0xff00  
  
n &= MASK;
```

5.5 Precedence and Order of Evaluation

The precedence and associativity of C operators affect the grouping and evaluation of operands in expressions. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first.

Table 5.1 summarizes the precedence and associativity of C operators, listing them in order of precedence from highest to lowest. Where several operators appear together in a line or large brace, they have equal precedence and are evaluated according to their associativity.

Table 5.1
Precedence and Associativity of C Operators

Symbol^a	Type of Operation	Associativity
() [] . ->	Expression	Left to right
- ~ ! * &	Unary ^b	Right to left
++ -- sizeof casts		
* / %	Multiplicative	Left to right
+ -	Additive	Left to right
<< >>	Shift	Left to right
< > <= >=	Relational (inequality)	Left to right
= = !=	Relational (equality)	Left to right
&	Bitwise AND	Left to right
^	Bitwise-exclusive OR	Left to right
	Bitwise-inclusive OR	Left to right
&&	Logical AND	Left to right
	Logical OR	Left to right
? :	Conditional	Right to left
= *= /= %=	Simple and compound	Right to left
+= -= <<= >>=	assignment ^c	
&= = ^=		
,	Sequential evaluation	Left to right

C Language Reference

- ^a Operators are listed in descending order of precedence. If several operators appear in the same line or in a large brace, they have equal precedence.
- ^b All unary operators have equal precedence.
- ^c All simple and compound-assignment operators have equal precedence.

As Table 5.1 shows, operands consisting of a constant, an identifier, a string, a function call, a subscript expression, a member-selection expression, or a parenthetical expression have the highest precedence and associate from left to right. Type-cast conversions have the same precedence and associativity as the unary operators.

An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator, either from right to left or from left to right. The direction of evaluation does not affect the results of expressions that include more than one multiplication (*), addition (+), or binary-bitwise (& | ^) operator at the same level. The compiler is free to evaluate such expressions in any order, even when parentheses in the expression appear to specify a particular order. Only the sequential-evaluation (,), logical-AND (&&), logical-OR (||), ternary (?:) and function-call operators constitute sequence points, and therefore guarantee a particular order of evaluation for their operands. The function-call operator is the set of parentheses following the function identifier. The sequential-evaluation operator (,) is guaranteed to evaluate its operands from left to right. (Note that the comma separating arguments in a function call is not the same as the sequential-evaluation operator and does not provide any such guarantee.) Sequence points are discussed in Section 5.2.12.

The unary plus operator (+) is intended to force specific groupings in certain situations. It is implemented syntactically, but not semantically. For further information on unary operators, see Section 5.3.2, Complement and Unary Plus Operators.

Logical operators also guarantee evaluation of their operands from left to right. However, they evaluate the smallest number of operands needed to determine the result of the expression. Thus, some operands of the expression may not be evaluated. For example, in the expression `x && y++`, the second operand, `y++`, is evaluated only if `x` is true (nonzero). Thus, `y` is not incremented if `x` is false (0).

The following list shows the default groupings for several sample expressions:

$a \& b \parallel c \quad (a \& b) \parallel c$

$a = b \parallel c \quad a = (b \parallel c)$

$q \&\& r \parallel s-- \quad (q \&\& r) \parallel s--$

In the first expression, the bitwise-AND operator ($\&$) has higher precedence than the logical-OR operator (\parallel), so $a \& b$ forms the first operand of the logical-OR operation.

In the second expression, the logical-OR operator (\parallel) has higher precedence than the simple-assignment operator ($=$), so $b \parallel c$ is grouped as the right-hand operand in the assignment. Note that the value assigned to a is either 0 or 1.

The third expression shows a correctly formed expression that may produce an unexpected result. The logical-AND operator ($\&\&$) has higher precedence than the logical-OR operator (\parallel), so $q \&\& r$ is grouped as an operand. Since the logical operators guarantee evaluation of operands from left to right, $q \&\& r$ is evaluated before $s--$. However, if $q \&\& r$ evaluates to a nonzero value, $s--$ is not evaluated, and s is not decremented. To correct this problem, $s--$ should appear as the first operand of the expression, or s should be decremented in a separate operation.



The following expression is illegal and produces a diagnostic message at compile time:

$p == 0 ? p += 1 : p += 2 \quad (p == 0 ? p += 1 : p) += 2$

In this expression, the equality operator ($==$) has the highest precedence, so $p == 0$ is grouped as an operand. The ternary operator ($? :$) has the next-highest precedence. Its first operand is $p == 0$, and its second operand is $p += 1$. However, the last operand of the ternary operator is considered to be p rather than $p += 2$, since this occurrence of p binds more closely to the ternary operator than it does to the compound-assignment operator. A syntax error occurs because $+= 2$ does not have a left-hand operand. You should use parentheses to prevent errors of this kind and produce more readable code. For example, you could use parentheses as shown to correct and clarify the preceding example:

$(p == 0) ? (p += 1) : (p += 2)$

C Language Reference

5.6 Type Conversions

Type conversions are performed in the following cases:

- When a value of one type is assigned to a variable of a different type
- When a value of one type is explicitly cast to a different type
- When an operator converts the type of its operand or operands before performing an operation
- When a value is passed as an argument to a function

Sections 5.6.1-5.6.4 outline the rules for each kind of conversion.

5.6.1 Assignment Conversions

In assignment operations, the type of the value being assigned is converted to the type of the variable that receives the assignment. C allows conversions by assignment between integral and floating types, even if information is lost in the conversion. The conversion methods used depend on the types involved in the assignment, as described in Section 5.3.1, “Usual Arithmetic Conversion,” and Sections 5.6.1.1 - 5.6.1.5.

Conversions from Signed Integral Types

A signed integer is converted to a shorter signed integer by truncating the high-order bits, and to a longer signed integer by sign extension.

When a signed integer is converted to an unsigned integer, the signed integer is converted to the size of the unsigned integer, and the result is interpreted as an unsigned value.

No information is lost when a signed integer is converted to a floating value, except that some precision may be lost when a **long int** or **unsigned long int** value is converted to a **float** value.

Table 5.2 summarizes conversions from signed integral types. This table assumes that the **char** type is signed by default. If you use a compile-time option to change the default for the **char** type to unsigned, the conversions given in Table 5.3 for the **unsigned char** type apply instead of the conversions in Table 5.2.

Table 5.2
Conversions from Signed Integral Types

From	To	Method
char^a	short	Sign extend
char	long	Sign extend
char	unsigned char	Preserve pattern; high-order bit loses function as sign bit
char	unsigned short	Sign extend to short ; convert short to unsigned short
char	unsigned long	Sign extend to long ; convert long to unsigned long
char	float	Sign extend to long ; convert long to float
char	double	Sign extend to long ; convert long to double
char	long double	Sign extend to long ; convert long to double
short	char	Preserve low-order byte
short	long	Sign extend
short	unsigned char	Preserve low-order byte
short	unsigned short	Preserve bit pattern; high-order bit loses function as sign bit
short	unsigned long	Sign extend to long ; convert long to unsigned long
short	float	Sign extend to long ; convert long to float
short	double	Sign extend to long ; convert long to double
short	long double	Sign extend to long ; convert long to double
long	char	Preserve low-order byte

C Language Reference

long	short	Preserve low-order word
long	unsigned char	Preserve low-order byte
long	unsigned short	Preserve low-order word
long	unsigned long	Preserve bit pattern; high-order bit loses function as sign bit
long	float	Represent as float . If long cannot be represented exactly, some precision is lost.
long	double	Represent as double . If long cannot be represented exactly as a double , some precision is lost.
long	long double	Represent as double . If long cannot be represented exactly as a double , some precision is lost.

^a All **char** entries assume that the **char** type is signed by default.

Note

The **int** type is equivalent to either the **short** type or the **long** type, depending on the implementation. Conversion of an **int** value proceeds the same as for a **short** or a **long**, whichever is appropriate.

Conversions from Unsigned Integral Types

An unsigned integer is converted to a shorter unsigned or signed integer by truncating the high-order bits, or to a longer unsigned or signed integer by zero extending.

When an unsigned integer is converted to a signed integer of the same size, the bit pattern does not change. However, the value it represents changes if the sign bit is set.

Unsigned integer values are converted to floating values by first converting the unsigned integer value to a signed **long** value, then converting that signed **long** value to a floating value.

Table 5.3 summarizes conversions from unsigned integral types.

Table 5.3
Conversions from Unsigned Integral Types

From	To	Method
unsigned char	char	Preserve bit pattern; high-order bit becomes sign bit
unsigned char	short	Zero extend
unsigned char	long	Zero extend
unsigned char	unsigned short	Zero extend
unsigned char	unsigned long	Zero extend
unsigned char	float	Convert to long ; convert long to float
unsigned char	double	Convert to long ; convert long to double
unsigned char	long double	Convert to long ; convert long to double
unsigned short	char	Preserve low-order byte
unsigned short	short	Preserve bit pattern; high-order bit becomes sign bit
unsigned short	long	Zero extend



C Language Reference

unsigned short	unsigned char	Preserve low-order byte
unsigned short	unsigned long	Zero extend
unsigned short	float	Convert to long ; convert long to float
unsigned short	double	Convert to long ; convert long to double
unsigned short	long double	Convert to long ; convert long to double
unsigned long	char	Preserve low-order byte
unsigned long	short	Preserve low-order word
unsigned long	long	Preserve bit pattern; high-order bit becomes sign bit
unsigned long	unsigned char	Preserve low-order byte
unsigned long	unsigned short	Preserve low-order word
unsigned long	float	Convert to long ; convert long to float
unsigned long	double	Convert to long ; convert long to double
unsigned long	long double	Convert to long ; convert long to double

Note

The **unsigned int** type is equivalent either to the **unsigned short** type or to the **unsigned long** type, depending on the implementation. Conversion of an **unsigned int** value proceeds in the same way as conversion of an **unsigned short** or an **unsigned long**, whichever is appropriate.

Conversions from **unsigned long** values to **float**, **double**, or **long double** are not accurate if the value being converted is larger than the maximum positive **long** value.

Conversions from Floating-Point Types

A **float** value converted to a **double** value undergoes no change in value. A **double** value converted to a **float** value is represented exactly, if possible. Precision may be lost if the value cannot be represented exactly.

A floating value is converted to an integral value by first converting to a **long**, then from the **long** value to the specific integral value, as described in Table 5.4. The decimal portion of the floating value is discarded in the conversion to a **long**; if the result is still too large to fit into a **long**, the result of the conversion is undefined.

Table 5.4 summarizes conversions from floating types.

Table 5.4
Conversions from Floating-Point Types

From	To	Method
float	char	Convert to long ; convert long to char
float	short	Convert to long ; convert long to short
float	long	Truncate at decimal point. If result is too large to be represented as long , result is undefined.
float	unsigned short	Convert to long ; convert long to unsigned short
float	unsigned long	Convert to long ; convert long to unsigned long
float	double	Change internal representation
float	long double	Change internal representation
double	char	Convert to float ; convert float to char
double	short	Convert to float ; convert float to short
double	long	Truncate at decimal point. If result is too large to be represented as long , result is undefined.
double	unsigned short	Convert to long ; convert long to unsigned short

C Language Reference

double	unsigned long	Convert to long ; convert long to unsigned long
double	float	Represent as a float . If double value cannot be represented exactly as float , loss of precision occurs. If value is too large to be represented as float , the result is undefined.
long double	char	Convert to float ; convert float to char
long double	short	Convert to float ; convert float to short
long double	long	Truncate at decimal point. If result is too large to be represented as long , result is undefined.
long double	unsigned short	Convert to long ; convert long to unsigned short
long double	unsigned long	Convert to long ; convert long to unsigned long
long double	float	Represent as a float . If double value cannot be represented exactly as float , loss of precision occurs. If value is too large to be represented as float , the result is undefined.
long double	double	The long double value is treated as double .

Note

Conversions from **float**, **double**, or **long double** values to **unsigned long** are not accurate if the value being converted is larger than the maximum positive **long** value.

Conversions to and from Pointer Types

A pointer to one type of value can be converted to a pointer to a different type. However, the result may be undefined because of the alignment requirements and sizes of different types in storage.

A pointer to **void** may be converted to or from a pointer to any type, without restriction.

In some implementations, you can use the special keywords **near**, **far**, and **huge** to change the size of pointers within a program. The conversion path depends on your implementation. For example, on an 8086 processor, the compiler might use a segment-register value to convert a 16-bit pointer to a 32-bit pointer. For information about pointer conversions, see your compiler guide.

A pointer value can also be converted to an integral value. The conversion path depends on the size of the pointer and the size of the integral type, according to the following rules:

- If the size of the pointer is greater than or equal to the size of the integral type, the pointer behaves like an unsigned value in the conversion, except that it cannot be converted to a floating value.
- If the pointer is smaller than the integral type, the pointer is first converted to a pointer with the same size as the integral type, then converted to the integral type. The implementation determines how a pointer is converted to a longer pointer; for information about pointer conversions, see your compiler guide.

5

Conversely, an integral type can be converted to a pointer type according to the following rules:

- If the integral type is the same size as the pointer type, the conversion simply causes the integral value to be treated as a pointer (an unsigned integer).
- If the size of the integral type is different from the size of the pointer type, the integral type is first converted to the size of the pointer, using the conversion paths given in Tables 5.2 and 5.3. It is then treated as a pointer value.

If the special keywords **near**, **far**, and **huge** are implemented, implicit conversions may be made on pointer values. In particular, the compiler may make assumptions about the default size of pointers and convert passed pointer values accordingly, unless a forward declaration is present to override the implicit conversion. For information about pointer conversions, see your compiler guide.

C Language Reference

Conversions from Other Types

Since an **enum** value is an **int** value by definition, conversions to and from an **enum** value are the same as those for the **int** type. An **int** is equivalent to either a **short** or a **long**, depending on the implementation.

No conversions between structure or union types are allowed.

The **void** type has no value, by definition. Therefore, it cannot be converted to any other type, and other types cannot be converted to **void** by assignment. However, you can explicitly cast a value to **void** type, as discussed in Section 5.6.2.

5.6.2 Type-Cast Conversions

You can use type casts to explicitly convert types. A type cast has the form

(type-name)operand

where *type-name* is a type and *operand* is a value to be converted to that type. (Type names are discussed in Section 4.9.)

The operand is converted as though it had been assigned to a variable of *type-name* type. The conversion rules for assignments (outlined in Section 5.6.1) apply to type casts as well.

You can use the type name **void** in a cast operation, but you cannot assign the resulting expression to any item.

5.6.3 Operator Conversions

The conversions performed by C operators depend on the operator and on the type of the operand or operands. Many operators perform the usual arithmetic conversions, outlined in Section 5.3.1.

C permits some arithmetic with pointers. In pointer arithmetic, integer values are converted to express memory positions. (For more information, see the discussions of additive operators, Section 5.3.6, and subscript expressions, Section 5.2.5.)

5.6.4 Function-Call Conversions

The type of conversion performed on the arguments in a function call depends on the presence of a function prototype (forward declaration) with declared argument types for the called function.

If a function prototype is present and includes declared argument types, the compiler performs type checking. The type-checking process is outlined in detail in the chapter on “Functions.”

If no function prototype is present, or if an old-style forward declaration omits the argument-type list, only the usual arithmetic conversions are performed on the arguments in the function call. These conversions are performed independently on each argument in the call. This means that a **float** value is converted to a **double**; a **char** or **short** value is converted to an **int**; and an **unsigned char** or **unsigned short** is converted to an **unsigned int**.

If the special keywords **near**, **far**, and **huge** are implemented, implicit conversions can also be made on pointer values passed to functions. You can override these implicit conversions by providing function prototypes to let the compiler perform type checking. For information about pointer conversions, see your compiler guide.



Chapter 6

Statements

- 6.1 Introduction 6-1
- 6.2 The break Statement 6-2
- 6.3 The Compound Statement 6-3
- 6.4 The continue Statement 6-4
- 6.5 The do Statement 6-4
- 6.6 The Expression Statement 6-5
- 6.7 The for Statement 6-6
- 6.8 The goto and Labeled Statements 6-8
- 6.9 The if Statement 6-9
- 6.10 The Null Statement 6-10
- 6.11 The return Statement 6-11
- 6.12 The switch Statement 6-13
- 6.13 The while Statement 6-15

6.1 Introduction

The statements of a C program control the flow of program execution. In C, as in other programming languages, several kinds of statements are available to perform loops, to select other statements to be executed, and to transfer control. This chapter describes C statements in alphabetical order, as follows:

break statement	goto and labeled statements
compound statement	if statement
continue statement	null statement
do statement	return statement
expression statement	switch statement
for statement	while statement

C statements consist of keywords, expressions, and other statements. The following keywords appear in C statements:

break	default	for	return
case	do	goto	switch
continue	else	if	while

The expressions in C statements are the expressions discussed in the “Expressions and Assignments” chapter. Statements appearing within C statements may be any of the statements discussed in this chapter. A statement that forms a component of another statement is called the “body” of the enclosing statement. Frequently the statement body is a “compound” statement: a single statement composed of one or more statements.

The compound statement is delimited by braces ({ }); all other C statements end with a semicolon(;).

Any C statement may begin with an identifying label consisting of a name and a colon. Since only the **goto** statement recognizes statement labels, statement labels are described along with the **goto** statement in Section 6.8.

When a C program is executed, its statements are executed in the order in which they appear in the program, except where a statement explicitly transfers control to another location.

C Language Reference

6.2 The break Statement

Syntax

```
break;
```

Execution

The **break** statement terminates the execution of the smallest enclosing **do**, **for**, **switch**, or **while** statement in which it appears. Control passes to the statement that follows the terminated statement. A **break** statement can appear only within a **do**, **for**, **switch**, or **while** statement.

Within nested statements, the **break** statement terminates only the **do**, **for**, **switch**, or **while** statement that immediately encloses it. You can use a **return** or **goto** statement to transfer control out of the nested structure.

Example

This example processes an array of variable-length strings stored in *lines*. The **break** statement causes an exit from the interior **for** loop after the terminating null character (`\0`) of each string is found and its position is stored in *lengths[i]*. Control then returns to the outer **for** loop. The variable *i* is incremented and the process is repeated until *i* is greater than or equal to *LENGTH*.

```
for (i = 0; i < LENGTH; i++) {
    for (j = 0; j < WIDTH; j++) {
        if (lines[i][j] == '\0') {
            lengths[i] = j;
            break;
        }
    }
}
```

6.3 The Compound Statement

Syntax

```

{
  [declaration]
  .
  .
  .
  statement
  [statement]
  .
  .
  .
}
```

Execution

A compound statement typically appears as the body of another statement, such as the **if** statement. When a compound statement is executed, its statements are executed in the order in which they appear, except where a statement explicitly transfers control to another location. The “Declarations” chapter describes the form and meaning of the declarations that can appear at the head of a compound statement.

Like other C statements, any of the statements in a compound statement can carry a label. Labeled statements are discussed in Section 6.8.

6

Example

In this example, if *i* is greater than 0, all of the statements in the compound statement are executed in order.

```

if (i > 0) {
    line[i] = x;
    x++;
    i--;
}
```

C Language Reference

6.4 The continue Statement

Syntax

```
continue;
```

Execution

The **continue** statement passes control to the next iteration of the **do**, **for**, or **while** statement in which it appears, bypassing any remaining statements in the **do**, **for**, or **while** statement body. The next iteration of a **do**, **for**, or **while** statement is determined as follows:

- Within a **do** or a **while** statement, the next iteration starts by re-evaluating the expression of the **do** or **while** statement.
- Within a **for** statement, the next iteration starts by evaluating the loop expression of the **for** statement. Then it evaluates the conditional expression and, depending on the result, either terminates or iterates the statement body. (The **for** statement is discussed in Section 6.7.)

Example

In this example, the statement body is executed if i is greater than 0. First $f(i)$ is assigned to x ; then, if x is equal to 1, the **continue** statement is executed. The rest of the statements in the body are ignored, and execution resumes at the top of the loop with the evaluation of $i-- > 0$.

```
while (i-- > 0) {  
    x = f(i);  
    if (x == 1)  
        continue;  
    y += x * x;  
}
```

6.5 The do Statement

Syntax

```
do  
    statement  
while (expression);
```

Execution

The body of a **do** statement is executed one or more times until *expression* becomes false (0). Execution proceeds as follows:

1. The statement body is executed.
2. The *expression* is evaluated. If *expression* is false, the **do** statement terminates and control passes to the next statement in the program. If *expression* is true (nonzero), the process is repeated, beginning with step 1.

The **do** statement may also terminate when a **break**, **goto**, or **return** statement is executed within the statement body.

Example

In this **do** statement, the two statements $y = f(x)$; and $x--$; are executed, regardless of the initial value of x . Then $x > 0$ is evaluated. If x is greater than 0, the statement body is executed again and $x > 0$ is reevaluated. The statement body is executed repeatedly as long as x remains greater than 0. Execution of the **do** statement terminates when x becomes 0 or negative. The body of the loop is executed at least once.

```
do {
    y = f(x);
    x--;
} while (x > 0);
```

6

6.6 The Expression Statement

Syntax

expression;

Execution

When an expression statement is executed, the expression is evaluated according to the rules outlined in the “Expressions and Assignments” chapter.

In C, assignments are expressions. The value of the expression is the value being assigned (sometimes called the “right-hand value”).

C Language Reference

Function calls are also considered expressions. The value of the expression is the value, if any, returned by the function. If a function returns a value, the expression statement usually includes an assignment to store the returned value when the function is called. The value returned by the function is usually used as an operand in another expression. If the value is to be used more than once, it can be assigned to another variable. If the value is neither used as an operand nor assigned, the function is called but the return value, if any, is not used.

Example 1

In this example, x is assigned the value of $y + 3$.

```
x = (y + 3);
```

Example 2

In this example, x is incremented.

```
x++;
```

Example 3

This example shows a function-call expression. The value of the expression, which includes any value returned by the function, is assigned to the variable z .

```
z = f(x) + 3;
```

6.7 The for Statement

Syntax

```
for ( [init-expression] ; [cond-expression] ; [loop-expression] )  
    statement
```

Execution

The body of a **for** statement is executed zero or more times until the optional *cond-expression* becomes false. You can use the optional *init-expression* and *loop-expression* to initialize and change values during the **for** statement's execution.

Execution of a **for** statement proceeds as follows:

1. The *init-expression*, if any, is evaluated.
2. The *cond-expression*, if any, is evaluated. Three results are possible:
 - If *cond-expression* is true (nonzero), *statement* is executed; then *loop-expression*, if any, is evaluated. The process then begins again with the evaluation of *cond-expression*.
 - If *cond-expression* is omitted, *cond-expression* is considered true, and execution proceeds exactly as described for case *a*. A **for** statement without a *cond-expression* argument terminates only when a **break** or **return** statement within the statement body is executed, or when a **goto** (to a labeled statement outside the **for** statement body) is executed.
 - If *cond-expression* is false, execution of the **for** statement terminates and control passes to the next statement in the program.

A **for** statement also terminates when a **break**, **goto**, or **return** statement within the statement body is executed.

Example

This example counts space (`'\x20'`) and tab (`'\t'`) characters in the array of characters named *line* and replaces each tab character with a space. First *i*, *space*, and *tab* are initialized to 0. Then *i* is compared with the constant *MAX*; if *i* is less than *MAX*, the statement body is executed. Depending on the value of *line* [*i*], the body of one or neither of the **if** statements is executed. Then *i* is incremented and tested against *MAX*; the statement body is executed repeatedly as long as *i* is less than *MAX*.

```

for (i = space = tab = 0; i < MAX; i++) {
    if (line[i] == ' ')
        space++;
    if (line[i] == '\t') {
        tab++;
        line[i] = ' ';
    }
}

```

6.8 The goto and Labeled Statements

Syntax

```
goto name;  
.  
.  
.  
name: statement
```

Execution

The **goto** statement transfers control directly to the statement that has *name* as its label. The labeled statement is executed immediately after the **goto** statement is executed. A statement with the given label must reside in the same function, and the given label can appear before only one statement in the same function.

A statement label is meaningful only to a **goto** statement; in any other context, a labeled statement is executed without regard to the label.

A label name is simply an identifier. (Section 2.4 describes the rules that govern the construction of identifiers.) Each statement label must be distinct from other statement labels in the same function.

Like other C statements, any of the statements in a compound statement can carry a label. Thus, you can use a **goto** statement to transfer into a compound statement. However, transferring into a compound statement is dangerous when the compound statement includes declarations that initialize variables. Since declarations appear before the executable statements in a compound statement, transferring directly to an executable statement within the compound statement bypasses the initializations. The results are undefined.

Example

In this example, a **goto** statement transfers control to the point labeled *exit* if an error occurs.

```

if (errorcode > 0)
    goto exit;
.
.
.
exit:
return (errorcode);

```

6.9 The if Statement

Syntax

```

if (expression)
    statement1
[ else
    statement2 ]

```

Execution

The body of an **if** statement is executed selectively, depending on the value of *expression*, described as follows:

1. The *expression* is evaluated.
 - If *expression* is true (nonzero), *statement1* is executed.
 - If *expression* is false, *statement2* is executed.
 - If *expression* is false and the **else** clause is omitted, *statement1* is ignored.
2. Control passes from the **if** statement to the next statement in the program.

6

Example 1

In this example, the statement $y = x/i$; is executed if i is greater than 0. If i is less than or equal to 0, i is assigned to x and $f(x)$ is assigned to y . Note that the statement forming the **if** clause ends with a semicolon.

```

if (i > 0)
    y = x/i;
else {
    x = i;
    y = f(x); }

```

Note

C does not offer an “else if” statement, but you can achieve the same effect by nesting **if** statements. An **if** statement can be nested within either the **if** clause or the **else** clause of another **if** statement.

When nesting **if** statements and **else** clauses, use braces to group the statements and clauses into compound statements that clarify your intent. If no braces are present, the compiler resolves ambiguities by pairing each **else** with the most recent **if** lacking an **else**.

Example 2

In this example, the **else** clause is associated with the inner **if** statement. If *i* is less than or equal to 0, no value is assigned to *x*.

```
if (i > 0)                /* Without braces */
    if (j > i)
        x = j;
    else
        x = i;
```

Example 3

In this example, the braces surrounding the inner **if** statement make the **else** clause part of the outer **if** statement. If *i* is less than or equal to 0, *i* is assigned to *x*.

```
if (i > 0) {              /* With braces */
    if (j > i)
        x = j;}
else
    x = i;
```

6.10 The Null Statement

Syntax

```
;
```

Execution

A “null statement” is a statement containing only a semicolon; it may appear wherever a statement is expected. Nothing happens when a null statement is executed.

Statements such as **do**, **for**, **if**, and **while** require that an executable statement appear as the statement body. The null statement satisfies the syntax requirement in cases that do not need a substantive statement body.

As with any other C statement, you can include a label before a null statement. To label an item that is not a statement, such as the closing brace of a compound statement, you can label a null statement and insert it immediately before the item to get the same effect.

Example

In this example, the loop expression of the **for** statement `line[i++] = 0` initializes the first 10 elements of `line` to 0. The statement body is a null statement, since no further statements are necessary.

```
for (i = 0; i < 10; line[i++] = 0)
    ;
```

6.11 The return Statement

Syntax

```
return [expression];
```

Execution

The **return** statement terminates the execution of the function in which it appears and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined.

By convention, parentheses enclose the *expression* argument of the **return** statement. However, C does not require the parentheses.

If no **return** statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function is executed. The return value of the called function is undefined. If a return value is not required, declare the function to have **void** return



C Language Reference

type.

Example

In this example, the *main* function calls two functions: *sq* and *draw*. The *sq* function returns the value of $x * x$ to *main*, where the return value is assigned to *y*. The *draw* function is declared as a **void** function and does not return a value. An attempt to assign the return value of *draw* would cause a diagnostic message to be issued.

```
main()
{
    void draw(int,int);
    long sq(int);
    .
    .
    .
    y = sq(x);
    draw(x, y);
    .
    .
    .
}

long sq(x)
int x;
{
    return (x * x);
}

void draw(x,y)
int x, y;
{
    .
    .
    .
    return;
}
```

6.12 The switch Statement

Syntax

```

switch (expression) {
    [declaration]
    .
    .
    .
    [case constant-expression :]
    .
    .
    .
    [statement]
    .
    .
    .
    [default :
    [statement]]
}

```

Execution

The **switch** statement transfers control to a statement within its body. Control passes to the statement whose **case** *constant-expression* matches the value of **switch** *expression*. The **switch** statement may include any number of **case** instances. Execution of the statement body begins at the selected statement and proceeds until the end of the body or until a statement transfers control out of the body.

6

The **default** statement is executed if no **case** *constant-expression* is equal to the value of **switch** *expression*. If the **default** statement is omitted, and no **case** match is found, none of the statements in the **switch** body is executed. The **default** statement need not come at the end; it can appear anywhere in the body of the **switch** statement.

The type of **switch** *expression* must be integral, but the resulting value is converted to **int**. Each **case** *constant-expression* is then converted using the usual arithmetic conversions. The value of each **case** *constant-expression* must be unique within the statement body. If the type of **switch** *expression* is larger than **int**, a diagnostic message is issued.

The **case** and **default** labels of the **switch** statement body are significant only in the initial test that determines where execution starts in the statement body. All statements between the statement where execution starts and the end of the body are executed regardless of their labels, unless a statement transfers control out of the body entirely.

Note

Declarations can appear at the head of the compound statement forming the **switch** body, but initializations included in the declarations are not performed. The **switch** statement transfers control directly to an executable statement within the body, bypassing the lines that contain initializations.

Example 1

In this example, all three statements of the **switch** body are executed if *c* is equal to *Execution control is transferred to the first statement (capa++;)* and continues in order through the rest of the body. If *c* is equal to *lettera* and *total* are incremented. Only *total* is incremented if *c* is not equal to or

```
switch (c) {
    case 'A':
        capa++;
    case 'a':
        lettera++;
    default :
        total++;
}
```

Example 2

In this example, a **break** statement follows each statement of the **switch** body. The **break** statement forces an exit from the statement body after one statement is executed. If *i* is equal to -1 , only *n* is incremented. The **break** following the statement *n++*; causes execution control to pass out of the statement body, bypassing the remaining statements. Similarly, if *i* is equal to 0, only *z* is incremented; if *i* is equal to 1, only *p* is incremented. The final **break** statement is not strictly necessary, since control passes out of the body at the end of the compound statement, but it is included for consistency.

```

switch (i) {
    case -1:
        n++;
        break;
    case 0 :
        z++;
        break;
    case 1 :
        p++;
        break;
}

```

Multiple Labels

A single statement can carry multiple **case** labels, as the following example shows:

```

case 'a' :
case 'b' :
case 'c' :
case 'd' :
case 'e' :
case 'f' : hexcvt (c);

```

Although you can label any statement within the body of the **switch** statement, no statement is required to carry a label. You can freely intermingle statements with and without labels. Keep in mind, however, that once the **switch** statement passes control to a statement within the body, all following statements in the block are executed, regardless of their labels.

6

6.13 The while Statement

Syntax

```

while (expression)
    statement

```

Execution

The body of a **while** statement is executed zero or more times until *expression* becomes false (0). Execution proceeds as follows:

1. The *expression* is evaluated.
2. If *expression* is initially false, the body of the **while** statement is never executed, and control passes from the **while** statement to the next statement in the program.

C Language Reference

If *expression* is true (nonzero), the body of the statement is executed and the process is repeated beginning at step 1.

The **while** statement may also terminate when a **break**, **goto**, or **return** within the statement body is executed.

Example

This example copies characters from *string2* to *string1*. If *i* is greater than or equal to 0, *string2*[*i*] is assigned to *string1*[*i*] and *i* is decremented. When *i* reaches or falls below 0, execution of the **while** statement terminates.

```
while (i >= 0) {
    string1[i] = string2[i];
    i--;
}
```

Chapter 7

Functions

- 7.1 Introduction 7-1
- 7.2 Function Definitions 7-3
 - 7.2.1 Storage Class 7-4
 - 7.2.2 Return Type and Function Name 7-5
 - 7.2.3 Formal Parameters 7-7
 - 7.2.4 Function Body 7-11
- 7.3 Function Prototypes (Declarations) 7-12
- 7.4 Function Calls 7-14
 - 7.4.1 Actual Arguments 7-17
 - 7.4.2 Calls with a Variable 7-20
 - 7.4.3 Recursive Calls 7-21

7.1 Introduction

A function is an independent collection of declarations and statements, usually designed to perform a specific task. C programs have at least one function, the **main** function, and they may have other functions. This chapter describes how to define, declare, and call C functions.

A function *definition* specifies the name of the function, the types and number of its formal parameters, and the declarations and statements that determine what it does. These declarations and statements are called the “function body.” The function definition also gives the function’s return type and its storage class. If the return type and storage class are not stated explicitly, they default to **int** and **extern**, respectively.

A function *prototype* (or declaration) establishes the name, return type, and storage class of a function fully defined elsewhere in the program. It can also include declarations giving the types and number of the function’s formal parameters. The formal parameter declarations can name the formal parameters, although such names go out of scope at the end of the declaration. The storage class **register** can also be specified for a formal parameter.

Example

This example contrasts the concise and clear prototype declaration and definition formats, and illustrates that the function prototype has the same form as the function definition except that the prototype ends with a semicolon instead of a function body.

The compiler uses the prototype or declaration to compare the types of actual arguments in subsequent calls to the function with the function’s formal parameters, even in the absence of an explicit definition of the function. Explicit prototypes and declarations are optional for functions whose return type is **int**. However, to ensure correct behavior, you must declare or define functions with other return types before calling them. (Function prototype declarations are discussed further in “Function Definitions (Prototypes)” in this chapter and in the “Declarations” chapter.)

If no prototype or declaration is provided, a default prototype is created from whatever information accompanies the first reference to the function name, whether that reference occurs in a call or a definition. However, such a default prototype may not adequately represent a subsequent definition of, or call to, the function.

A function “call” passes execution control from the calling function to the called function. The actual arguments, if any, are passed by value to

C Language Reference

the called function. Execution of a **return** statement in the called function returns control and possibly a value to the calling function.

Note

The use of function prototypes is strongly recommended. Sometimes they provide the only basis on which the compiler can enforce correct argument passing. Prototypes let the compiler either diagnose, or handle correctly, argument mismatches that would otherwise be undetectable until program execution.

The Microsoft C Compiler can generate function prototypes automatically from program source files. These can then be stored in a file that can be included in the compilation of the program. See your compiler guide for more information.

```
/** Prototype-Style Function Declarations and Definitions **/  
  
double new_style(int a, double *x);      /* Function  
                                         Prototype      */  
double alt_style (int, double *);       /* Alternative  
                                         Prototype form */  
double old_style ();                    /* Obsolete  
   .                                     * form of function  
   .                                     * declaration  
   .                                     */  
double new_style(int a, double *real)  /* Prototype-style */  
{                                       /* Function      */  
    return (*real + a) ;                /* Definition   */  
}  
  
double alt_style(a , real)              /* Old Form of   */  
double *real ;                          /* Function      */  
int a ;                                  /* Definition   */  
{  
    return (*real + a) ;  
}
```

7.2 Function Definitions

Syntax

```
[sc-specifier][type-specifier] declarator ([formal-parameter-list])  
function-body
```

A “function definition” specifies the name, formal parameters, and body of a function. It can also stipulate the function’s return type and storage class.

The optional *sc-specifier* gives the function’s storage class, which must be either **static** or **extern**.

The optional *type-specifier* and mandatory *declarator* together specify the function’s return type and name. The *declarator* is a combination of the identifier that names the function and the parentheses following the function name.

The *formal-parameter-list* is a sequence of formal parameter declarations separated by commas. The following syntax illustrates the form of each formal parameter in a formal parameter list.

```
[register] type-specifier [declarator]  
[,...]
```

The formal parameter list contains declarations for the function’s parameters. If no arguments are to be passed to the function, the list should contain the keyword **void**. The empty parentheses form `(())` can be used, but is obsolete and, if used, conveys no information about whether arguments will be passed. The formal parameter list can be full or partial. The second line of the syntax above shows the “ellipsis notation,” a comma followed by three periods `(,...)`. A partial formal parameter list can be terminated by the ellipsis notation to indicate that there may be more arguments passed to the function, but no more information is given about them. Type checking is not performed on such arguments. At least one formal parameter must precede the ellipsis notation and the ellipsis notation must be the last token in the formal parameter list. Without the ellipsis notation, the behavior of a function is undefined if it receives parameters in addition to those declared in the formal parameter list. When a prototype is available, argument checking and conversion are automatically performed. If no information is given concerning the formal parameters, any undeclared arguments simply undergo the usual arithmetic conversions.

C Language Reference

The *type-specifier* can be omitted only if **register** storage class is specified for a value of **int** type.

The *function-body* is a compound statement containing local variable declarations, references to externally declared items, and statements.

Note

The old forms for function declaration and definition are still supported, but considered obsolete. Use of the prototype form is recommended in new code. The old function-definition form is represented in the following syntax:

```
[ sc-specifier ][ type-specifier ] declarator ( [ identifier-list ] )  
[ parameter-declarations ]  
function-body
```

The *identifier-list* is an optional list of identifiers that the function will use as the names of formal parameters. The *parameter-declaration* arguments establish the types of the formal parameters.

Sections 7.2.1-7.2.4 describe the parts of a function definition in detail.

7.2.1 Storage Class

The storage-class specifier in a function definition gives the function either **extern** or **static** storage class. If a function definition does not include a storage-class specifier, the storage class defaults to **extern**. You can explicitly give the **extern** storage-class specifier in a function definition, but it is not required.

A function with **static** storage class is visible only in the source file in which it is defined. All other functions, whether they are given **extern** storage class explicitly or implicitly, are visible throughout all the source files that make up the program.

If **static** storage class is desired, it must be declared on the first occurrence of a declaration (if any) of the function, and on the definition of the function.

Note

A Microsoft extension to the ANSI C standard offers some latitude on functions declared without a storage-class specifier. When the extensions are enabled, a function originally declared without a storage class (or with **extern** storage class) is given **static** storage class if the function definition is in the same source file and explicitly specifies **static** storage class. For information on enabling and disabling extensions, see your compiler guide.

7.2.2 Return Type and Function Name

Syntax

[sc-specifier][type-specifier] declarator ([formal-parameter-list])

The return type of a function establishes the size and type of the value returned by the function and corresponds to *type-specifier* in the syntax above. The *type-specifier* can specify any fundamental, structure, or union type. If you do not include *type-specifier*, the return type **int** is assumed.

The *declarator* is the function identifier, which may be modified to a pointer type. The parentheses following the identifier establish the item as a function. Functions cannot return arrays or functions, but they can return pointers to any type, including arrays and functions.

The return type given in the function definition must match the return type in declarations of the function elsewhere in the program. You need not declare functions with **int** return type before you call them, although prototypes are recommended so that correct argument checking will be enabled. However, functions with other return types must be defined or declared before they are called.

A function's return type is used only when the function returns a value. A function returns a value when a **return** statement containing an expression is executed. The expression is evaluated, converted to the return value type if necessary, and returned to the point at which the function was called. If no **return** statement is executed, or if the **return** statement

C Language Reference

does not contain an expression, the return value is undefined. If the calling function expects a return value, the behavior of the program is also undefined.

Example 1

In this example, the return type of *add* is **int** by default. The function has **static** storage class, which means that only functions in the same source file can call it. The formal parameters declared in the header include one **int** value, *x*, for which register storage is requested, and a second **int** value, *y*. The second function, *subtract*, is defined in the old form. Its return type is **int** by default. The formal parameters are declared between the header and the opening brace.

```
/* prototype-style definition: */
static add (register x, int y)
{
    return (x+y);
}

/* old-style definition: */
subtract (x , y)
    int x, y;
{
    return (x-y);
}
```

Example 2

This example defines the *STUDENT* type with a **typedef** declaration and defines the function *sortstu* to have *STUDENT* return type. The function selects and returns one of its two structure arguments. This prototype-style definition has the formal parameters declared in the header. In subsequent calls to the function, the compiler checks to make sure the argument types are *STUDENT*. Efficiency would be enhanced by passing pointers to the structure, rather than the entire structure.

```

typedef struct {
    char name[20];
    int id;
    long class;
} STUDENT;

    /* return type is STUDENT: */

STUDENT sortstu (STUDENT a, STUDENT b)
{
    return ( (a.id < b.id) ? a : b );
}

```

Example 3

This example uses the old form to define a function returning a pointer to an array of characters. The function takes two character arrays (strings) as arguments and returns a pointer to the shorter of the two strings. A pointer to an array points to the type of the array elements; thus, the return type of the function is pointer to **char**.

```

    /* return type is char pointer: */

Char *smallstr(s1, s2)
char s1[], s2[];
{
    int i;

    i=0;
    while ( s1[i] != '\0' && s2[i] != '\0' )
        i++;
    if ( s1[i] == '\0' )
        return (s1);
    else
        return (s2);
}

```

7.2.3 Formal Parameters

“Formal parameters” are variables that receive values passed to a function by a function call. In a function prototype-style definition, the parentheses following the function name contain complete declarations of the function’s formal parameters.

Note

In the old form of a function definition, the formal parameters were declared following the closing parenthesis, immediately before the beginning of the compound statement constituting the function body. In that form, an identifier list within the parentheses specifies the name of each of the formal parameters and the order in which they take on values in the function call. The identifier list consists of zero or more identifiers, separated by commas. The list must be enclosed in parentheses, even if it is empty. This form is obsolete and should not be used in new code.

If at least one formal parameter occurs in the formal parameter list, the list can end with a comma followed by three periods (...). This construction, called the “ellipsis notation,” indicates a variable number of arguments to the function. However, a call to the function is expected to have at least as many arguments as there are formal parameters before the last comma. In the obsolete definition form, the ellipsis notation can follow the last identifier in the identifier list.

If no arguments are to be passed to the function, the list of formal parameters is replaced by the keyword **void**. This use of **void** is distinct from its use as a type specifier.

Note

To maintain compatibility with previous versions, a Microsoft extension to the ANSI C standard allows a comma without trailing periods (,) at the end of the list of formal parameters to indicate a variable number of arguments. However, it is recommended that code be changed to incorporate the ellipsis notation. For information on enabling and disabling extensions, see your compiler guide.

Formal parameter declarations specify the types, sizes, and identifiers of values stored in the formal parameters. In the obsolete function definition form, these declarations have the same form as other variable declarations (see the “Declarations” chapter). However, in a function prototype-style definition, each identifier in the *formal-parameter-list* must be preceded by its appropriate type specifier. For example, in the

following (obsolete form) definition of the function *old*, *double x, y, z* ; can be declared simply by separating identifiers with commas:

```
void old(x, y, z)
    double z, y ;
    double x ;
    {
        ;
    }

void new(double x, double y, double z)
    {
        ;
    }
```

The function called *new* is defined in prototype format, with a list of formal parameters in the parentheses. In this form, the type specifier *double* must be repeated for each identifier.

The order and type of formal parameters, including any use of the ellipsis notation, must be the same in all the function declarations (if any) and in the function definition. The types of the actual arguments in calls to a function must be assignment compatible with the types of the corresponding formal parameters, up to the point of the ellipsis notation. Arguments following the ellipsis are not checked. A formal parameter can have any fundamental, structure, union, pointer, or array type.

The only storage class you can specify for a formal parameter is **register**. Undeclared identifiers in the parentheses following the function name are assumed to have **int** type. In the old function-definition form, formal parameter declarations can be in any order.

The identifiers of the formal parameters are used in the function body to refer to the values passed to the function. These identifiers cannot be redefined in the outermost block of the function body, but they can be redefined in inner, nested blocks.

In the obsolete form, only identifiers appearing in the identifier list can be declared as formal parameters. Functions having variable-length argument lists should use the new prototype form. You are responsible for determining the number of arguments passed, and for retrieving additional arguments from the stack within the body of the function. (For information about macros that let you do this in a portable way, see your compiler guide.)

The compiler performs the usual arithmetic conversions independently on each formal parameter and on each actual argument, if necessary. After

C Language Reference

conversion, no formal parameter is shorter than an **int**, and no formal parameter has **float** type. This means, for example, that declaring a formal parameter as a **char** has the same effect as declaring it as an **int**.

If the **near**, **far**, and **huge** keywords are implemented, the compiler can also convert pointer arguments to the function. The conversions performed depend on the default size of pointers in the program and the presence or absence of a list of argument types for the function. For specific information about pointer conversions, see your compiler guide.

The converted type of each formal parameter determines the interpretation of the arguments that the function call places on the stack. A type mismatch between an actual argument and a formal parameter can cause the arguments on the stack to be misinterpreted. For example, if a 16-bit pointer is passed as an actual argument, then declared as a **long** formal parameter, the first 32 bits on the stack are interpreted as a **long** formal parameter. This error creates problems not only with the **long** formal parameter, but with any formal parameters that follow it. You can detect errors of this kind by declaring function prototypes for all functions.

Example

This example contains a structure-type declaration, a prototype of the function *match*, a call to *match*, and a prototype-style definition of *match*. Note that the same name, *student*, can be used without conflict both for the structure tag and for the structure variable name.

The *match* function is declared to have two arguments: the first, represented by *r*, is a pointer to the *struct student* type; the second, represented by *n*, is a pointer to a value of type **char**.

In the definition, the two formal parameters of the *match* function are declared in the formal parameter list in the parentheses following the function name, with the identifiers *r* and *n*. The parameter *r* is declared as a pointer to the *struct student* type; the parameter *n* is declared as a pointer to a **char** type value.

The function is called with two arguments, both members of the *student* structure. Because there is a prototype of *match*, the compiler performs type checking between the actual arguments and the types specified in the prototype and between the actual arguments and the formal parameters in the definition. Since the types match, no warnings or conversions are necessary.

Note that the array name given as the second argument in the call evaluates to a **char** pointer. The corresponding formal parameter is also declared as a **char** pointer and is used in subscripted expressions as

though it were an array identifier. Since an array identifier evaluates to a pointer expression, the effect of declaring the formal parameter as *char *n* is the same as declaring it *char n[]*.

Within the function, the local variable *i* is defined and used to monitor the current position in the array. The function returns the *id* structure member if the *name* member matches the array *n*; otherwise, it returns 0.

```

struct student {
    char name[20];
    int id;
    long class;
    struct student *nextstu;
} student;

main()
{
    /* declaration of function prototype: */

    int match ( struct student *r, char *n );
    .
    .
    if (match (student.nextstu, student.name) > 0) {
        .
        .
        .
    }
}

/* prototype style function definition */

match ( struct student *r, char *n )
{
    int i = 0;

    while ( r->name[i] == n[i] )
        if ( r->name[i++] == '\0' )
            return (r->id);

    return (0);
}

```

7.2.4 Function Body

A “function body” is a compound statement containing the statements that define what the function does. It can also contain declarations of variables used by these statements. (Section 6.3 discusses compound statements.)

C Language Reference

All variables declared in a function body have **auto** storage class unless otherwise specified. When the function is called, storage is created for the local variables and local initializations are performed. Execution control passes to the first statement in the compound statement and continues sequentially until a **return** statement is executed or the end of the function body is encountered. Control then returns to the point at which the function was called.

A **return** statement containing an expression must be executed if the function is to return a value. The return value of a function is undefined if no **return** statement is executed or if the **return** statement does not include an expression.

7.3 Function Prototypes (Declarations)

A “function prototype” declaration specifies the name, return type, and storage class of a function. It can also establish types and identifiers of some or all of the function’s arguments. The prototype has the same format as the function definition, except that it is terminated by a semicolon immediately following the closing parenthesis and therefore has no body. (See the “Declarations” chapter for a detailed description of the syntax of function declarations.)

You can declare a function implicitly, or you can use a “function prototype” (sometimes called a “forward declaration”) to declare it explicitly. A prototype is a declaration that precedes the function definition. In either case, the return type must agree with the return type specified in the function definition.

If a call to a function precedes its declaration or definition, a default prototype of the function is constructed, giving it **int** return type. The types and number of the actual arguments are used as the basis for declaring the formal parameters. Thus a call to the function is an implicit declaration, but the prototype generated may not adequately represent a subsequent definition of, or call to, the function.

A prototype establishes the attributes of a function so that calls to the function that precede its definition (or occur in other source files) can be checked for argument- and return-type mismatches. If you specify the **static** storage-class specifier in a prototype, you must also specify the **static** storage class in the function definition.

If you specify the **extern** storage-class specifier or omit the storage-class specifier entirely, the function has **extern** class. (For an explanation of the Microsoft extension that offers some latitude in function storage-class specification, see the *Note* in Section 7.2.1, “Storage Class.”)

Function prototypes have the following important uses:

- They establish the return type for functions that return any type other than **int**. If you call such a function before you declare or define it, the results are undefined. Although functions that return **int** values do not require prototypes, they are recommended.
- If the prototype contains a full list of parameter types, the types of the arguments occurring in a function call or definition can be checked. The prototype can include both the type of, and an identifier for, each expression that will be passed as an actual argument. However, such identifiers have scope only until the end of the declaration. The prototype can also reflect the fact that the number of arguments will be variable, or that there will be no arguments passed.

The parameter list in a prototype is a list of type names, separated by commas, corresponding to the actual arguments in the function call. The list is used for checking the correspondence of actual arguments in the function call with the formal parameters in the function definition. Without such a list, mismatches may not be revealed, so the compiler cannot generate diagnostic messages concerning them. (Type checking is further discussed in Section 7.4.1, “Actual Arguments.”)

- Prototypes are used to initialize pointers to functions before those functions are defined.

Example

In this example, the function *intadd* is implicitly declared to return an **int** value, since it is called before it is defined. The compiler creates a prototype using the information in the first call. Therefore, when the second call to *intadd* is encountered, the compiler sees the mismatch between *vall*, which is a **float**, and the **int** type of the first argument in its self-created prototype. The **float** is converted to an **int** and passed. Note that if the calls to *intadd* were reversed, the prototype created would expect a **float** as the first argument to *intadd*. When the second call is made, the variable *a* would be converted at the call, but when the value is actually passed to *intadd*, a diagnostic message would be issued because the **int** type specified in the definition does not match the **float** type in the compiler-created prototype.

The function *realadd* returns a **double** value instead of an **int** value. Therefore, the prototype of *realadd* in the *main* function is necessary because the *realadd* function is called before it is defined. Note that the definition of *realadd* matches the forward declaration by specifying the **double** return type.

C Language Reference

The forward declaration of *realadd* also establishes the types of its two arguments. The actual argument types match the types given in the declaration and also match the types of the formal parameters in the definition.

```
main()
{
    int a = 0, b = 1;
    float val1= 2.0, val2 = 3.0;

    /* function prototype: */

    double realadd(double x, double y);

    a = intadd (a, b);    /* first call to intadd */
    val1 = realadd(val1, val2);
    a = intadd(val1,b);  /* second call to intadd */
}

/* functions defined with formal parameters in header: */

intadd(int a, int b)
{
    return (a + b);
}

double realadd(double x, double y)
{
    return (x + y);
}
```

7.4 Function Calls

Syntax

expression([*expression-list*])

A “function call” is an expression that passes control and actual arguments (if any) to a function. In a function call, *expression* evaluates to a function address and *expression-list* is a list of expressions (separated by commas). The values of these latter expressions are the actual arguments passed to the function. If the function takes no arguments, *expression-list* can be empty.

When the function call is executed:

1. The expressions in *expression-list* are evaluated and converted using the usual arithmetic conversions. If a function prototype is available, the results of these conversions may be further converted consistent with the formal parameter declarations.

2. The expressions in *expression-list* are passed to the formal parameters of the called function. The first expression in the list always corresponds to the first formal parameter of the function, the second expression corresponds to the second formal parameter, and so on through the list. Since the called function uses copies of the actual arguments, any changes it makes to the arguments do not affect the values of variables from which the copies may have been made.
3. Execution control passes to the first statement in the function.
4. The execution of a **return** statement in the body of the function returns control and possibly a value to the calling function. If no **return** statement is executed, control returns to the caller after the last statement of the called function is executed. In such cases, the return value is undefined.

Note

The expressions in the function argument list can be evaluated in any order, so arguments whose values may be changed by side effects from another argument have undefined values. The sequence point defined by the function-call operator guarantees only that all side effects in the argument list are evaluated before control passes to the called function. See the “Expressions and Assignments” chapter for more information on sequence points.

The only requirement in a function call is that the expression before the parentheses must evaluate to a function address. This means that a function can be called through any function-pointer expression.

A function is called in much the same way it is declared. For instance, when you declare a function, you specify the name of the function, followed by a list of formal parameters in parentheses. Similarly, when a function is called, you need only specify the name of the function, followed by an argument list in parentheses. The indirection operator (*) is not required to call the function because the name of the function evaluates to the function address.



C Language Reference

The same principle applies when you call a function using a pointer. For example, suppose a function pointer has the following prototype:

```
int (*fpointer)(int num1, int num2);
```

The identifier *fpointer* is declared to point to a function taking two **int** arguments, represented by *num1* and *num2*, respectively, and returning an **int** value. A function call using *fpointer* might look like this:

```
(*fpointer)(3, 4)
```

The indirection operator (*) is used to obtain the address of the function to which *fpointer* points. The function address is then used to call the function. If a prototype of the pointer to the function precedes the call, the same checking will be performed as with any other function.

Example 1

In this example, the *realcomp* function is called in the statement *rp = realcomp(a, b)*; Two **double** arguments are passed to the function. The return value, a pointer to a **double** value, is assigned to *rp*.

```
double *realcomp(double value1, double value2);
double a, b, *rp;
.
.
.
rp = realcomp(a, b);
```

Example 2

In this example, the function call in *main* passes an integer variable and the address of the function *lift* to the function *work*:

```
work (count, lift);
```

Note that the function address is passed simply by giving the function identifier, since a function identifier evaluates to a pointer expression. To use a function identifier in this way, the function must be declared or defined before the identifier is used; otherwise, the identifier is not recognized. In this case, a prototype for *work* is given at the beginning of the *main* function.

The formal parameter *function* in *work* is declared to be a pointer to a function taking one **int** argument and returning a **long** value. The parentheses around the parameter name are required; without them, the declaration would specify a function returning a pointer to a **long** value.

The function *work* calls the selected function by using the following function call:

```
(*function) (i);
```

One argument, *i*, is passed to the called function.

```
main ()
{
    /* function prototypes: */

    long lift(int), step(int), drop(int);
    void work (int number, long (*function) (int i));

    int select, count;
    .
    .
    .
    select = 1;
    switch ( select ) {
        case 1: work(count, lift);
                break;

        case 2: work(count, step);
                break;

        case 3: work(count, drop);

        default:
                break;
    }
}

/* function definition with formal parameters in header: */
void work ( int number, long (*function)(int i) )
{
    int i;
    long j;

    for (i = j = 0; i < number; i++)
        j += (*function)(i);
}
```

7.4.1 Actual Arguments

An actual argument can be any value with fundamental, structure, union, or pointer type. Although you cannot pass arrays or functions as parameters, you can pass pointers to these items.

C Language Reference

All actual arguments are passed by value. A copy of the actual argument is assigned to the corresponding formal parameter. The function uses this copy without affecting the variable from which it was originally derived.

Pointers provide a way for a function to access a value by reference. Since a pointer to a variable holds the address of the variable, the function can use this address to access the value of the variable. Pointer arguments allow a function to access arrays and functions, even though arrays and functions cannot be passed as arguments.

The expressions in a function call are evaluated and converted as follows:

- The usual arithmetic conversions are performed on each actual argument in the function call. If a prototype is available, the resulting argument type is compared to the prototype's corresponding formal parameter. If they do not match, either a conversion is performed, or a diagnostic message is issued. The formal parameters also undergo the usual arithmetic conversions.
- If no prototype is available, the usual arithmetic conversions are performed on each actual argument before it is passed to the function. A prototype is created whose formal parameter types correspond to the types of the actual arguments after conversion.

If the **near**, **far**, and **huge** keywords are implemented, implementation-dependent conversions on pointer arguments can also be performed. For information about pointer conversions, see your compiler guide.

The number of expressions in the expression list must match the number of formal parameters, unless the function's prototype or definition explicitly specifies a variable number of arguments. In this case, the compiler checks as many arguments as there are type names in the list of formal parameters and converts them, if necessary, as described above.

If the prototype's formal parameter list contains only the keyword **void**, the compiler expects zero actual arguments in the function call and zero formal parameters in the definition. A diagnostic message is issued if it finds otherwise.

The type of each formal parameter also undergoes the usual arithmetic conversions. The converted type of each formal parameter determines how the arguments on the stack are interpreted; if the type of the formal parameter does not match the type of the actual argument, the data on the stack may be misinterpreted.

Note

Type mismatches between actual arguments and formal parameters can produce serious errors, particularly when the sizes are different. The compiler may not be able to detect these errors unless you declare complete prototypes of functions prior to calling them. In the absence of explicit prototypes, the compiler constructs prototypes from whatever information is available in the first reference to the function.

As an example of a serious error, consider a call to a function with an **int** argument. If the function is defined to take a **long**, and the definition occurs in a different module, the compiler-generated prototype will not match the definition, but the error will not be detected because the separate modules will compile without diagnostic messages.

Example

In this example, the *swap* function is declared in *main* to have two arguments, represented respectively by identifiers *num1* and *num2*, both of which are pointers to **int** values. The formal parameters *num1* and *num2* in the prototype-style definition are also declared as pointers to **int** type values. In the following function call the address of *x* is stored in *num1* and the address of *y* is stored in *num2*.

```
swap (&x, &y)
```

Now two names, or “aliases,” exist for the same location. References to **num1* and **num2* in *swap* are effectively references to *x* and *y* in *main*. The assignments within *swap* actually exchange the contents of *x* and *y*. Therefore, no **return** statement is necessary.

The compiler performs type checking on the arguments to *swap* because the prototype of *swap* includes argument types for each formal parameter. The identifiers within the parentheses of the prototype and definition can be the same or different. What is important is that the types of the actual arguments match those of the formal parameter lists in both the prototype and the eventual definition.

C Language Reference

```
main ()
{
    /* function prototype: */

    void swap (int *num1, int *num2);
    int x, y;
    .
    .
    .
    swap(&x, &y);
}

/* function definition: */

void swap (int *num1, int *num2)
{
    int t;

    t = *num1;
    *num1 = *num2;
    *num2 = t;
}
```

7.4.2 Calls with a Variable

To call a function with a variable number of arguments, simply specify any number of arguments in the function call. If there is a prototype declaration of the function, a variable number of arguments can be specified by placing a comma followed by three periods (,...), the “ellipsis notation,” at the end of the formal parameter list or list of argument types (see Section 4.5, “Function Declarations”). The function call must include one argument for each type name declared in the formal parameter list or the list of argument type.

Similarly, the formal parameter list (or identifier list, in the obsolete form) in the function definition can end with the ellipsis notation to indicate a variable number of arguments. For more information about the form of the formal parameter list, see Section 7.2, “Function Definitions.”

Note

To maintain compatibility with previous versions, a Microsoft extension to the ANSI C standard allows a comma without trailing periods (,) at the end of the list of formal parameters to indicate a variable number of arguments. For information on enabling and disabling extensions, see your compiler guide.

All the arguments specified in the function call are placed on the stack. The number of formal parameters declared for the function determines how many of the arguments are taken from the stack and assigned to the formal parameters. You are responsible for retrieving any additional arguments from the stack and for determining how many arguments are present. For information about macros that you can use to handle a variable number of arguments in a portable way, see your compiler guide.

7.4.3 Recursive Calls

Any function in a C program can be called recursively; that is, it can call itself. The C compiler allows any number of recursive calls to a function. Each time the function is called, new storage is allocated for the formal parameters and for the **auto** and **register** variables so that their values in previous, unfinished calls are not overwritten. Parameters are only directly accessible to the instance of the function in which they are created. Previous parameters are not directly accessible to ensuing instances of the function.

Note that variables declared with **static** storage do not require new storage with each recursive call. Their storage exists for the lifetime of the program. Each reference to such a variable accesses the same storage area.

Although the C compiler does not limit the number of times a function can be called recursively, the operating environment may impose a practical limit. Since each recursive call requires additional stack memory, too many recursive calls can cause a stack overflow.



Chapter 8

Preprocessor Directives and Pragmas

- 8.1 Introduction 8-1
- 8.2 Manifest Constants and Macros 8-2
 - 8.2.1 Preprocessor Operators 8-2
 - 8.2.2 The #define Directive 8-3
 - 8.2.3 The #undef Directive 8-9
- 8.3 Include Files 8-10
- 8.4 Conditional Compilation 8-12
 - 8.4.1 The #if, #elif, #else, and #endif Directives 8-12
 - 8.4.2 The #ifdef and #ifndef Directives 8-16
- 8.5 Line Control 8-17
- 8.6 Pragmas 8-18

8.1 Introduction

A “preprocessor directive” is an instruction to the C preprocessor. The C preprocessor is a text processor that manipulates the text of a source file as the first phase of compilation. Though the compiler ordinarily invokes the preprocessor in its first pass, the preprocessor can also be invoked separately to process text without compiling.

Preprocessor directives are typically used to make source programs easy to change and easy to compile in different execution environments. Directives in the source file tell the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text.

The C preprocessor recognizes the following directives:

#define	#if	#line
#elif	#ifdef	#undef
#else	#ifndef	
#endif	#include	

The number sign (#) must be the first non-white-space character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives include arguments or values. Any text that follows a directive (except an argument or value that is part of the directive) must be enclosed in comment delimiters (*/* */*).

Preprocessor directives can appear anywhere in a source file, but they apply only to the remainder of the source file in which they appear.

A “preprocessor operator” is an operator that is only recognized as an operator within the context of preprocessor directives. There are only three preprocessor-specific operators: the “stringizing” operator (#), the “token-pasting” (##) operator, and the **defined** operator. The first two are discussed in the context of the **#define** directive, later in this chapter. The **defined** operator is also discussed later in this chapter. “The **#if**, **#elif**, **#else**, and **#endif** Directives.”



C Language Reference

A “pragma” is a “pragmatic,” or practical, instruction to the C compiler. Pragas in C source files are typically used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. (Section 8.6 describes the syntax for pragmas). However, the compiler implementation defines the particular pragmas that are available and their meanings. For information about the use and effects of specific pragmas, see your compiler guide.

8.2 Manifest Constants and Macros

The **#define** directive is typically used to associate meaningful identifiers with constants, keywords, and commonly used statements or expressions. Identifiers that represent constants are called “manifest constants.” Identifiers that represent statements or expressions are called “macros.”

Once you have defined an identifier, you cannot redefine it to a different value without first removing the original definition. However, you can redefine the identifier with exactly the same definition. Thus, the same definition can appear more than once in a program.

The **#undef** directive removes the definition of an identifier. Once you have removed the definition, you can redefine the identifier to a different value. Sections 8.2.2 and 8.2.3 discuss the **#define** and **#undef** directives, respectively.

In practical terms there are two types of macros. “Object-like” macros take no arguments, while “function-like” macros can be defined to accept arguments so that they look and act like function calls. Because macros do not generate actual function calls, you can make programs faster by replacing function calls with macros. However, macros can create problems if you do not define and use them with care. You may have to use parentheses in macro definitions with arguments to preserve the proper precedence in an expression. Also, macros may not handle expressions with side effects correctly. For more information, see the examples in Section 8.2.2, “The # define Directive.”

8.2.1 Preprocessor Operators

There are three preprocessor-specific operators, one of which is represented by the number sign (**#**), one by a double number sign (**##**), and the third by the word **defined**. The “stringizing” operator (**#**) preceding a macro formal-parameter name in the body of a preprocessor macro causes the corresponding actual argument to be enclosed in string quotation marks. The “token-pasting” operator (**##**) allows tokens used as actual arguments to be concatenated to form other tokens. These two operators

are used in the context of the **#define** directive and are described in Sections 8.2.2.1 and 8.2.2.2.

Finally, the **defined** operator simplifies the writing of compound expressions in certain macro directives. It is used in conditional compilation, and is therefore discussed in Section 8.4.1, “The **#if**, **#elif**, **#else**, and **#endif** Directives.”

8.2.2 The **#define** Directive

Syntax

```
#define identifier substitution-text  
#define identifier(parameter-list) substitution-text
```

The **#define** directive substitutes *substitution-text* for all subsequent occurrences of *identifier* in the source file. The *identifier* is replaced only when it forms a token. (Tokens are described in the “Elements of C” chapter and in the “Syntax Summary.”) For instance, *identifier* is not replaced if it appears within a string or as part of a longer identifier.

If *parameter-list* appears after *identifier*, the **#define** directive replaces each occurrence of *identifier*(*parameter-list*) with a version of the *substitution-text* argument that has actual arguments substituted for formal parameters.

The *substitution-text* argument consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate *substitution-text* from *identifier* (or from the closing parenthesis following *parameter-list*). This white space is not considered part of the substituted text, nor is any white space following the last token of the text. Text longer than one line can be continued onto the next line by placing a backslash (\) before the new-line character.

The *substitution-text* argument can also be empty. Choosing this option removes occurrences of *identifier* from the source file. The *identifier* is still considered defined, however, and yields the value 1 when tested with the **#if** directive (discussed in Section 8.4.1).

The optional *parameter-list* consists of one or more formal parameter names separated by commas. Each name in the list must be unique, and the list must be enclosed in parentheses. No spaces can separate *identifier*



C Language Reference

and the opening parenthesis. The scope of a formal parameter name extends to the new line that ends *substitution-text*.

Formal parameter names appear in *substitution-text* to mark the places where actual values will be substituted. Each parameter name can appear more than once in *substitution-text*, and the names can appear in any order.

The actual arguments following an instance of *identifier* in the source file are matched to the corresponding formal parameters of *parameter-list*. Each formal parameter in *substitution-text* that is not preceded by a stringizing (#) or token-pasting (##) operator, or followed by a ## operator, is replaced by the corresponding actual argument. Any macros in the actual argument will be expanded before it replaces the formal parameter. (The # and ## operators are described in Sections 8.2.2.1 and 8.2.2.2.) The actual-argument list must have the same number of arguments as *parameter-list*.

If the name of the macro being defined occurs in *substitution-text* (even as a result of another macro expansion), it is not expanded.

Arguments with side effects sometimes cause macros to produce unexpected results. A given formal parameter may appear more than once in *substitution-text*. If that formal parameter is replaced by an expression with side effects, the expression, with its side effects, may be evaluated more than once (see Example 4 in Section 8.2.2.2, “Token-Pasting Operator”).

Stringizing Operator (#)

The number-sign or “stringizing” operator (#) is used only with macros that take arguments. If it precedes a formal parameter in the macro definition, the actual argument passed by the macro invocation is enclosed in quotation marks and treated as a string literal. The string literal then replaces each occurrence of a combination of the stringizing operator and formal parameter within the macro definition. White space preceding the first token of the actual argument and following the last token of the actual argument is ignored. Any white space between the tokens in the actual argument is reduced to a single white space in the resulting string literal. Thus, if a comment occurs between two tokens in the actual argument, it is reduced to a single white space. The resulting string literal is automatically concatenated with any adjacent string literals from which it is separated only by white space. Furthermore, if a character contained in the argument normally requires an escape sequence when used in a string literal—for example, the quotation-mark (") or backslash (\) characters—the necessary escape backslash is

Preprocessor Directives and Pragmas

automatically inserted before the character. The following example shows a macro definition that includes the stringizing operator and a main function that invokes the macro:

```
#define stringer(x) printf(#x "\n")

main()
{
    stringer (I will be in quotes in the printf function call);
    stringer ("I will be in quotes when printed to the screen");
    stringer (This: \" prints an escaped double quote mark);
}
```

Such invocations would be expanded during preprocessing, producing the following code:

```
printf("I will be in quotes in the printf function call" "\n");
printf("\"I will be in quotes when printed to the screen\""\n");
printf("This \" prints an escaped double quote mark");
```

When the program is run, screen output for each line would be as follows:

```
I will be in quotes in the printf function call
"I will be in quotes when printed to the screen"
This: \" prints an escaped double quote mark
```

Note

The Microsoft extension to the ANSI C standard that previously enabled expansion of macro formal arguments appearing in string literals and character constants is no longer supported. Code that relied on this extension should be rewritten using the stringizing (#) operator.

C Language Reference

Token-Pasting Operator (##)

The double-number-sign or “token-pasting” operator (##) is used in both object-like and function-like macros. It permits separate tokens to be joined into a single token, and therefore cannot be the first or last token in the macro definition.

If a formal parameter in a macro definition is preceded or followed by the token-pasting operator, the formal parameter is immediately replaced by the unexpanded actual argument. Macro expansion is not performed on the argument prior to replacement. Then, each occurrence of the token-pasting operator in *substitution-text* is removed, and the tokens preceding and following it are concatenated. The resulting token must be a valid token. If it is, the token is rescanned for possible replacement if it represents a macro name. Example 7 shows how tokens can be pasted together using the token-pasting operator.

Example 1

This example defines the identifier *WIDTH* as the integer constant 80 and defines *LENGTH* in terms of *WIDTH* and the integer constant 10. Each occurrence of *LENGTH* is replaced by (*WIDTH* + 10). In turn, each occurrence of *WIDTH* + 10 is replaced by the expression (*80* + 10).

```
#define WIDTH      80
#define LENGTH    (WIDTH + 10)
```

The parentheses around *WIDTH* + 10 are important because they control the interpretation in statements such as the following:

```
var = LENGTH * 20;
```

After the preprocessing stage the statement becomes

```
var = (80 + 10) * 20;
```

which evaluates to 1800. Without parentheses, the result is

```
var = 80 + 10 * 20;
```

which evaluates to 280.

Example 2

This example defines the identifier `The`. The definition is extended *FILEMESSAGE*. to a second line by using the convention of a backslash followed by a new-line character.

```
#define FILEMESSAGE "Attempt to create file \  
failed because of insufficient space"
```

Example 3

This example defines three identifiers, *REG1*, *REG2*, and *REG3*. *REG1* and *REG2* are defined as the keyword **register**. The definition of *REG3* is empty, so each occurrence of *REG3* is removed from the source file. These directives can be used to ensure that the program's most important variables (declared with *REG1* and *REG2*) are given **register** storage. (For an expanded version of this example, see the discussion of the **#if** directive in Section 8.4.1.)

```
#define REG1      register  
#define REG2      register  
#define REG3
```

Example 4

This example defines a macro named *MAX*. Each occurrence of the identifier *MAX* after the definition in the source file is replaced by the expression

```
((x) > (y)) ? (x) : (y)
```

where actual values replace the parameters *x* and *y*. For example, the occurrence

```
MAX(1, 2)
```

is replaced by

```
((1) > (2)) ? (1) : (2)
```

and the occurrence

```
MAX(i, s[i])
```

is replaced by

```
((i) > (s[i])) ? (i) : (s[i])
```

```
#define MAX(x,y)    ((x) > (y)) ? (x) : (y)
```



C Language Reference

Because this macro is easier to read than the corresponding expression, the source program is easier to understand.

Note that arguments with side effects may cause this macro to produce unexpected results. For example, the occurrence $MAX(i, s[i++])$ is replaced by $((i) > (s[i++])) ? (i) : (s[i++])$. The expression $(s[i++])$ may be evaluated twice, so by the time the ternary expression has been fully evaluated, i will have been incremented either once or twice, depending on the result of the comparison. **Example 5**

This example defines the macro *MULT*. Once the macro is defined, an occurrence such as $MULT(3, 5)$ is replaced by $(3) * (5)$. The parentheses around the parameters are important because they control the interpretation when complex expressions form the arguments to the macro. For instance, the occurrence $MULT(3 + 4, 5 + 6)$ is replaced by $(3 + 4) * (5 + 6)$, which evaluates to 77. Without the parentheses, the result would be $3 + 4 * 5 + 6$. This result evaluates to 29 because the multiplication operator (*) has higher precedence than the addition operator (+).

```
#define MULT(a,b)    ((a) * (b))
```

Example 6

This example defines two macros, one an object-like macro that expands to the string literal *Hello,World!*, and the other a function-like macro called *show*, which takes one argument. However, the definition of the second macro includes the stringizing operator (#) immediately preceding the formal parameter x . When an argument is passed to the *show* macro, the formal parameter is replaced by the actual argument enclosed in double quotation marks, thus “stringizing” it.

```
#define GREETING Hello, World!
#define show(x) printf(#x)

main()
{
    show( x + z );
    printf("\n");
    show(n /* some comment */ + p);
    printf("\n");
    show(GREETING); /* GREETING is not expanded; */
    printf("\n"); /* it is stringized instead */
    show{'\x'};
}
```

As the preprocessor progresses through the source file, the references to *show* are expanded as follows:

show(x + z); produces *printf("x + z");*

show(n / comment */ + p);* produces *printf("n + p");*

show(GREETING); produces *printf("GREETING");*

and finally, *show('\x');* produces *printf("\x");*

When the program is run, the screen output would be:

```
x + z
n + p
GREETING
'\x'
```

Example 7

This example illustrates use of both the “stringizing” and “token-pasting” operators in specifying program output.

```
#define paster(n) printf("token" #n " = %d", token##n)
```

If *token9* is declared, and the macro is called with a numeric argument like:

```
paster(9) ;
```

the macro yields:

```
printf("token" "9" " = %d", token9) ;
```

which becomes

```
printf("token9 = %d", token9) ;
```

8.2.3 The #undef Directive

Syntax

```
#undef identifier
```

The **#undef** directive removes the current definition of *identifier*. Consequently, subsequent occurrences of *identifier* are ignored by the preprocessor. To remove a macro definition using **#undef**, give only the macro *identifier*; do not give a parameter list.



C Language Reference

You can also apply the **#undef** directive to an identifier that has no previous definition. This ensures that the identifier is undefined.

The **#undef** directive is typically paired with a **#define** directive to create a region in a source program in which an identifier has a special meaning. For example, a specific function of the source program can use manifest constants to define environment-specific values that do not affect the rest of the program. The **#undef** directive also works with the **#if** directive (see Section 8.4.1) to control conditional compilation of the source program.

Example

In this example, the **#undef** directive removes definitions of a manifest constant and a macro. Note that only the identifier of the macro is given.

```
#define WIDTH          80
#define ADD(X,Y)      (X) + (Y)
.
.
.
#undef WIDTH
#undef ADD
```

8.3 Include Files

Syntax

```
#include "path-spec"
#include <path-spec>
```

The **#include** directive adds the contents of a given “include file” to another file. You can organize constant and macro definitions into include files and then use **#include** directives to add these definitions to any source file. Include files are also useful for incorporating declarations of external variables and complex data types. You only need to define and name the types once in an include file created for that purpose.

The **#include** directive tells the preprocessor to treat the contents of the named file as if they appeared in the source program at the point where the directive appears. The new text can also contain preprocessor directives. The preprocessor carries out directives in the new text, then continues processing the original text of the source file.

The *path-spec* is a file name optionally preceded by a directory specification. It must name an existing file. The syntax of the file specification depends on the operating system the program is compiled on.

The preprocessor uses the concept of a “standard” directory or directories to search for include files. The location of the standard directories for include files depends on the implementation and the operating system. For a definition of the standard directories, see your compiler guide.

The preprocessor stops searching as soon as it finds a file with the given name. If you specify a complete, unambiguous path specification for the include file, between two sets of double quotation marks (“ ”), the preprocessor searches only that path specification and ignores the standard directories.

If the *path-spec* enclosed in double quotation marks is an incomplete path specification, the preprocessor first searches the “parent” file’s directory. A parent file is the file containing the **#include** directive. For example, if you include a file named *file2* within a file named *file1*, *file1* is the parent file.

Include files can be “nested,” that is, an **#include** directive can appear in a file named by another **#include** directive. For example, *file2*, above, could include *file3*. In this case, *file1* would still be the parent of *file2*, but would be the “grandparent” of *file3*.

When include files are nested, directory searching begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source currently being processed. If the file is not found, the search moves to directories specified on the compiler command line. Finally, the standard directories are searched.

If the file specification is enclosed in angle brackets, the preprocessor does not search the current working directory. It begins by searching for the file in the directories specified on the compiler command line, then in the standard directories.

Nesting of include files can continue up to 10 levels. Once the nested **#include** is processed, the preprocessor continues to insert the enclosing include file into the original source file.

Example 1

This example adds the contents of the file named *stdio.h* to the source program. The angle brackets cause the preprocessor to search the standard



C Language Reference

directories for *stdio.h*, after searching directories specified on the command line.

```
#include <stdio.h>
```

Example 2

This example adds the contents of the file specified by *defs.h* to the source program. The double quotation marks mean that the preprocessor searches the directory containing the “parent” source file first.

```
#include "defs.h"
```

8.4 Conditional Compilation

This section describes the syntax and use of directives that control “conditional compilation.” These directives let you suppress compilation of parts of a source file by testing a constant expression or identifier to determine which text blocks will be passed on to the compiler and which text blocks will be removed from the source file during preprocessing.

8.4.1 The `#if`, `#elif`, `#else`, and `#endif` Directives

Syntax

```
#if restricted-constant-expression  
  [ text-block ]  
[ #elif restricted-constant-expression  
  text-block ]  
[ #elif restricted-constant-expression  
  text-block ]  
.  
.  
.  
[ #else  
  text-block ]  
#endif
```

The `#if` directive, together with the `#elif`, `#else`, and `#endif` directives, controls compilation of portions of a source file. Each `#if` directive in a source file must be matched by a closing `#endif` directive. Any number of `#elif` directives can appear between the `#if` and `#endif` directives, but at most one `#else` directive is allowed. The `#else` directive, if present, must be the last directive before `#endif`.

The preprocessor selects one of the given occurrences of *text-block* for further processing. A block specified in *text-block* can be any sequence of text. It can occupy more than one line. Usually *text-block* is program text that has meaning to the compiler or the preprocessor.

The preprocessor processes the selected *text-block* and passes it to the compiler. If *text-block* contains preprocessor directives, the preprocessor carries out those directives.

Any text blocks not selected by the preprocessor are removed from the file during preprocessing. Thus, these text blocks are not compiled.

The preprocessor selects a single *text-block* by evaluating the restricted constant expression following each **#if** or **#elif** directive until it finds a true (nonzero) restricted constant expression. It selects all text (including other preprocessor directives beginning with **#**) up to its associated **#elif**, **#else**, or **#endif**.

If all occurrences of *restricted-constant-expression* are false, or if no **#elif** directives appear, the preprocessor selects the text block after the **#else** clause. If the **#else** clause is omitted, and all instances of *restricted-constant-expression* in the **#if** block are false, no text block is selected.

Each *restricted-constant-expression* follows the rules for restricted constant expressions discussed in Section 5.2.10. Such expressions cannot contain **sizeof** expressions, type casts, or enumeration constants. However, they can contain the preprocessor operator **defined** in special constant expressions, as shown by the following syntax:

defined(*identifier*)

This constant expression is considered true (nonzero) if the *identifier* is currently defined; otherwise, the condition is false (0). An identifier defined as empty text is considered defined.

The **#if**, **#elif**, **#else**, and **#endif** directives can nest in the text portions of other **#if** directives. Each nested **#else**, **#elif**, or **#endif** directive belongs to the closest preceding **#if** directive.

C Language Reference

Example 1

In this example, the `#if` and `#endif` directives control compilation of one of three function calls. The function call to `credit` is compiled if the identifier `CREDIT` is defined. If the identifier `DEBIT` is defined, the function call to `debit` is compiled. If neither identifier is defined, the call to `printerror` is compiled. Note that `CREDIT` and `credit` are distinct identifiers in C because their cases are different.

```
#if defined(CREDIT)
    credit();
#elif defined(DEBIT)
    debit();
#else
    printerror();
#endif
```

Example 2

Examples 2 and 3 assume a previously defined manifest constant named `DLEVEL`.

Example 2 shows two sets of nested `#if`, `#else`, and `#endif` directives. The first set of directives is processed only if `DLEVEL > 5` is true. Otherwise, the second set is processed.

```
#if DLEVEL > 5
    #define SIGNAL 1
    #if STACKUSE == 1
        #define STACK 200
    #else
        #define STACK 100
    #endif
#else
    #define SIGNAL 0
    #if STACKUSE == 1
        #define STACK 100
    #else
        #define STACK 50
    #endif
#endif
```

Example 3

In Example 3, **#elif** and **#else** directives are used to make one of four choices, based on the value of *DLEVEL*. The manifest constant *STACK* is set to 0, 100, or 200, depending on the definition of *DLEVEL*. If *DLEVEL* is greater than 5, *display(debugptr)*; is compiled and *STACK* is not defined.

```
#if DLEVEL == 0
    #define STACK 0
#elif DLEVEL == 1
    #define STACK 100
#elif DLEVEL > 5
    display( debugptr );
#else
    #define STACK 200
#endif
```

Example 4

Example 4 uses preprocessor directives to control the meaning of **register** declarations in a portable source file. The compiler assigns register storage to variables in the order in which the **register** declarations appear in the source file. If a program contains more **register** declarations than the machine allows, the compiler honors earlier declarations over later ones. The program may be less efficient if the variables declared later are more heavily used.

```
#define REG1    register
#define REG2    register

#if defined(M_86)
    #define REG3
    #define REG4
    #define REG5
#else
    #define REG3    register
    #if defined(M_68000)
        #define REG4    register
        #define REG5    register
    #else
        #define REG4    register
        #define REG5
    #endif
#endif
#endif
```



C Language Reference

The definitions listed in Example 4 can be used to give priority to the most important register declarations. *REG1* and *REG2* are defined as the **register** keyword to declare **register** storage for the two most important variables in the program. For example, in the following fragment, *b* and *c* have higher priority than *a* or *d*:

```
func(a)

REG3 int a;

{
    REG1 int b;
    REG2 int c;
    REG4 int d;
    .
    .
    .
}
```

When *M_86* is defined, the preprocessor removes the *REG3* identifier from the file by replacing it with empty text. This prevents *a* from receiving **register** storage at the expense of *b* and *c*. When *M_68000* is defined, all four variables are declared to have **register** storage. When neither *M_86* nor *M_68000* is defined, *a*, *b*, and *c* are declared with **register** storage.

8.4.2 The #ifdef and #ifndef Directives

Syntax

```
#ifdef identifier
#ifndef identifier
```

The **#ifdef** and **#ifndef** directives perform the same task as the **#if** directive used with **defined(identifier)**. You can use the **#ifdef** and **#ifndef** directives anywhere **#if** can be used. These directives are provided only for compatibility with previous versions of the language. The **defined(identifier)** constant expression used with the **#if** directive is preferred.

When the preprocessor encounters an **#ifdef** directive, it checks to see whether the *identifier* is currently defined. If so, the condition is true (nonzero); otherwise, the condition is false (0).

The **#ifndef** directive checks for the opposite of the condition checked by **#ifdef**. If the identifier has not been defined (or its definition has been removed with **#undef**), the condition is true (nonzero). Otherwise, the condition is false (0).

8.5 Line Control

Syntax

```
#line constant ["filename"]
```

The **#line** directive tells the preprocessor to change the compiler's internally stored line number and file name to a given line number and file name. The compiler uses the line number and file name to refer to errors that it finds during compilation. The line number normally refers to the current input line, and the file name refers to the current input file. The line number is incremented after each line is processed.

If you change the line number and file name, the compiler ignores the previous values and continues processing with the new values. The **#line** directive is typically used by program generators to cause error messages to refer to the original source file instead of to the generated program.

The *constant* value in the **#line** directive can be any integer constant. The *filename* can be any combination of characters and must be enclosed in double quotation marks (""). If *filename* is omitted, the previous file name remains unchanged.

The current line number and file name are always available through the predefined identifiers **__LINE__** and **__FILE__**. You can use the **__LINE__** and **__FILE__** identifiers to insert self-descriptive error messages into the program text.

The **__FILE__** identifier expands to a string whose contents are the file name, surrounded by double quotation marks ("").

C Language Reference

Example 1

In this example, the internally stored line number is set to 151 and the file name is changed to *copy.c*.

```
#line 151 "copy.c"
```

Example 2

In this example, the macro *ASSERT* uses the predefined identifiers `__LINE__` and `__FILE__` to print an error message about the source file if a given “assertion” is not true.

```
#define ASSERT(cond)          if(!cond)\
{printf("assertion error line %d, file(%s)\n", \
__LINE__, __FILE__);} else
```

8.6 Pragmas

Syntax

#pragma *character-sequence*

A **#pragma** is an implementation-de-fined instruction to the compiler. The *character-sequence* is a series of characters that gives a specific compiler instruction and arguments, if any. The number sign (#) must be the first non-white-space character on the line containing the pragma; white-space characters can separate the number sign and the word **pragma**.

See your compiler guide for information about the pragmas available in your compiler implementation.

Appendix A

Differences Between K&R C and Microsoft C

A.1 Introduction A-1

A.1 Introduction

This appendix summarizes differences between Microsoft C and the description of the C language found in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, published in 1978 by Prentice-Hall, Inc. The following is a list of the differences with cross-references to the corresponding section numbers in *The C Programming Language*:



Section Number in Kernighan and Ritchie	Microsoft C
---	-------------

- | | |
|-------|--|
| 2.2 | Identifiers (including those used in preprocessor directives) are significant to 31 characters. External identifiers are also significant to 31 characters. |
| 2.3 | The identifiers asm and entry are no longer keywords. New keywords are const , volatile , enum , signed , and void . (The volatile keyword is implemented syntactically, but not semantically.) The identifiers cdecl , far , fortran , huge , near , and pascal may be keywords, depending on whether the corresponding options are enabled when a program is compiled (see your system documentation). |
| 2.4.1 | As a result of the method used to assign types to hexadecimal and octal constants, these constants always act like unsigned integers in type conversions. |
| 2.4.3 | Hexadecimal bit patterns consisting of a backslash (\), the letter x , and up to three hexadecimal digits are permitted as character constants (for example, <code>\x012</code>).

Microsoft C defines three additional escape sequences: <code>\v</code> represents a vertical tab (VT), <code>\"</code> represents the double-quotation-mark character, and <code>\a</code> represents the bell (also called alert).

Character constants always have type int , with the result that they are sign extended in type conversions.

Adjacent quoted string literals are concatenated and treated as a single null-terminated string. |
| 2.6 | The short type is always 16 bits long, and the long type is 32 bits long. The size of an int is machine dependent. On |

C Language Reference

8086/8088, 80186, and 80286 processors an **int** is 16 bits long, and on 80386 and 68000 processors it is 32 bits long.

- 4 The **char** type is signed by default, with the result that a **char** value is sign extended in type conversions. (In some implementations, the default for the **char** type can be changed to unsigned at compile time.)

Two additional unsigned types are supported: **unsigned char** and **unsigned long**.

The keyword **unsigned** or **signed** can be applied as an adjective to an integer type. When **unsigned** appears alone, it means **unsigned int**. Similarly, when **signed** appears alone, it means **int**. The additional floating type **long double** is supported, but the **long float** type is no longer recognized. References to **long float** should be recoded to **double**.

The type specifiers **const** and **volatile** can be used as modifiers for any fundamental, aggregate, or pointer type. The **const** keyword indicates that the object or pointer value will not be modified. The **volatile** keyword means the object may be changed by some process beyond the control of the currently running program. Both the syntax and semantics of **const** are implemented, but only the syntax of **volatile** is implemented.

Microsoft C offers an additional fundamental type: the **enum** (enumeration) type. Variables of **enum** type are treated as integers in all cases.

The keyword **void** has three different uses. As a function-return-type specifier, it indicates that the function will not return a value. In an otherwise empty formal-parameter list, **void** means that no arguments will be passed. In the construction **void ***, it indicates a pointer to an object of unspecified type.

- 6.4 If the **near**, **far**, and **huge** keywords are enabled, pointers of different sizes can be used in a program. Operations with pointers of different sizes can cause conversion of pointers; the path of the conversion is implementation defined.

Differences Between K&R C and Microsoft C

- 6.6 Arithmetic conversions carried out by the compiler are outlined in the “Expressions and Assignments” chapter. Although compatible with the Kernighan and Ritchie conversions, Microsoft C conversions are described in greater detail, including the specific path for each type of conversion.



In addition to the usual arithmetic conversions, conversions between pointers of different sizes can be routinely carried out when the **near**, **far**, and **huge** keywords are enabled. The path of the pointer conversions is implementation defined.

- 7.2 In connection with the **sizeof** operator, a byte is defined as an 8-bit quantity.
- 7.14 A structure can be assigned to another structure of the same type.
- 8.2 The keywords **enum**, **const**, **volatile**, and **void** are additional type specifiers. The **volatile** keyword is implemented syntactically, but not semantically. The keywords **signed** and **unsigned** can serve either as type specifiers or as adjectives modifying an integral type.

Therefore, the following additional combinations are acceptable:

signed char
signed short
signed short int
signed long
signed long int
unsigned char
unsigned short
unsigned short int
unsigned long
unsigned long int

The **long float** type is not recognized. The **long double** type is recognized and treated in all instances the same as **double**.

- 8.4 The **const** and **volatile** keywords can be used to modify any fundamental, aggregate, or pointer object. The order of the type specifiers is not significant.

C Language Reference

Optional formal-parameter lists or argument-type lists can be included in function declarations to notify the compiler of the number and types of arguments expected in a function call.

- 8.5 Bitfields can be declared to be any **signed** or **unsigned** integral type, except **enum**. However, in expressions, bitfields are always treated as **unsigned**.

The names of structure and union members are not required to be distinct from structure and union tags or from the names of other variables.

No relationship exists between the members of two different structure types.

- 8.6 Unions can be initialized by giving a value for the first member of the union.

- 9.7 The *expression* of a **switch** can be any integral expression, but the value of the expression is always converted to an **int** type. An **enum** type is permitted for *expression*. Each of the **case** constant expressions is cast to the type of *expression*.

- 10.1 New styles for function declarations and definitions, as specified in the Draft Proposed American National Standard — Programming Language C, are completely supported. This includes the function prototype declaration, the prototype-style definition with formal parameters declared in the header, and the default creation of prototypes from the first reference to a function (if no explicit prototype is provided). The old function declaration and definition forms are also supported.

The formal parameter list in a function definition or declaration can end with a comma followed by three periods (...) or just a comma (,) to indicate that the number of parameters is variable. The latter is supported only for compatibility with older versions of the compiler and should not be used in new code.

- 12 The number sign (#) introducing the preprocessor directive can be preceded by any combination of white-space characters. White space can also separate the number sign and the preprocessor keyword.

Differences Between K&R C and Microsoft C

In addition to preprocessor directives, the source file can contain pragmas. Pragmas, like directives, are introduced by a number sign as the first non-white-space character in a line. The action defined by a particular pragma is implementation dependent.



Three preprocessor-only operators are supported: the “stringizing” operator (#), the concatenation or “token-pasting” operator (##), and the **defined** operator.

- 12.3 The new combination **#if defined** (*identifier*) is intended to supplant the **#ifdef** and **#ifndef** directives. Use of the latter directives is discouraged.

The new directive **#elif** (else if) is designed for use in **#if** and **#if defined** blocks.

- 14.1 A structure or union can be assigned to another structure or union of the same type. Structures and unions can be passed by value to functions and returned by functions.

In expressions involving the structure-pointer operator (\rightarrow), the expression preceding the arrow must have the same type (or must be cast to the same type) as the structure to which the member on the right-hand side of the arrow belongs.

- 17 The listed anachronisms are not recognized.

Appendix B

Syntax Summary

- B.1 Tokens B-1
 - B.1.1 Keywords B-1
 - B.1.2 Identifiers B-2
 - B.1.3 Constants B-2
 - B.1.4 Strings B-5
 - B.1.5 Operators B-6
 - B.1.6 Separators B-6
- B.2 Expressions B-7
- B.3 Declarations B-9
- B.4 Statements B-13
- B.5 Definitions B-14
- B.6 Preprocessor Directives B-15
- B.7 Pragmas B-15

B.1 Tokens

keyword
identifier
constant
string
operator
separator

**B.1.1 Keywords**

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile [†]
do	if	static	while

[†] Semantics not yet implemented

The following identifiers may be keywords in some implementations. For information, see your compiler guide.

cdecl
far
fortran
huge
near
pascal

C Language Reference

B.1.2 Identifiers

identifier:

letter
underscore
identifier letter
identifier underscore
identifier digit

letter—one of the following:

a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

underscore:

—

digit—one of the following:

0 1 2 3 4 5 6 7 8 9

B.1.3 Constants

constant:

integer-constant
long-constant
floating-point-constant
char-constant
enum-constant

integer-constant:

0

decimal-constant

octal-constant

hexadecimal-constant

decimal-constant:

nonzero-digit

decimal-constant digit

nonzero-digit—one of the following:

1 2 3 4 5 6 7 8 9

octal-constant:

0*octal-digit*

octal-constant octal-digit

octal-digit—one of the following:

0 1 2 3 4 5 6 7

hexadecimal-constant:

0x*hexadecimal-digit*

0X*hexadecimal-digit*

hexadecimal-constant hexadecimal-digit

hexadecimal-digit—one of the following:

0 1 2 3 4 5 6 7 8 9

a b c d e f

A B C D E F

long-constant:

integer-constant l

integer-constant L



C Language Reference

floating-point-constant:

fractional-constant exponent

fractional-constant

digit-seq exponent

fractional-constant:

digit-seq . digit-seq

cc +

+cc

digit-seq .

digit-seq:

digit

digit-seq digit

exponent:

e sign digit-seq

E sign digit-seq

e digit-seq

E digit-seq

sign:

+

-

char-constant:

'char'

char:

rep-char

escape-sequence

rep-char:

Any single representable character except the single quotation-mark (`'`), backslash (`\`), or new-line character. Note that the single-quotation-mark character cannot be used alone in a character constant, and the double quotation-mark character cannot be used alone in a string literal.

escape-sequence—one of the following:

```

\'      \"      \\      \d      \dd      \ddd
\xd     \xdd    \xddd  \a      \b      \f
\n      \r      \t      \v

```

enum-constant:
identifier

B.1.4 Strings

string-literal:
""
"char-seq"

char-seq:
char
char-seq char



C Language Reference

B.1.5 Operators

operator—one of the following:

!	~	++	--	+
-	*	/	%	<<
>>	<	<=	>	>=
==	!=		&	^
&&		=	+=	-=
=	/=	%=	>>=	<<=
&=	^=	=	?:	,
[]	()	.	->	

B.1.6 Separators

separator—one of the following:

[]	()	{	}
*	,	:	=	;	#

B.2 Expressions*expression:*

identifier
constant
string
expression(expression-list)
expression(void)
expression[expression]
expression.identifier
expression->identifier
unary-expression
binary-expression
ternary-expression
assignment-expression
(expression)
(type-name)expression
constant-expression

expression-list:

expression
expression-list , expression

unary-expression:

unop expression
sizeof(expression)

unop—one of the following:

- ~ ! * &

lvalue:

identifier
expression[expression]
expression.expression
expression->expression
**expression*
(type-name)expression
(lvalue)



C Language Reference

type-name:

See Section B.3, “Declarations.”

binary-expression:

expression binop expression

binop—one of the following:

* / % + -
<< >> < > <=
>= == != & |
^ && || ,

ternary-expression:

expression ? expression : expression

assignment-expression:

lvalue++
lvalue--
++lvalue
--lvalue
lvalue assignment-op expression

assignment-op—one of the following:

= *= /= %= += -=
<<= >>= &= |= ^=

constant-expression:

identifier
constant
(type-name)constant-expression
unary-expression
binary-expression
ternary-expression
(constant-expression)

B.3 Declarations

declaration:

sc-specifier type-specifier-list declarator-list;
type-specifier-list declarator-list;
sc-specifier declarator-list;
typedef *type-specifier-list declarator-list;*



sc-specifier:

auto
extern
register
static

type-specifier:

char
double
long double
enum-specifier
float
int
long
short
struct-specifier
typedef-name
union-specifier
unsigned
signed
const
volatile
void

type-specifier-list:

type-specifier
type-specifier-list type-specifier

enum-specifier:

enum *tag* {*enum-list*}
enum {*enum-list*}
enum *tag*

C Language Reference

tag:
identifier

enum-list:
enumerator
enum-list , enumerator

enumerator:
identifier
identifier = constant-expression

struct-specifier:
struct *tag* {*member-declaration-list*}
struct {*member-declaration-list*}
struct *tag*

member-declaration-list:
member-declaration
member-declaration-list member-declaration

member-declaration:
type-specifier declarator-list;
type-specifier identifier : constant-expression;
type-specifier : constant-expression;

declarator-list:
declarator
declarator = initializer
declarator-list , declarator

declarator:
identifier
modifier-list identifier
declarator[]
declarator[constant-expression]
**declarator*
declarator(void)
declarator([formal-parameter-list])
(declarator)



modifier-list
modifier
modifier-list modifier

formal-parameter-list
formal-parameter
formal-parameter-list, formal-parameter
formal-parameter-list,...
formal-parameter-list,
formal-parameter
sc-spec type-spec declarator
sc-spec type-spec abstract-declarator

arg-type-list:
type-name
arg-type-list, type-name
arg-type-list,...
arg-type-list,

type-name:
type-specifier
type-specifier abstract-declarator

C Language Reference

abstract-declarator:

*modifier**
[]
(arg-type-list)
**abstract-declarator*
*abstract-declarator**
abstract-declarator[]
abstract-declarator[constant-expression]
[]abstract-declarator
[constant-expression]abstract-declarator
abstract-declarator(void)
abstract-declarator(formal-parameter-list)
abstract-declarator(arg-type-list)
(abstract-declarator)

initializer:

expression
{initializer-list}

initializer-list:

initializer
initializer-list, initializer

typedef-name:

identifier

union-specifier:

union *tag* {*member-declaration-list*}
union {*member-declaration-list*}
union *tag*

modifier:

cdecl
far
fortran
huge
near
pascal

modifier-list
modifier
modifier-list modifier

B.4 Statements



statement:
break;
case *constant-expression* : *statement*
compound-statement
continue;
default : *statement*
do *statement* **while**(*expression*);
expression;
for ([*expression*];[*expression*];[*expression*]) *statement*;
goto *identifier*;
identifier : *statement*
if (*expression*) *statement* [**else** *statement*]
;
return [*expression*];
switch (*expression*) *statement*
while (*expression*) *statement*

compound-statement:
 {[*declaration-list*][*statement-list*]}

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

C Language Reference

B.5 Definitions

definition:

function-definition
data-definition

function-definition:

*[sc-specifier] [type-specifier] declarator ([formal-parameter-list])
compound-statement*
*[sc-specifier] [type-specifier] declarator ([parameter-list])
[parameter-decs] compound-statement*

parameter-list:

fixed-parameter-list
variable-parameter-list

fixed-parameter-list:

identifier
parameter-list , identifier

variable-parameter-list:

fixed-parameter-list,...
fixed-parameter-list,

parameter-decs:

declaration
declaration-list declaration

data-definition:

declaration

B.6 Preprocessor Directives

directive:

```
#define identifier [[parameter-list]][token-seq]  
#elif restricted-constant-expression  
#else  
#endif  
#if restricted-constant-expression  
#ifdef identifier  
#ifndef identifier  
#include "string"  
#include <string>  
#line digit-seq  
#line digit-seq string  
#undef identifier
```

token-seq:

```
token  
token-seq token
```

restricted-constant-expression:

```
defined (identifier)  
Any constant-expression except sizeof expressions,  
casts, and enumeration constants
```

B.7 Pragmas

pragma:

```
#pragma char-seq
```



Replace this Page
with Tab Marked:

Index



Index

—> (arrow), in member-selection expressions 5-7
—> (member-selection operator) A-5, 5-7
> (greater-than operator) 5-24
>>= (greater-than-or-equal-to operator) 5-24
>> (right-shift operator) 5-23
<< (angle brackets) 8-10
\ (backslash character) 2-4, 2-5
{ } (braces) 4-47, 6-1, 6-3
[] (brackets) 1-6
[] (brackets)
 array declarators, used in 4-9, 4-25
[N] (brackets)
 subscript expressions, used in 5-4, 5-5
?: (conditional operator) 5-29a
?: (ternary operator) 5-13, 5-29
() (function modifier) 4-9
(token-pasting operator)
 described 8-2, 8-5
 differences from Kernighan and Ritchie 8-1
+ (addition operator) 5-21
& (address-of operator) 5-17
& (bitwise-AND operator) 5-26
- (arithmetic negation operator) 5-15
- (subtraction operator) 5-22
- (two's complement operator) 5-15
~ (bitwise-complement operator) 5-15
~ (one's complement operator) 5-15
^ (bitwise-exclusive-OR operator) 5-26
| (bitwise-inclusive-OR operator) 5-26
: (colon), with bitfield structure members 4-21
, (comma)
 argument-type lists, used in 4-33
 declarations, used in 4-16, 4-31
 function calls, used in 5-3, 7-14
 initialization, used in 4-47
 sequential-evaluation operator 5-28
-- (decrement operator) 5-32
/ (division operator) 5-20
... (ellipsis notation) 4-33
== (equality operator) 5-24
++ (increment operator) 5-32
* (indirection operator) 5-16
!= (inequality operator) 5-24
<< (left-shift operator) 5-23
< (less-than operator) 5-24
<= (less-than-or-equal-to operator) 5-24
&& (logical-AND operator) 5-27
! (logical-NOT operator) 5-15

|| (logical-OR operator) 5-27
. (member-selection operator) 5-7
* (multiplication operator) 5-20
(number sign) 8-1
() (parentheses)
 complex declarators, used in 4-9
 expressions, used in 5-9
 function calls, used in 5-3
 function declarators, used in 4-9, 4-31
 macros, used in 8-8
* (pointer modifier) 4-9, 4-27
(quotation marks)
 notational conventions 1-6
% (remainder operator) 5-20
+ (unary plus operator) 5-15
_ (underscore character) 2-14

A

Abstract declarators 4-54
Actual arguments *See* Arguments, actual
Addition operator (+) 5-21
Address-of operator (&) 5-17
Aggregate data-type category 4-7
Aggregate types
 array 4-25
 initialization 4-45, 4-47
 structure 4-20
 union 4-24
Anachronisms A-5
AND operators
 bitwise (&) 5-26
 logical (&&) 5-27
Angle brackets (<>) 8-10
ANSI standard
 enabling ANSI 1-1
 extensions 1-1
Apostrophe (') *See* Escape sequences
argc parameter 3-5
Argument type checking
 conversions 7-18
 default prototypes 7-13
 formal parameters 7-9
 function calls 7-18
 variable-length parameter list 4-34
Arguments
 actual

Index

Arguments (*continued*)

- actual (*continued*)
 - conversion 7-18
 - evaluation, order of 7-15
 - macros 8-4, 8-8
 - passing 7-17, 7-18
 - pointers 7-15, 7-18
 - side effects 7-15
 - type checking 7-18
 - variable number 7-20
 - command line 3-5
 - formal *See* Formal parameters
 - main function 3-5
 - variable number 4-33, 7-20
- ### Argument-type lists
- abstract declarator, used with 4-54
 - default prototype 7-13
 - described 4-32
 - pointer arguments, used with 4-34
 - variable length 4-33
 - void *, used with 4-34
 - void keyword, used with 4-34

argv parameter 3-5

Arithmetic conversions 5-13, A-3

Arithmetic data-type category 4-7

Arithmetic negation operator (-) 5-15

Array declarators ([]) 4-9, 4-25

Arrays

- declarations 4-9, 4-25
- elements 5-4
- identifiers 5-2
- initialization 4-45, 4-47, 4-50
- multidimensional 4-26, 5-5
- references to 5-2, 5-4
- storage 4-26, 5-6
- subscripts 5-4

asm keyword A-1

Assignments

- conversions 5-38
- defined 5-1
- expressions 5-9
- operators 5-30

Associativity

- modifiers 4-10
 - operators 5-35
- auto storage class 4-37, 4-41, 4-45

B

Backslash character (\) 2-4, 2-5

Backspace escape sequence (\b) 2-4

Bell character (\a) 2-4, A-1

Binary expressions 5-8

Binary operators, table 2-6, 5-13

Bitfields 4-21, 4-22, A-4

Bitwise-AND operator (&) 5-26

Bitwise-complement operator (~) 5-15

Bitwise-exclusive-OR operator (^) 5-26

Bitwise-inclusive-OR operator (|) 5-26

Blocks 3-6

Braces ({ })

- compound statement, used in 6-1, 6-3
- initialization, used in 4-47

Brackets

- array declarators, used in 4-9, 4-25
- subscript expressions, used in 5-4, 5-5

Brackets ([]) 1-6

Branch statements 6-9, 6-13

break statement 6-2

Bytes, size of A-3

C

C character set 2-1

Call by reference *See* Passing by reference

Call by value *See* Passing by value

Calls *See* function calls

Carriage-return escape sequence (\r) 2-4

case keyword 6-13

Case sensitivity 2-2, 2-14, 2-15

Casts *See* type casts

cdecl keyword 2-16, 4-13, A-1

char type

- conversion 5-39
- described 4-2
- differences from Kernighan and Ritchie A-2
- range of values 4-5
- storage 4-5

Character constants

- differences from Kernighan and Ritchie A-1

form 2-11

sign extension 2-12

type 2-12

Character sets 2-1

Characters

- backslash (\) 2-4, 2-5
- backspace escape sequence 2-4
- bell (\a) 2-4, A-1
- carriage-return escape sequence (\r) 2-4
- case 2-2, 2-14, 2-15
- continuation (\) 2-5

- Characters (*continued*)
 - differences from Kernighan and Ritchie A-1
 - digits 2-2
 - double-quotation-mark escape sequence (") 2-4
 - escape sequences 2-4
 - form-feed escape sequence (f) 2-4
 - hexadecimal escape sequences 2-4
 - horizontal tab escape sequence (t) 2-4
 - letters 2-2
 - new-line escape sequence (n) 2-4
 - octal escape sequences 2-4
 - punctuavion 2-3
 - single-quotation-mark escape sequence (') 2-4
 - special 2-3
 - underscore *_) 2-2
 - vertical-tab escape sequence (v) 2-4
 - white space 2-2, 2-4
- Colon (:), with bitfield structure members 4-21
- Comma (,)
 - argument-type lists, used in 4-33
 - declarations, used in 4-16, 4-31
 - function calls, used in 5-3, 7-14
 - initialization, used in 4-47
 - sequential-evaluation operator (,) 5-28
- Command-line arguments 3-5
- Comments 2-16
- Comparison operators *See* Also Relational operators
- Compilation, conditional 8-12, 8-16
- Complement operators (~) 5-15
- Complex declarators 4-10, 4-13
- Compound statements 6-3
- Compound-assignment operators 5-34
- Concatenation of string literals 2-13
- Concatenation operator, differences from Kernighan and Ritchie A-5
- Conditional compilation 8-12, 8-16
- Conditional operator (? :) 5-29
- Conditional statements 6-9, 6-13
- const
 - keyword A-1
 - pointer modifier, used as 4-27
 - type specifier 4-3
- Constant expressions
 - case 6-13
 - conversion 4-7
 - defined (identifier) 8-13
 - described 5-1
 - directives, used in 5-10, 8-13
 - form 5-10
 - initializers 5-10
- Constant expressions (*continued*)
 - restricted 5-10, 8-13
 - switch statement, used in 6-13
- Constants
 - character *See* Character constants
 - conversion 4-7
 - decimal integer 2-8, 2-9
 - described 2-8
 - enumeration 4-19
 - floating point 2-10, 2-11, 4-6
 - hexadecimal integer
 - conversion 2-10, 4-7
 - form 2-8
 - type 2-9
 - integer
 - differences from Kernighan and Ritchie A-1
 - form 2-8
 - long 2-10
 - negative 2-9
 - octal *See* Octal constants
 - type 2-9
 - manifest 8-2, 8-3, 8-9
 - string *See* String literals
 - summarized B-2
 - type 5-2
- Continuation character (\) 2-5
- continue statement 6-4
- Control, returning 6-11
- Conventions, notational 1-4
- Conversions
 - actual arguments 7-18
 - assignment 5-38
 - constant expressions 4-7
 - constants 4-7
 - enumeration types 5-46
 - floating types 5-43
 - formal parameters 7-9, 7-18
 - function call 5-47, 7-18
 - function prototypes 5-47
 - hexadecimal constants 4-7
 - implicit 5-45
 - octal constants 4-7
 - operator 5-46
 - pointer types 5-44
 - range of values, effects on 4-7
 - signed integral types 5-38, 5-45
 - structure types 5-46
 - type cast 5-46
 - union types 5-46
 - unsigned integral types 5-41, 5-45
 - usual arithmetic 5-13, A-3
 - void type 5-46

Index

D

Data type categories 4-7

Data types *See* Types

Decimal integer constants 2-8, 2-9

Declarations

defining 3-2

form 4-1

formal parameter names 4-31

formal parameters 7-7, 7-8

forward *See* Function declarations
(prototypes)

function *See* Function declarations
(prototypes)

pointer 4-9, 4-27, 7-13

referencing 3-2

storage allocation 3-2

summarized B-9

type 4-51

typedef 4-51, 4-52

variable

array 4-25

default storage class 4-40

described 3-1

enumeration 4-18

external 4-38

form 4-16

global 4-38

internal 4-38

local 4-41

multidimensional arrays 4-26

pointer 4-27

simple 4-17

structure 4-20

union 4-24

Declarators

abstract 4-54

array 4-9

complex 4-9, 4-10, 4-13

described 4-8

function 4-9

parentheses, enclosed in 4-9

pointer 4-9

special keywords, used with 4-13

Decrement operator (--) 5-32

default keyword 6-13

Default return type 4-31

Default storage class

function declarations 4-44

global variable declarations 4-40

local variable declarations 4-41

#define directive 8-3

defined (identifier) constant

defined (identifier) constant (*continued*)
expression 8-13

defined preprocessor operator 8-1,
8-2, A-5

Defining declaration 4-38

Definitions

function

described 3-2, 7-1, 7-3

full prototype form 7-3

obsolete form 7-4

storage class 7-4

summarized B-14

visibility 7-4

removing 8-9

storage allocation 3-2

variable

described 3-2, 4-38

storage class 4-38

summarized B-14

visibility 4-39, 4-41, 4-42

Differences from Kernighan
and Ritchie A-0

Digits 2-2

Dimensions *See* Multidimensional arrays

Directives

constant expressions, used in 5-10, 8-13

#define 8-3

described 3-1, 8-1

differences from Kernighan
and Ritchie A-4, A-5

#elif

described 8-12

differences from Kernighan
and Ritchie A-5

nesting 8-13

#else 8-12, 8-13

#endif 8-12, 8-13

#if 8-12, 8-13, A-5

#ifdef 8-16, A-5

#ifndef 8-16, A-5

#include 8-10

lifetime 3-3

#line 8-17

restricted constant expressions 5-10

summarized B-15

#undef 8-9

Division operator (/) 5-20

do statement

described 6-4

execution

continuation of 6-4

termination of 6-2

Double quotation mark (") *See*

Quotation marks

double type
 conversion 5-43
 described 4-2
 internal representation 4-6
 range of values 4-5
 storage 4-5
 Double-quotation-mark escape sequence
See Escape sequences

E

Elements 5-4, 5-5
 #elif directive
 described 8-12
 differences from Kernighan
 and Ritchie A-5
 nesting 8-13
 Ellipsis notation (...) 1-5
 #else directive 8-12, 8-13
 else keyword 6-9
 #endif directive 8-12, 8-13
 entry keyword A-1
 enum type specifier 4-18, A-1
 Enumeration constants 3-12, 4-19
 Enumeration expressions 5-2
 Enumeration set 4-18
 Enumeration types
 conversion 5-46
 declaration 4-18, 4-51
 described 4-2
 differences from Kernighan
 and Ritchie A-2
 identifiers 5-2
 range of values 4-5
 storage 4-5, 4-18
 tags
 defined 3-13
 naming class 3-13
 type declarations 4-51
 variable declarations 4-18
 Enumeration variables 4-16
 envp 3-5
 Equality operator (==) 5-24
 described 2-4
 differences from Kernighan
 and Ritchie A-1
 \ (single quotation mark) 2-4
 \a (bell) 2-4
 \b (backspace) 2-4
 \\ (backslash) 2-4
 \f (form feed) 2-4

Escape sequences (*continued*)
 \" (double quotation mark) 2-4
 \n (new line) 2-4
 \r (carriage return) 2-4
 \t (horizontal tab) 2-4
 \v (vertical tab) 2-4
 Evaluation
 order of 5-27, 5-36
 unary plus (+), forcing order with 5-15
 Execution *See* Program execution
 Exit from functions 6-11
 Exponents 2-10
 Expressions
 assignment 5-9
 binary 5-8
 case constant 6-13
 constant *See* Constant expressions
 described 5-1
 enumeration 5-2
 floating type 5-2
 function call 5-3
 grouping 5-35
 integral 5-2
 list 5-3
 lvalue 5-31
 member selection 5-7, A-5
 operators, used in 5-8
 order of evaluation 5-36
 parentheses, enclosed in 5-9
 pointer 5-2
 side effects 5-11
 statements 6-5
 string literal 5-3
 structure 5-2
 subscript 5-4, 5-5
 summarized B-7
 switch 6-13, A-4
 ternary 5-8
 type cast 5-10
 unary 5-8
 union 5-2
 Extensions to ANSI C standard 1-1
 extern storage class
 described 4-37
 function
 declarations 4-44
 definitions 7-4
 function declarations 7-12
 global variables 4-38
 local variables 4-41
 External declarations
 described 4-38
 function 4-44
 External level 3-2

Index

F

far keyword

- conversions 7-18
- described 4-13
- differences from Kernighan and Ritchie A-1
- listed 2-16

Fields *See* Bitfields

FILE identifier 8-17

Files

- inclusion 8-10
- name, changing 8-17
- nesting 8-11

float type

- conversion 5-43
- described 4-2
- internal representation 4-6
- range of values 4-5
- storage 4-5

Floating point

- constants
 - form 2-10
 - internal representation 4-6
 - negative 2-11
- data-type category 4-7
- expressions 5-2
- identifiers 5-2
- types
 - described 4-2
 - internal representation 4-6
- types, conversion of 5-43

for statement

- described 6-6
- execution continuation 6-4
- execution termination 6-2

Forcing evaluation order 5-15

Formal parameters

- conversion 7-9, 7-18
- declaration 7-8
- described 4-32, 7-7
- following function header 7-4
- identifiers 7-9
- list 7-3
- macro 8-3
- names 4-31
- naming class 3-12
- obsolete form 7-8
- storage class 7-9
- type checking 7-9, 7-18

Form-feed escape sequence (\f) 2-4

fortran keyword 2-16, 4-13, A-1

Forward declarations *See* Function

Forward declarations *See* Function
(*continued*)

declarations (prototypes)

Function

body 7-4, 7-11

calls

- argument type checking 7-18
- arguments, variable number of 7-20
- conversions 5-47, 7-18
- described 7-1
- expressions 5-3
- form 5-3, 7-14
- indirect 7-15
- operator, used as sequence point 5-12
- pointers, use of 7-15
- recursive 7-21

declarations (prototypes)

- arguments, variable number of 4-33
- arguments, without 4-34
- default return type 4-31
- default storage class 4-44
- described 3-1, 7-1, 7-12
- differences from Kernighan and Ritchie A-3
- implicit 7-12
- parameter list 4-34
- pointer 4-31
- pointer arguments 4-34
- return type 4-32, 7-12
- return value 7-12
- storage class 4-44, 7-12
- visibility 4-44, 7-12

definition

- full prototype form 7-3
- obsolete form 7-4

definitions *See* Definitions function

modifier () 4-9

names *See* Identifiers

pointers 7-13, 7-15

prototypes

- conversions 5-47
- defined 4-34, 7-1
- return type *See* Return type
- type *See* Return type

Function-like macros 8-2

Functions

- described 7-1
- exit from 6-11
- identifiers 5-2
- main 3-5
- naming class 3-12
- return value 6-11

G

- Global
 - level 3-2
 - lifetime 3-6, 4-37
 - variables
 - described 3-8
 - initialization 4-45
 - references to 4-42
 - visibility 3-7
- Global declarations
 - variable 4-38
- goto statement 6-8
- Greater-than operator (>) 5-24
- Greater-than-or-equal-to operator (>=) 5-24
- Grouping 5-35

H

- Hexadecimal
 - constants
 - conversion 2-10, 4-7
 - differences from Kernighan and Ritchie A-1
 - form 2-8
 - sign extension 2-10
 - type 2-9
 - escape sequences 2-4, A-1
- Horizontal-tab escape sequence (\t) 2-4
- huge keyword
 - conversion 7-18
 - described 4-13
 - differences from Kernighan and Ritchie A-1
 - listed 2-16

I

- Identifier lists 7-8
- Identifiers
 - array 5-2
 - characters allowed 2-14
 - differences from Kernighan and Ritchie A-1
 - enumeration 5-2
 - __FILE__ 8-17
 - floating type 5-2
 - Identifiers (*continued*)
 - formal parameters 7-9
 - function 5-2
 - integral 5-2
 - length 2-14
 - __LINE__ 8-17
 - modified 4-9
 - naming classes 3-11
 - pointer 5-2
 - structure 5-2
 - summarized B-2
 - union 5-2
 - #if directive 8-12, 8-13, A-5
 - if statement 6-9
 - #ifdef directive 8-16, A-5
 - #ifndef directive 8-16, A-5
 - #include directive 8-10
 - Include files 8-10, 8-11
 - Increment operator (++) 5-32
 - Indirection operator (*) 5-16
 - Inequality operator (!=) 5-24
 - Initialization
 - arrays 4-45, 4-47, 4-50
 - auto storage class 4-45
 - constant expressions 5-10
 - differences from Kernighan and Ritchie A-4
 - fundamental types 4-45
 - global variables 4-45
 - link time 4-40
 - pointers 4-45
 - register storage class 4-45
 - restrictions 4-45
 - static variables 4-45
 - string literals 4-50
 - structure variables 4-45, 4-47
 - union variables 4-45, 4-47
 - Insertion of files 8-10
 - int type
 - conversion 5-40
 - described 4-2
 - differences from Kernighan and Ritchie A-1
 - portability 4-6
 - range of values 4-5
 - storage 4-5
 - Integer constants
 - decimal 2-8, 2-9
 - differences from Kernighan and Ritchie A-1
 - hexadecimal 2-8, 2-9, 2-10
 - long 2-10
 - negative 2-9
 - octal 2-8, 2-9, 2-10

Index

Integral

- data-type category 4-7
- expressions 5-2
- identifiers 5-2
- types
 - conversion 5-38, 5-41, 5-45
 - described 4-2

Internal

- declarations 4-38
- representation 4-6, 4-7

Internal level 3-2

Italics 1-4

Iterative statements

- do 6-4
- for 6-6
- while 6-15

K

Keywords

- differences from Kernighan and Ritchie A-1, A-3
- listed 2-15, B-1
- notational conventions 1-4
- special 4-13, 4-28
- statements, used in 6-1
- system dependent 2-16

L

Labeled statements 6-8

Labels

- case 6-13
- default 6-13
- described 6-1
- form 6-8
- naming class 3-13

Left-shift operator (<<) 5-23

Less-than operator (<)

See Relational operators

Less-than-or-equal-to operator (<=)

See Relational operators

Letters 2-2

Lifetime

- described 3-6
- directives 3-3
- global 3-6, 4-37
- local 3-6, 4-37

Line control 8-17

#line directive 8-17

__LINE__ identifier 8-17

Lines, continuation 2-5

Linked lists 4-21

Local

- declarations 4-41
- level 3-2
- lifetime 3-6, 4-37
- variables 3-8, 7-12

Logical-AND operator (&&) 5-27

Logical-NOT operator (!) 5-15

Logical-OR operator (||) 5-27

long type

- conversion 5-39
- described 4-2
- differences from Kernighan and Ritchie A-1
- range of values 4-5
- storage 4-5

long-double type, conversion 5-44

long-float type 4-2

Loops

- do statement 6-4
- for statement 6-6
- while statement 6-15

Lvalue expressions 5-31

M

Macros

- actual arguments 8-4
- #define directive 8-3
- described 8-2
- empty definition 8-3
- example, with arguments 8-8
- example, with side effects 8-8
- function like 8-2
- object like 8-2
- side effects of arguments 8-4
- #undef, effect of 8-9

Main function 3-5

Manifest constants 8-2, 8-3, 8-9

Members

- bitfields 4-21
- naming class 3-13
- referring to 5-7
- structure 4-20
- union 4-24

Member-selection expressions 5-7, A-5

Member-selection operators

(->) and .) 5-7

Member-selection operators

Member-selection operators (*continued*)
 (-> and .) A-5

Modifiers

- array 4-9, 4-25
- associativity 4-10
- function 4-9
- pointer 4-9, 4-27
- precedence 4-10

Multidimensional arrays 4-26, 5-5

Multiplication operator (*) 5-20

N

Names *See* Identifiers

Naming classes 3-11, A-4

near keyword

- conversions 7-18
- described 4-13
- differences from Kernighan
and Ritchie A-1
- listed 2-16

Negation 5-15

Nested visibility 3-8

New-line escape sequence (\n) 2-4

Nongraphic escape sequences 2-4

NOT operator (!) *See* Logical-NOT operator

Notational conventions 1-4

Null statement 6-10

Number sign (#) 8-1

O

Object-like macros 8-2

Octal

constants

- conversion 2-10, 4-7
 - differences from Kernighan
and Ritchie A-1
 - form 2-8
 - sign extension 2-10
 - type 2-9
- escape sequences 2-4

One's complement operator (~) 5-15

Operands 5-1

Operators

- addition (+) 5-21
- address of (&) 5-17
- arithmetic negation (-) 5-15
- assignment

Operators (*continued*)

assignment (*continued*)

- compound 5-34
- listed 5-30
- simple (=) 5-33

associativity 5-35

binary

- described 5-13
- table 2-6

bitwise AND (&) 5-26

bitwise complement (~) 5-15

bitwise-exclusive OR (^) 5-26

bitwise-inclusive OR (|) 5-26

complement 5-15

compound assignment 5-34

conditional (? :) 5-29

conversions 5-46

decrement (--) 5-32

differences from Kernighan

and Ritchie A-5

division (/) 5-20

equality (==) 5-24

expressions, used in 5-8

increment (++) 5-32

indirection (*) 5-16

inequality (!=) 5-24

left-shift (<<) 5-23

listed 2-6, B-6

logical

described 5-27

evaluation, order of 5-27

logical AND (&&) 5-27

logical NOT (!) 5-15

logical OR (||) 5-27

multiplication (*) 5-20

one's complement (~) 5-15

precedence 5-35

preprocessor

differences from Kernighan

and Ritchie A-5

stringizing A-5

token pasting A-5

preprocessor specific, listed 8-2

relational (>, <, <=, >=) 5-24

remainder (%) 5-20

right shift (>>) 5-23

sequence points, used as 5-12

sequential evaluation (,) 5-28

shift (<< and >>) 5-23

simple assignment (=) 5-33

sizeof 5-18

subtraction (-) 5-22

ternary (?:) 5-13

ternary (?:) 5-29

Index

Operators (*continued*)

- two's complement (-) 5-15
- unary 2-6, 5-13

OR operators

- bitwise exclusive (^) 5-26
- bitwise inclusive (&) 5-26
- logical (||) 5-27

Overview 1-1

P

Parameter list 4-34

Parameters

- argc 3-5
- argv 3-5
- envp 3-5
- formal *See* Formal parameters
- macro 8-3
- main function 3-5

Parentheses

- complex declarators, used in 4-9
- expressions, used in 5-9
- function calls, used in 5-3
- function declarators, used in 4-9, 4-31
- macros, used in 8-8

pascal keyword 2-16, 4-13, A-1

Passing by

- reference 7-18
- value 7-14, 7-18

Pointer

- modifier (*) 4-9, 4-27
- void (void *) 4-27

Pointer data-type category 4-7

Pointers

- adding 5-22
- arithmetic 5-22
- comparisons 5-25
- const, modified by 4-27
- conversion 5-44
- declarations 4-9, 4-27, 7-13
- differences from Kernighan and Ritchie A-2
- expressions 5-2
- function calls through 7-15
- functions 7-13, 7-15
- identifiers 5-2
- implicit conversion 5-45
- initialization 4-45
- storage 4-28
- structure 4-27
- subtraction 5-22
- union 4-28

Pointers (*continued*)

- volatile, modified by 4-27

Portability 4-6

Pound sign (#) *See* Number sign

Pragmas

- described 3-1, 8-2
- differences from Kernighan and Ritchie A-5
- form 8-18

Precedence

- modifiers 4-10
- operators 5-35

Predefined identifiers 8-17

Preprocessor directives *See* Directives

Preprocessor operators

- described 8-1
- listed 8-2

Program execution 3-5

Program structure 3-1

Prototypes, function 4-34, 7-1

Punctuation characters 2-3

Q

" (quotation marks)

- #include directives, used in 8-10
- representation A-1

\\" (quotation marks)

- representation 2-4

Quotation marks (N)

- notational conventions 1-6

Quotation marks (")

- #include directives, used in 8-10

Quotation marks (")

- representation A-1, 2-4

R

Recursion 7-21

Reference, passing by 7-18

References to global

- variables 4-38, 4-39, 4-42

Referencing declarations 4-38

register storage class

- described 4-42
- initialization 4-45
- lifetime 4-37
- local variables 4-41

Relational operators (>,<,<=,>=) 5-24
 Representable character set 2-1
 Representation, internal 4-6, 4-7
 Reserved words *See* Keywords
 Restricted constant expressions 5-10, 8-13
 return statement 6-11
 Return type
 declaration 7-12
 default 4-31
 described 4-32, 7-5
 implicit 7-12
 Return value 6-11, 7-12
 Right-shift operator (>>) 5-23

S

Scalar data-type category 4-7
 Selection statements 6-9, 6-13
 Sensitivity, case 2-2
 Separators B-6
 Sequence points
 described 5-1, 5-12
 listed 5-12
 operators, other than 5-12
 Sequential-evaluation operator (,) 5-28
 Shift operators (<< and >>) 5-23
 short type
 conversion 5-39
 described 4-2
 differences from Kernighan
 and Ritchie A-1
 range of values 4-5
 storage 4-5
 Side effects
 expressions 5-1, 5-11
 macros, used with 8-4, 8-8
 sequence points, used with 5-12
 Sign extension 2-10, 2-12
 signed
 char type 4-2, A-3
 int type 4-2
 keyword 4-3, A-2
 long int type A-3
 long type 4-2, A-3
 short int type 4-2, A-3
 short type 4-2, A-3
 type 4-2, A-2
 Simple variable declarations 4-17
 Simple-assignment operator (=) 5-33
 Single-quotation-mark escape sequence (?)
 See Escape sequences
 sizeof operator 5-18

Source files 3-3
 Special characters 2-3
 Special keywords
 conversions 7-18
 declarators, used with 4-28
 differences from Kernighan
 and Ritchie A-1
 Standard directories 8-11
 Statement labels
 described 6-1
 form 6-8
 naming class 3-13
 Statements
 body 6-1
 break 6-2
 compound 6-3
 continue 6-4
 do 6-4
 expression 6-5
 for 6-6
 form 6-1
 goto 6-8
 if 6-9
 keywords 6-1
 labeled 6-1, 6-8
 listed 6-1
 null 6-10
 return 6-11
 summarized B-13
 switch 6-13
 while 6-15
 static storage class
 described 4-37
 function
 declarations 4-44, 7-12
 definitions 7-4
 global variables 4-38
 initialization 4-45
 local variables 4-41
 Storage
 bitfields 4-22
 global 4-37
 local 4-37
 type
 char 4-5
 double 4-5
 float 4-5
 int 4-5
 long 4-5
 unsigned char 4-5
 unsigned int 4-5
 unsigned long 4-5
 void 4-5
 types

Index

- Storage (*continued*)
 - types (*continued*)
 - array 4-26, 5-6
 - enumeration 4-5, 4-18
 - pointer 4-28
 - structure 4-22
 - union 4-24
 - Storage allocation for variables 3-2
 - Storage classes
 - described 4-37
 - formal parameters 7-9
 - function
 - declarations 7-12
 - function declarations 4-44
 - function definitions 7-4
 - global variable declarations 4-40
 - local variable declarations 4-41
 - Storage-class specifiers
 - auto 4-37, 4-41
 - extern *See* extern storage class
 - listed 4-37
 - register 4-37, 4-41
 - static *See* static storage class
 - String concatenation 2-13
 - String literals
 - concatenation 2-13
 - form 2-12, 5-3
 - initializers 4-50
 - length 2-14, 5-3
 - storage 2-14
 - type 2-14
 - Stringizing preprocessor operator (#)
 - described 8-2, 8-4
 - differences from Kernighan and Ritchie A-5
 - Strings *See* String literals
 - struct type-specifier 4-20
 - Structures
 - conversion 5-46
 - declaration 4-20, 4-51
 - differences from Kernighan and Ritchie A-3, A-4, A-5
 - expressions 5-2
 - identifiers 5-2
 - initialization 4-45, 4-47
 - members *See* Members
 - pointers to 4-28
 - storage 4-22
 - tags
 - naming class 3-13
 - type declarations 4-51
 - variable declarations 4-20
 - Subscript expressions 5-4, 5-5
 - Subtraction operator (-) 5-22
 - switch statement
 - constant expressions, used in 6-13
 - described 6-13
 - differences from Kernighan and Ritchie A-4
 - termination of execution 6-2
 - Symbolic constants *See* Manifest constants
 - Syntax
 - conventions *See* Notational conventions
 - summary B-1
 - System-dependent keywords 2-16
- ## T
- Tab escape sequences 2-4
 - Tags
 - enumeration 4-18, 4-51
 - naming class 3-13
 - structure 4-20, 4-51
 - union 4-51
 - Ternary expressions 5-8
 - Ternary operator (?:) 5-13
 - Ternary operator (? :) 5-29
 - Token-pasting preprocessor operator (##)
 - described 8-2, 8-5
 - differences from Kernighan and Ritchie A-5
 - Tokens 2-6, 2-17, B-1
 - Transfer statements
 - break 6-2
 - continue 6-4
 - goto 6-8
 - labeled statements 6-8
 - Two's complement operator (-) 5-15
 - Type
 - checking *See* Arguments
 - declarations 4-51
 - modifiers
 - differences from Kernighan and Ritchie A-3
 - names
 - argument-type lists, used in 4-34
 - described 4-53
 - sizeof, used with 5-18
 - void 7-18
 - specifiers
 - abbreviations 4-3
 - const 4-3
 - differences from Kernighan and Ritchie A-1
 - enum 4-2, 4-18
 - fundamental types 4-2

Type (*continued*)
 specifiers (*continued*)
 struct 4-20
 union 4-24
 volatile 4-3
 Type-cast conversions 5-46
 Type-cast expressions
 constraints 5-10
 defined 5-10
 void, to and from 5-10
 typedef
 declarations 4-51, 4-52
 types 3-13, 4-52
 Types
 array
 declaration 4-9, 4-25
 initialization 4-45, 4-47, 4-50
 multidimensional 4-26
 storage 4-26, 5-6
 char *See* char type
 const
 described 4-3
 pointers, used with 4-27
 conversions *See* Conversions
 differences from Kernighan
 and Ritchie A-1, A-2
 double 4-2, 4-5, 4-6
 enumeration *See* Enumeration types
 float *See* float type
 floating point
 described 4-2
 internal representation 4-6
 function *See* Return type
 fundamental
 declaration 4-17
 described 4-2
 differences from Kernighan
 and Ritchie A-2
 initialization 4-45
 listed 4-2
 range of values 4-5
 storage 4-5
 int *See* int type
 integral
 conversion 5-38, 5-41, 5-45
 described 4-2
 long double, differences from
 Kernighan and Ritchie A-2
 long float A-2
 long *See* long type
 pointer
 conversion 5-44
 declaration 4-9, 4-27
 implicit conversion 5-45

Types (*continued*)
 pointer (*continued*)
 initialization 4-45
 storage 4-28
 short *See* short type
 signed
 char 4-2, A-3
 int 4-2
 long 4-2
 short 4-2
 structure
 conversion 5-46
 declaration 4-20, 4-51
 initialization 4-45, 4-47
 pointers to 4-28
 storage 4-22
 typedef 3-13, 4-52
 union
 conversion 5-46
 declaration 4-24, 4-51
 initialization 4-45, 4-47
 pointers to 4-28
 storage 4-24
 unsigned char *See* unsigned char type
 unsigned int *See* unsigned int type
 unsigned long *See* unsigned long type
 unsigned short *See* unsigned short type
 user defined 4-51, 4-52
 void 4-3, 4-5
 volatile
 described 4-3
 pointers, used with 4-27

U

Unary expressions 5-8
 Unary operators, table 2-6, 5-13
 Unary plus operator (+) 5-15
 #undef directive 8-9
 Underscore character (_) 2-2, 2-14
 Union declarations
 types 4-51
 variables 4-24
 union type specifier 4-24
 Unions
 conversion 5-46
 declaration 4-24, 4-51
 differences from Kernighan
 and Ritchie A-4, A-5
 expressions 5-2
 identifiers 5-2
 initialization 4-45, 4-47

Index

Unions (*continued*)

- members
 - described 4-24
 - naming class 3-13
 - referring to 5-7
- pointers to 4-28
- storage 4-24
- tags 3-13, 4-51
- unsigned
 - char type
 - conversion 5-41
 - described 4-2
 - differences from Kernighan and Ritchie A-2, A-3
 - range of values 4-5
 - storage 4-5
 - int type
 - conversion 5-42
 - described 4-2
 - portability 4-6
 - range of values 4-5
 - storage 4-5
 - keyword 4-3, A-2
 - long int type *See* unsigned long type
 - long type
 - conversion 5-42
 - described 4-2
 - differences from Kernighan and Ritchie A-2, A-3
 - range of values 4-5
 - storage 4-5
 - short int type *See* unsigned short type
 - short type
 - conversion 5-41
 - described 4-2
 - differences from Kernighan and Ritchie A-3
 - range of values 4-5
 - storage 4-5
 - type 4-2, A-2
- Unspecified type, pointer to (void *) 4-27
- User-defined types *See* Types
- Usual arithmetic conversions 5-13, A-3

V

- Value
 - range of 4-5, 4-7
- Values
 - range of 4-5
- Values, passing by 7-14, 7-18
- Variable names *See* Identifiers

Variables

- array
 - declaration 4-25
 - initialization 4-47, 4-50
 - storage 4-26
 - auto 4-37, 4-41, 4-45
 - communal 4-40
 - declarations
 - array 4-9, 4-25, 4-26
 - described 3-1
 - enumeration 4-18
 - external 4-38
 - form 4-16
 - fundamental types 4-17
 - global 4-38, 4-40, 4-41
 - internal 4-38
 - local 4-41
 - multidimensional arrays 4-26
 - pointer 4-27
 - simple 4-17
 - structure 4-20
 - summarized B-9
 - union 4-24
 - visibility 4-38
 - definitions
 - described 3-2, 4-38
 - summarized B-14
 - visibility 4-39, 4-41, 4-42
 - enumeration 4-18
 - extern 4-38, 4-42
 - fundamental types 4-17, 4-45
 - global 3-8, 4-38, 4-42
 - lifetime
 - global 3-6, 4-37, 4-45
 - local 3-8, 7-12
 - local 3-8, 7-12
 - multidimensional arrays 4-26, 5-5
 - naming class 3-12, A-4
 - pointer 4-27, 4-28, 4-45
 - register 4-42, 4-45
 - simple 4-17
 - static 4-38, 4-42, 4-45
 - storage allocation 3-2
 - structure 4-20, 4-22, 4-47
 - union 4-24, 4-47
 - visibility 4-38
- Vertical-tab escape sequence (\v) 2-4, A-1
- ### Visibility
- described 3-6
 - function declarations 4-44, 7-12
 - function definitions 7-4
 - global 3-7
 - nested 3-8
 - variable declarations 4-38

- Visibility (*continued*)
 - variable definitions 4-39, 4-41, 4-42
- void
 - argument-type list 4-32, 4-34
 - formal parameter list, used in A-2
 - function-return type 4-32
 - keyword A-1
 - pointer modifier, used as A-2
 - pointer to 4-27
 - type name 7-18
- void type
 - conversion 5-46
 - described 4-2, 4-3
 - range of values 4-5
 - storage 4-5
 - type specifier A-2
- volatile
 - keyword A-1
 - pointer modifier, used as 4-27
 - type specifier 4-3

W

- while statement
 - described 6-15
 - execution, continuation of 6-4
 - execution, termination of 6-2
- White-space characters 2-2, 2-4

10-31-88
SCO-514-210-035