XENIX® System V

Development System

Programmer's Guide

This document was typeset with an IMAGEN® 8/300 Laser Printer.

SCO Document Number: XG-6-21-87-4.0

Contents

Chapter 1 Introduction

1.1 Overview

This guide explains how to use the XENIX Development system to create and maintain C language and assembly language programs. The system provides a broad spectrum of programs and commands to help you design and develop applications and system software. These programs and commands enable you to create C and assembly language programs for execution on the XENIX system. They also let you debug these programs, automate their creation, and maintain different versions of the programs you develop.

The following sections introduce the programs and commands of the XENIX Development System, and explain the steps you can take to develop programs for the XENIX system. Most of the programs and commands in these introductory sections are fully explained later in this guide. Some commands mentioned here are part of the XENIX Operating System. These are explained in the XENIX *User's Guide* and XENIX *Operations Guide*.

1.2 Creating Programs

The C programming language can meet the needs of most programming projects. A complete description of how to write, compile, link, and run C programs under the XENIX operating system is provided in three documents: the *C User's Guide*, the *C Language Reference*, and the *C Library Reference*.

You may also create assembly language programs using **masm**(CP), the XENIX assembler. **masm** assembles source files and produces relocatable object files that can be linked to your C language programs with **ld**(CP). The **ld** program is the XENIX linker. It links relocatable object files created by the C compiler or assembler to produce executable programs. Note that the **cc**(CP) command automatically invokes the linker and the assembler, so use of either **masm** or **ld** is optional.

You can create source files for lexical analyzers and parsers using the program generators **lex**(CP) and **yacc**(CP). Lexical analyzers are used in programs to pick patterns out of complex input and convert these patterns into meaningful values or tokens. Parsers arc used in programs to convert meaningful sequences of tokens and values into actions. The **lex** program is the XENIX lexical analyzer generator. It generates lexical analyzers, written in C program statements, from given specification files. The **yacc** program is the XENIX parser generator. It generates parsers, written in C program statements, from given specification files. **lex** and **yacc** are often used together to make complete programs.

You can preprocess C and assembly language source files, or even **lex** and **yacc** source files using the **m4**(CP) macro processor. The **m4** program performs several preprocessing functions, such as converting macros to their defined values and including the contents of files into a source file.

1.3 Creating and Maintaining Libraries

You can create libraries of useful C and assembly language functions and programs using the **ar** and **ranlib**(CP) programs. **ar**(CP), the XENIX archiver, can be used to create libraries of relocatable object files. **ranlib**, the XENIX random library generator, converts archive libraries to random libraries and places a table of contents at the front of each library.

The **lorder**(CP) command finds the ordering relation in an object library. The **tsort**(CP) command topologically sorts object libraries so that dependencies are apparent.

1.4 Maintaining Program Source Files

You can automate the creation of executable programs from C and assembly language source files and maintain your source files using the **make** program and the SCCS commands.

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

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

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

Other commands let you examine object and executable binary files. The **nm**(CP) command prints the list of symbol names in a program. The **hd**(C) command performs a hexadecimal dump of given files, printing files in a variety of formats, one of which is hexadecimal. The **size**(CP) command reports the size of an object file. The **strings**(CP) command finds and prints readable text (strings) in an object or other binary file. The **strip**(CP) command removes symbols and relocation bits from executable files. The **sum**(C) command computes a checksum value for a file and a count of its blocks. It is used in looking for bad spots in a file and for verifying transmission of data between systems. The **xstr** command extracts strings from C programs to implement shared strings.

1.5 Creating Programs With Shell Commands

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

The **csh**(C) command invokes the C-shell, a XENIX command interpreter. The C-shell interprets and executes commands taken from the keyboard or from a command file. It has a C-like syntax which makes programming in this command language easy. It also has an aliasing facility, and a command history mechanism.

1.6 About This Guide

This guide is intended for programmers who are familiar with the C programming language and with the XENIX system. It is organized as follows:

Chapter 1, "Introduction," introduces the XENIX software development programs provided with this package and summarizes the organization of this guide and the conventions used.

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

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

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

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

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

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

Appendix A, "m4: A Macro Processor," explains how to use, create and process macros using the **m4** C functions.

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

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

Assembly language programmers should read the *Macro Assembler User's Guide* for an explanation of the XENIX assembler and Chapter 4, "adb" in the *C User's Guide* for an explanation of how to debug programs.

Programmers who wish to automate the compilation process of their programs should read Chapter 2 for an explanation of the **make** program. Programmers who wish to organize and maintain multiple versions of their programs should read Chapter 3 for an explanation of the Source Code Control System (SCCS) commands.

Special project programmers who need a convenient way to produce lexical analyzers and parsers should read Chapters 6 and 7 for explanations of the **lex** and **yacc** program generators.

1.7 Notational Conventions

This guide uses a number of notational conventions to describe the syntax of XENIX commands:

boldface	Boldface indicates a command, option, flag, or program name to be entered as shown. Boldface indicates the name of a library routine, global variable, standard type, constant, keyword, or identifier used by the C library. (To find more information on a given library routine consult the "Alphabetized List" in your XENIX *Reference* for the manual page that describes it.)
italics	Italics indicate a filename. This pertains to library include filenames (i.e. *stdio.h*), as well as, other filenames (i.e. */etc/ttys*). Italics indicate a placeholder for a command argument. When entering a command, a placeholder must be replaced with an appropriate filename, number, or option.

	Italics indicate a specific identifier, supplied for variables and functions, when mentioned in text.
	Italics indicate user named routines. (User named routines are followed by open and close parentheses, ().)
	Italics indicate emphasized words or phrases in text.
CAPITALS	Capitals indicate names of environment variables (i.e. TZ and PATH).
SMALL CAPITALS	Small capitals indicate keys and key sequences (i.e. RETURN).
[ ]	Brackets indicate that the enclosed item is optional. If you do not use the optional item, the program selects a default action to carry out.
...	Ellipses indicate that you can repeat the preceding item any number of times.
	Vertical ellipses indicate that a portion of a program example is omitted.
" "	Quotation marks indicate the first use of a technical term.
	Quotation marks indicate a reference to a word rather than a command.

Replace this Page
with Tab Marked:

make

Chapter 2
make:
A Program Maintainer

2.1 Introduction

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

This chapter explains how to use **make** to automate medium-sized programming projects. It explains how to create makefiles for each project, and how to invoke **make** for creating programs and updating files. For more details about the program, see **make**(CP) in the XENIX *Reference*.

2.2 Creating a Makefile

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

target ... : [*dependent* ...] [;*command* ...]

where *target* is the filename of the file to be updated, *dependent* is the filename of the file on which the target depends, and *command* is the XENIX command needed to create the target file. Each dependency line must have at least one command associated with it, even if it is only the null command (;).

You may give more than one target filename or dependent filename if desired. Each filename must be separated from the next by at least one space. The target filenames must be separated from the dependent filenames by a colon (:). Filenames must be spelled as defined by the XENIX system. Shell metacharacters, such as star (*) and question mark (?), can also be used.

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

You may add a comment to a makefile by starting the comment with a number sign (#) and ending it with a newline character. All characters after the number sign are ignored. Comments may be place at the end of a dependency line if desired. If a command contains a number sign, it must be enclosed in double quotation marks (").

If a dependency line is too long, you can continue it by entering a backslash (\) and a newline character.

The makefile should be kept in the same directory as the given source files. For convenience, the filenames *makefile*, *Makefile*, *s.makefile*, and *s.Makefile* are provided as default filenames. These names are used by **make** if no explicit name is given at invocation. You may use one of these names for your makefile, or choose one of your own. If the filename begins with the *s.* prefix, **make** assumes that it is an SCCS file and invokes the appropriate SCCS command to retrieve the lastest version of the file.

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

```
#include "defs"
```

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

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

In the first dependency line, *prog* is the target file and *x.o*, *y.o*, and *z.o* are its dependents. The command sequence:

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

on the next line tells how to create *prog* if it is out of date. The program is out of date if any one of its dependents has been modified since *prog* was last created.

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

2.3 Invoking make

Once you have a makefile and wish to update and modify one or more target files in the file, you can invoke **make** by typing its name and optional arguments. The invocation has the form

```
make [option]... [macdef]... [target]...
```

where *option* is a program option used to modify program operation, *macdef* is a macro definition used to give a macro a value or meaning, and *target* is the filename of the file to be updated. It must correspond to one of the target names in the makefile. All arguments are optional. If you give more than one argument, you must separate them with spaces.

You can direct **make** to update the first target file in the makefile by typing just the program name. In this case, **make** searches for the files *makefile*, *Makefile*, *s.makefile*, and *s.Makefile* in the current directory, and uses the first one it finds as the makefile. For example, assume that the current makefile contains the dependency lines given in the last section. Then the command

```
make
```

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

You can direct **make** to update a given target file by giving the filename of the target. For example,

```
make x.o
```

causes **make** to recompile the *x.o* file, if the *x.c* or *defs* files have changed since the object file was last created. Similarly, the command

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

causes **make** to recompile *x.o* and *z.o* if the corresponding dependents have been modified. **make** processes target names from the command line in a left to right order.

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

```
-f filename
```

where *filename* is the name of the makefile. You must supply a full pathname if the file is not in the current directory. For example, the command

```
make -f makeprog
```

reads the dependency lines of the makefile named **makeprog** found in the current directory. You can direct **make** to read dependency lines from the standard input by giving "-" as the *filename*. **make** reads the standard input until the end-of-file character is encountered.

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

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

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

2.4 Using Pseudo-Target Names

It is often useful to include dependency lines that have pseudo-target names, i.e., names for which no files actually exist or are produced. Pseudo-target names allow **make** to perform tasks not directly connected with the creation of a program, such as deleting old files or printing copies of source files. For example, the following dependency line removes old copies of the given object files when the pseudo-target name "cleanup" is given in the invocation of **make**.

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

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

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

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

The pseudo-target name ".PRECIOUS" prevents dependents of the current target from being deleted when **make** is terminated using the INTERRUPT or QUIT key, and the pseudo-target name ".SILENT" has the same effect as the **-s** option.

2.5 Using Macros

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

A macro definition is a line containing a name, an equal sign (=), and a value. The equal sign must not be preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

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

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

A macro is invoked by preceding the macro name with a dollar sign; macro names longer than one character must be placed in parentheses. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations.

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

The last two invocations are identical.

Macros are typically used as placeholders for values that may change from time to time. For example, the following makefile uses one macro for the names of object files to be linked and one for the names of the library.

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

If this makefile is invoked with the command:

```
make
```

it will load the three object files with the **lex**(CP) library specified with the **-lln** option.

You may include a macro definition in a command line. A macro definition in a command line has the same form as a macro definition in a makefile. If spaces are to be used in the definition, double quotation marks must be used to enclose the definition. Macros in a command line override corresponding definitions found in the makefile.

Forexample, the command:

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

loads, and assignsthelibraryoptions **-lln** and -lm to "LIBES".

You can modify all or part of the value generated from a macro invocation without changing the macro itself byusingthe "substitution sequence". The sequence has the form

name : *st1* =[*st2*]

where *name* is the name of the macro whose value is to be modified, *st1* is the character or characters to be modified, and *st2* is the character or characters to replace the modified characters. If *st2* is not given, *st1* is replaced by a null character.

The substitution sequence is typically used to allow user-defined metacharacters in a makefile. For example, suppose that ".x" is to be used as a metacharacter for a prefix and suppose that a makefile contains the definition

```
FILES =prog1.xprog2.xprog3.x
```

Then the macro invocation

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

generates the value

```
prog1.o prog2.oprog3.o
```

The actual value of "FILES" remains unchanged.

make has five built-in macros that can be used when writing dependency lines. The followingis a list of these macros.

$*	Contains the name of the current target with the suffix removed. Thus, if the current target is *prog.o*, $* contains *prog*. It maybe be used in dependency lines that redefine the built-in rules.
$@	Contains the full pathname of the current target. It may be used in dependency lines with user-defined target names.
$<	Contains the filename of the dependent that is more recent than the given target. It may be used in dependency lines with built-in target names or the **.DEFAULT** pseudo-target name.

$?	Contains the filenames of the dependents that are more recent than the given target. It may be used in dependency lines with user-defined target names.
$%	Contains the filename of a library member. It may be used with target library names (see the section "Using Libraries" later in this chapter). In this case, $@ contains the name of the library and $% contains the name of the library member.

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

2.6 Using Shell Environment Variables

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

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

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

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

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

make has two shell variables, "MAKE" and "MAKEFLAGS", that correspond to two special-purpose macros.

The "MAKE" macro provides a way to override the **-n** option and execute selected commands in a makefile. When "MAKE" is used in a command,

make will always execute that command, even if **-n** has been given in the invocation. The variable may be set to any value or command sequence.

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

The "MAKE" and "MAKEFLAGS" variables, together with the **-n** option, are typically used to debug makefiles that generate entire software systems. For example, in the following makefile, setting "MAKE" to "make" and invoking this file with the **-n** options displays all the commands used to generate the programs *prog1, prog2,* and *prog3* without actually executing them.

```
system : prog1 prog2prog3
        @echo System complete.

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

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

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

2.7 Using the Built-In Rules

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

When no explicit dependency line for a given file is given in a makefile, **make** automatically checks the default dependents of the file. It then forms the name of the dependents by removing the suffix of the given file and appending the predefined dependent suffixes. If the given file is out of date with respect to these default dependents, **make** searches for a built-in rule that defines how to create an up-to-date version of the file, then executes it. There are built-in rules for the following files.

.o	Object file
.c	C source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.l	Lex source grammar

For example, if the file *x.o* is needed and there is an *x.c* in the description or directory, it is compiled. If there is also an *x.l*, that grammar would be run through **lex**(CP) before compiling the result.

The built-in rules are designed to reduce the size of your makefiles. They provide the rules for creating common files from typical dependents. Reconsider the example given in the section 2.2, "Creating a Makefile". In this example, the program *prog* depended on three object files *x.o*, *y.o*, and *z.o*. These files in turn depended on the C language source files *x.c*, *y.c*, and *z.c*. The files *x.c* and *y.c*, also depend on the include file, *defs*. In the original example, eacb dependency and corresponding command sequence was explicitly given. Many of these dependency lines were unnecessary, since the built-in rules could have been used instead. The following is all that is needed to show the relationships between these files:

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

x.o y.o: defs
```

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

2.8 Changing the Built-in Rules

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

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

The rules and macros are displayed on the standard output.

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

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

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

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

> *suffix-rule* :
> *command*

where *suffix-rule* is a combination of suffixes showing the relationship of the implied target and dependent, and *command* is the XENIX command required to carry out the rule. If more than one command is needed, they are given on separate lines.

The new rule must begin with an appropriate *suffix-rule*. The available *suffix-rules* are:

.c	.c~
.sh	.sh~
.c.o	.c~.o
.c~.c	.s.o
.s~.o	.y.o
.y~.o	.l.o
.l~.o	.y.c
.y~.c	.l.c
.c.a	.c~.a
.s~.a	.h~.h

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

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

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

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

.SUFFIXES: *suffix ...*

where *suffix* is a lowercase letter preceded by a dot (.). If more than one suffix is given, you must use spaces to separate them.

The order of the suffixes is significant. Each suffix is a dependent of the suffixes preceding it. For example, the suffix list:

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

causes *prog.c* to be a dependent of *prog.o*, and *prog.y* to be a dependent of *prog.c*.

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

If a ".SUFFIXES" list appears more than once in a makefile, the suffixes are combined into a single list. If a ".SUFFIXES" is given without a list, all suffixes are ignored.

2.9 Using Libraries

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

lib(member-name)

where *lib* is the name of the library containing the file, and *member-name* is the name of the file. For example, the library name:

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

refers to the object file *print.o*, in the archive library *libtemp.a*.

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

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

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

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

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

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

2.10 Troubleshooting

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

```
#include "defs"
```

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

Use the **-n** option to get a listing of which commands **make** will execute, without actually executing them. For example, the command:

```
make -n
```

prints out a listing of the commands **make** would normally execute.

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

If a change to a file is small (for example, adding a new definition to an include file), the **-t** (touch) option can save a lot of time. Instead of constantly recompiling, **make** updates the modification times on the affected file. Thus, the command:

```
make -ts
```

which stands for touch silently, causes the relevant files to appear up to date.

2.11 Using make: An Example

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

make usually prints out each command before issuing it. The output shown below results from entering the following command:

```
make
```

in a directory containing only the source and makefile:

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

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

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

Figure 2-1. Makefile Contents

```
# Description file for the make command

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

#targets: dependents
#<TAB>actions

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

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

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

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

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

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

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

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

Replace this Page
with Tab Marked:

sccs

Chapter 3

SCCS: A Source Code Control System

3.1 Introduction

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

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

This chapter explains how to make SCCS files, how to update the files contained in SCCS files, and how to maintain the SCCS files once they are created. The following sections describe the basic information you need to start using the SCCS commands. Later sections describe the commands in detail.

3.2 Basic Information

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

- Files and directories
- Deltas and SIDs
- SCCS working files
- SCCS command arguments
- File administration

3.2.1 Files and Directories

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

To simplify s-file storage, all logically related files (e.g., files belonging to the same project) should be kept in the same directory. Such directories should contain s-files only, and should have read and examine permission for everyone, and write permission for the user only.

Note that you must not use the XENIX ln(C) command to create multiple copies of an s-file.

3.2.2 Deltas and SIDs

Unlike an ordinary textfile, an SCCS file (or s-file for short) contains nothing more than lists of changes. Each list corresponds to the changes needed to construct exactly one version of the file. The lists can then be combined to create the desired version from the original.

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

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

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

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

When you create a new version, the SCCS system usually creates a new SID by incrementing the level number of the original version. If you wish to create a new release, you must explicitly instruct the system to do so. A change to a release number indicates a major new version of the file. How to create a new version of a file, and change release numbers, are described later.

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

Version numbers can become quite complicated. In general, it is wise to keep the numbers as simple as possible by carefully planning the creation of each new version.

3.2.3 SCCS Working Files

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

s-file	A permanent file that contains all versions of the given text file. The versions are stored as deltas, that is, lists of changes to be applied to the original file to create the given version. The name of an s-file is formed by placing the file prefix *s.* at the beginning of the original filename.
x-file	A temporary copy of the s-file. It is created by SCCS commands which change the s-file. It is used instead of the s-file to carry out the changes. When all changes are complete, the SCCS system removes the original s-file and gives the x-file the name of the original s-file. The name of the x-file is formed by placing the prefix *x.* at the beginning of the originalfile.
g-file	An ordinary text file created by applying the deltas in a given s-file to the original file. The g-file represents a copy of the given version of the original file, and receives the same filename as the original. When created, a g-file is placed in the current working directory of the user who requested the file.
p-file	A special file containing information about the versions of an s-file currently being edited. The p-file is created when a g-file is retrieved from the s-file. The p-file exists until all currently retrieved files have been saved in the s-file; it is then deleted. The p-file contains one or more entries describing the SID of the retrieved g-file, the proposed SID of the new, edited g-file, and the login name of the user who retrieved the g-file. The p-file name is formed by placing the prefix *p.* at the beginning of the original filename.
z-file	A lock file is used by SCCS commands to prevent two users from updating a single SCCS file at the same time. Before a command modifes an SCCS file, it creates a z-file and copies its own process ID to it. Any other command which attempts to access the file while the z-file is present displays an error message and stops. When the original command

	has finished its tasks, it deletes the z-file before stopping. The z-file name is formed by placing the prefix *z.* at the beginning of the original filename.
l-file	A special file containing a list of the deltas required to create a given version of a file. The l-file name is formed by placing the prefix *l.* at the beginning of the original filename.
d-file	A temporary copy of the g-file used to generate a new delta.
q-file	A temporary file used by the **delta**(CP) command when updating the p-file. The file is not directly accessible.

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

3.2.4 SCCS Command Arguments

Almost all SCCS commands accept two types of arguments: options and filenames. These appear in the SCCS command line immediately after the command name.

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

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

The special symbol - may be used to cause the given command to read a list of filenames from the standard input. These filenames are then used as names for the files to be processed. The list must terminate with an end-of-file character.

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

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

3.2.5 File Administrator

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

3.3 Creating and Using S-files

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

This section explains how to use the **admin**(CP), **get**(CP), and **delta**(CP) commands to create and use s-files. In particular, it describes how to create the first version of a file, how to retrieve versions for reading and editing, and how to save new versions.

3.3.1 Creating an S-file

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

admin -i*filename s.filename*

where -i*filename* gives the name of the text file from which the s-file is to be created, and *s.filename* is the name of the new s-file. The name must begin with *s.* and must be unique; no other s-file in the same directory may have the same name. For example, suppose the file named *demo.c* contains the following C language program:

```
#include <stdio.h>

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

To create an s-file, enter:

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

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

After creating an s-file, the original text file should be removed using the **rm** command, since it is no longer needed. If you wish to view the text file or make changes to it, you can retrieve the file using the **get** command described in the next section.

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

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

In general, this message can be ignored unless you have specifically included keywords in your file (see the section, "Using Identification Keywords" later in this chapter).

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

3.3.2 Retrieving a File for Reading

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

get *s.filename* ...

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

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

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

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

```
1.1
6 lines
```

You may also retrieve more than one file at a time by giving multiple s-file names in the command line. For example, the command:

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

retrieves the contents of the s-files *s.demo.c* and *s.def.h* and copies them to the text files *demo.c* and *def.h*. When giving multiple s-file names in a command, you must separate each with at least one space. When the **get** command displays information about the files, it places the corresponding filename before the relevant information.

3.3.3 Retrieving a File for Editing

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

get -e *s.filename* ...

where *s.filename* is the name of the s-file containing the text file. You may give more than one filename if you wish. If you do, you must separate each name with a space.

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

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

The command retrieves the program and copies it to the file named *demo.c*. You may edit the file just as you do any other text file.

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

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

```
1.1
newdelta 1.2
6lines
```

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

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

3.3.4 Saving a New Version of a File

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

```
delta s.filename
```

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

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

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

```
comments?
```

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

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

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

```
#include <stdio.h>

main()
{
        int i=2;

        printf("This is version 1.%d0, i);
}
```

the command displays the message:

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

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

3.3.5 Retrieving a Specific Version

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

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

where **-e** is the edit option, -r*SID* gives the SID of the version to be retrieved, and *s.filename* is the name of the s-file containing the file to be retrieved. You may give more than one filename. The names must be separated with spaces.

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

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

retrieves version 1.1 from the s-file *s.demo.c*, but the command:

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

retrieves for editing a version 1.1 from both *s.demo.c* and *s.def.h*. If you give the number of a version that does not exist, the command displays an error message.

You may omit the level number of a version number if you wish, by just giving the release number. If you do, the command automatically retrieves the most recent version having the same release number. For example, if the most recent version in the file *s.demo.c* is numbered 1.4, the command:

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

retrieves version 1.4. If there is no version with the given release number, the command retrieves the most recent version in the previous release.

3.3.6 Changing the Release Number of a File

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

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

where **-e** is the required edit option, -r*rel-num* is the new release number of the file, and *s.filename* is the name of the s-file containing the file to be retrieved. The new release number must be an entirely new number, that is, no existing version may have this number. You may give more than one filename.

The command retrieves the most recent version from the s-file, then copies the new release number to the p-file. On the subsequent **delta** command, the new version is saved using the new release number and level number 1. For example, if the most recent version in the s-file *s.demo.c* is 1.4, the command:

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

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

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

3.3.7 Creating a Branch Version

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

For example, if version 1.4 already exists, the command:

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

retrieves version 1.3 for editing and gives 1.3.1.1 as the proposed SID.

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

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

3.3.8 Retrieving a Branch Version

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

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

retrieves branch version 1.3.1.1.

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

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

You may omit the sequence number if you wish. In this case, the command retrieves the most recent branch version with the given branch number.

For example, if the most recent branch version in the s-file *s.def.h* is 1.3.1.4, the command:

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

retrieves version 1.3.1.4.

3.3.9 Retrieving the Most Recent Version

You can always retrieve the most recent version of a file by using the **-t** option with the **get** command. For example, the command:

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

retrieves the most recent version from the file *s.demo.c*. You may combine the **-r** and **-t** options, to retrieve the most recent version of a given release number. For example, if the most recent version with release number 3 is 3.5, then the command:

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

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

3.3.10 Displaying a Version

You can display the contents of a version on the standard output by using the **-p** option of the **get** command. For example, the command:

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

displays the most recent version in the s-file, *s.demo.c*, on the standard output. Similarly, the command:

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

displays version 2.1 on the standard output.

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

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

copies the most recent version in the s-file, *s.demo.c*, to the file *version.c*. The SID of the file and its size is copied to the standard error file.

3.3.11 Saving a Copy of a New Version

The **delta** command normally removes the edited file after saving it in the s-file. You can save a copy of this file by using the **-n** option of the **delta** command. For example, the command:

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

first saves a new version in the s-file, *s.demo.c*, then saves a copy of this version in the file *demo.c*. You may display the file as desired, but you cannot edit the file.

3.3.12 Displaying Helpful Information

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

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

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

You may use the error code as an argument to the **help** command to display additional information about the error. The command has the form:

help *code*

where *code* is the error code given in an error message. The command displays one or more lines of text that explain the error, and suggests a possible remedy. For example, the command:

```
help co1
```

displays the message

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

The **help** command can be used at anytime.

3.4 Using Identification Keywords

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

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

3.4.1 Inserting a Keyword into a File

You may insert a keyword into any text file. A keyword is simply an uppercase letter enclosed in percent signs (%). No special characters are required. For example, "%I%" is the keyword representing the SID of the current version, and "%H%" is the keyword representing the current date.

When the program is retrieved for reading using the **get** command, the keywords are replaced by their current values. For example, if the "%M%", "%I%", and "%H" keywords are used in place of the module name, SID, and current data in the following program statement:

```
char header[100] = "%M% %I% %H% ";
```

then these keywords are expanded in the retrieved version of the program as shown below:

```
char header[100] = "MODNAME 2.3 07/07/77";
```

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

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

```
Noid keywords (cm7)
```

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

3.4.2 Assigning Values to Keywords

The values of most keywords are predefined by the system, but some, such as the value for the "%M%" keyword, can be explicitly defined by the user. To assign a value to a keyword, you must set the corresponding s-file flag to the desired value. You can do this by using the **-f** option of the **admin** command.

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

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

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

The **t** and **q** flags are also associated with keywords. A description of these flags and the corresponding keywords can be found in the section **get**(CP) in the XENIX *Reference*. You can change keyword values at any time.

3.4.3 Forcing Keywords

If a version is found to contain no keywords, you can force a fatal error by setting the **i** flag in the given s-file. The flag causes the **delta** and **admin** commands to stop processing of the given version and report an error. The flag is useful for ensuring that keywords are used properly in a given file.

To set the **i** flag, you must use the **-f** option of the **admin** command. For example, the command:

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

sets the **i** flag in the s-file, *s.demo.c*. If the given version does not contain keywords, subsequent **delta** or **admin** commands that access this file print an error message.

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

3.5 Using S-file Flags

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

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

3.5.1 Setting S-file Flags

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

```
admin -fflag s.filename
```

where -f*flag* gives the flag to be set, and *s.filename* gives the name of the s-file in which the flag is to be set. For example, the command:

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

sets the **i** flag in the s-file *s.demo.c*.

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

```
admin -fmdmod s.demo.c
```

which sets the **m** flag to the module name "dmod".

3.5.2 Using the i Flag

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

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

3.5.3 Using the d Flag

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

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

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

3.5.4 Using the v Flag

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

When the **v** flag is set, the **delta** command asks for the modification requests just before asking for comments. The **v** flag also allows the **-m** option to be used in the **delta** and **admin** commands.

3.5.5 Removing an S-file Flag

You can remove an s-file flag from an s-file by using the **-d** option of the **admin** command. The command has the form:

admin -d*flag s.filename*

where -d*flag* gives the name of the flag to be removed and *s.filename* is the name of the s-file from which the flag is to be removed. For example, the command:

```
admin -dis.demo.c
```

removes the **i** flag from the s-file *s.demo.c*. When removing a flag which takes a value, only the flag name is required. For example, the command:

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

removes the m flag from the s-file.

The **-d** and **-i** options must not be used at the same time.

3.6 Modifying S-file Information

Every s-file contains information about the deltas it contains. Normally, this information is maintained by the SCCS commands and is not directly accessible by the user. Some information, however, is specific to the user who creates the s-file, and may be changed as desired to meet the user's requirements. This information is kept in two special parts of the s-file called the "delta table" and the "description field".

The delta table contains information about each delta, such as the SID and the date and time of creation. It also contains user-supplied information, such as comments and modification requests. The description field contains a user-supplied description of the s-file and its contents. Both parts can be changed or deleted at any time to reflect changes to the s-file contents.

3.6.1 Adding Comments

You can add comments to an s-file by using the **-y** option of the **delta** and **admin** commands. This option causes the given text to be copied to the s-file as the comment for the new version. The comment may be any combination of letters, digits, and punctuation symbols. No embedded newline characters are allowed. If spaces are used, the comment must be enclosed in double quotes. The complete command must fit on one line. For example, the command:

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

saves the comment "George Wheeler" in the s-file *s.demo.c*.

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

3.6.2 Changing Comments

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

cdc -r*SID s.filename*

where -r*SID* gives the SID of the version whose comment is to be changed, and *s.filename* is the name of the s-file containing the version. The command asks for a new comment by displaying the prompt:

```
        comments?
```

You may enter any sequence of characters up to 512 characters long. The sequence may contain embedded newline characters if they are preceded by a backslash (\). The sequence must be terminated with a newline character. For example, the command:

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

prompts for a new comment for version 3.4.

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

3.6.3 Adding Modification Requests

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

The **-m** option causes the given command to save the requests following the option. A request may be any combination of letters, digits, and punctuation symbols. If you give more than one request, you must separate them with spaces and enclose the request in double quotes. For example, the command:

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

copies the requests "error35" and "optimize10" to *s.demo.c*, while saving the new version.

The **-m** option, when used with the **admin** command, must be combined with the **-i** option. Furthermore, the **v** flag must be explicitly set with the **-f** option. For example, the command:

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

inserts the modification request "error0," in the new file *s.def.h*.

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

3.6.4 Changing Modification Requests

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

```
MRs?
```

You may enter any number of requests. Each request may have any combination of letters, digits, or punctuation symbols. No more than 512 characters are allowed, and the last request must be terminated with a newline character. If you wish to remove a request, you must precede the request with an exclamation mark (!). For example, the command:

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

asks for changes to the modification requests. By responding

```
MRs? error36 !error35
```

the request "error36" is added and "error35" is removed.

3.6.5 Adding Descriptive Text

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

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

admin -t*filename s.filename*

where -t*filename* gives the name of the file containing the descriptive text, and *s.filename* is the name of the s-file to receive the descriptive text. The file to be inserted may contain any amount of text. For example, the command:

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

inserts the contents of the file *cdemo* into the description field of the s-file *s.demo.c*.

The **-t** option may also be used to initialize the description field when creating the s-file. For example, the command:

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

inserts the contents of the file *cdemo* into the new s-file *s.demo.c*. If **-t** is not used, the description field of the new s-file is left empty.

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

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

removes the descriptive text from the s-file *s.demo.c*.

3.7 Printing from an S-file

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

3.7.1 Using a Data Specification

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

prs -d*spec s.filename*

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

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

For example, the command:

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

may produce the line:

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

A complete list of the data keywords is given in the section **prs**(CP) in the XENIX *Reference*.

3.7.2 Printing a Specific Version

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

```
prs -rSID s.filename
```

where -r*SID* gives the SID of the desired version, and *s.filename* is the name of the s-file containing the version. For example, the command:

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

prints information about version 2.1 in the s-file *s.demo.c*.

If the **-r** option is not specified, the command prints information about the most recently created delta.

3.7.3 Printing Later and Earlier Versions

You can print information about a group of versions by using the **-l** and **-e** options of the **prs** command. The **-l** option causes the command to print information about all versions immediately succeeding the given version. The **-e** option causes the command to print information about all versions immediately preceding the given version. For example, the command:

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

prints all information about versions which precede version 1.4 (e.g., 1.3, 1.2, and 1.1). The command:

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

prints information about versions which succeed version 1.4 (e.g., 1.5, 1.6, and 2.1).

If both options are given, information about all versions is printed.

3.8 Editing by Several Users

The SCCS system allows any number of users to access and edit versions of a given s-file. Since users are likely to access different versions of the s-file at the same time, the system is designed to allow concurrent editing of different versions. Normally, the system allows only one user at a time to edit a given version, but you can allow concurrent editing of the same version by setting the **j** flag in the given s-file.

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

3.8.1 Editing Different Versions

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

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

3.8.2 Editing a Single Version

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

You can set the flag by using the **-f** option of the **admin** command. For example, the command:

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

sets the flag for the s-file, *s.demo.c*.

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

3.8.3 Saving a Specific Version

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

delta -r*SID s.filename*

where -r*SID* is the SID of the version being saved, and *s.filename* is the name of the s-file to receive the new version. The *SID* may be the SID of the version you have just edited, or the proposed SID for the new version. For example, if you have retrieved version 1.4 for editing (and no version 1.5 exists), both commands:

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

and

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

save version 1.5.

3.9 Protecting S-files

The SCCS system uses the normal XENIX system file permissions to protect s-files from changes by unauthorized users. In addition to the XENIX system protections, the SCCS system provides two ways to protect the s-files; the "user list" and the "protection flags". The user list is a list of login names and group IDs of users who are allowed to access the s-file and create new versions of the file. The protection flags are three special s-file flags that define which versions are currently accessible to authorized users. The following sections explain how to set and use the user list and protection flags.

3.9.1 Adding a User to the User List

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

```
admin -aname s.filename
```

where -a*name* gives the login name of the user or the group name of a group of users to be added to the list, and *s.filename* gives the name of the s-file to receive the new users. For example, the command:

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

adds the users "johnd" and "suex", and the group "marketing" to the user list of the s-file *s.demo.c*.

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

3.9.2 Removing a User from a User List

You can remove a user or a group of users from the user list of a given s-file by using the **-e** option of the **admin** command. The option is similar to the **-a** option but performs the opposite operation. The command has the form:

```
admin -ename s.filename
```

where *-ename* is the login name of a user or the group name of a group of users to be removed from the list, and *s.filename* is the name of the s-file from which the names are to be removed. For example, the command:

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

removes the user "johnd" and the group "marketing" from the user list of the s-file *s.demo.c*.

3.9.3 Setting the Floor Flag

The floor flag, **f**, defines the release number of the lowest version a user may edit in a given s-file. You can set the flag by using the **-f** option of the **admin** command. For example, the command:

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

sets the floor to release number 2. If you attempt to retrieve any versions with a release number less than 2, an error results.

3.9.4 Setting the Ceiling Flag

The ceiling flag, **c**, defines the release number of the highest version a user may edit in a given s-file. You can set the flag by using the **-f** option of the **admin** command. For example, the command:

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

sets the ceiling to release number 5. If you attempt to retrieve any versions with a release number greater than 5, an error results.

3.9.5 Locking a Version

The lock flag, **l**, lists by release number all versions in a given s-file which are locked against further editing. You can set the flag by using the **-f** flag of the **admin** command. The flag must be followed by one or more release numbers. Multiple release numbers must be separated by commas (,).

For example, the command:

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

locks all versions with release number 3 against further editing. The command:

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

locks all versions with release numbers 4, 5, and 9.

Note that the special symbol "a" may be used to specify all release numbers. The command:

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

locks all versions in the file *s.demo.c*.

3.10 Repairing SCCS Files

The SCCS system carefully maintains all SCCS files, making damage to the files very rare. However, damage can result from hardware malfunctions, which cause incorrect information to be copied to the file. The following sections explain how to check for damage to SCCS files, and how to repair the damage or regenerate the file.

3.10.1 Checking an S-file

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

```
        corrupted file (co6)
```

indicating damage to the file. For example, the command:

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

checks the s-file, *s.demo.c*, for damage by generating a new checksum for the file, and comparing the new sum with the existing sum.

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

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

3.10.2 Editing an S-file

When an s-file is discovered to be damaged, it is a good idea to restore a backup copy of the file from a backup disk, rather than attempting to repair the file. (Restoring a backup copy of a file is described in the XENIX *Operations Guide.*) If this is not possible, the file may be edited using a XENIX text editor.

To repair a damaged s-file, use the description of an s-file given in the section in the XENIX *Reference*, to locate the part of the file which is damaged. Use extreme care when making changes; small errors can cause unwanted results.

3.10.3 Changing an S-file's Checksum

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

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

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

3.10.4 Regenerating a G-file for Editing

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

3.10.5 Restoring a Damaged P-file

The **-g** option of the **get** command may be used to generate a new copy of a p-file that has been accidentally removed. For example, the command:

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

creates a new p-file entry for the most recent version in *s.demo.c*. If the file *demo.c* already exists, it will not be changed by this command.

3.11 Using Other Command Options

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

3.11.1 Getting Help With SCCS Commands

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

```
help rmdel
```

displays the message

```
rmdel:
        rmdel -rSID filename...
```

3.11.2 Creating a File With the Standard Input

You can direct **admin** to use the standard input as the source for a new s-file by using the **-i** option without a filename. For example, the command:

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

causes **admin** to create a new s-file named *s.demo.c*, which uses the text file *demo.c* as its first version.

This method of creating a new s-file is typically used to connect **admin** to a pipe. For example, the command:

```
cat mod1.c mod2.c | admin -is.mod.c
```

creates a new s-file, *s.mod.c*, which contains the first version of the concatenated files *mod1.c* and *mod2.c*.

3.11.3 Starting At a Specific Release

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

admin -r*rel-num s.filename*

where *-rrel-num* gives the value of the starting release number, and *s.filename* is the name of the s-file to be created. For example, the command:

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

starts with release number 3. The first version is 3.1.

3.11.4 Adding a Comment to the First Version

You can add a comment to the first version of file by using the **-y** option of the **admin** command when creating the s-file. For example, the command:

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

inserts the comment "George Wheeler" in the new s-file, *s.demo.c*.

The comment may be any combination of letters, digits, and punctuation symbols. If spaces are used, the comment must be enclosed in double quotes. The complete command must fit on one line.

If the **-y** option is not used when creating an s-file, a comment of the following form:

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

is automatically inserted.

3.11.5 Suppressing Normal Output

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

The **-s** option is often used with the **-p** option to pipe the output of the **get** command to other commands. For example, the command:

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

copies the most recent version in the s-file, *s.demo.c*, to the line printer.

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

3.11.6 Including and Excluding Deltas

You can explicitly define which deltas you wish to include, and which you wish to exclude, when creating a g-file, by using the **-i** and **-x** options of the **get** command.

The **-i** option causes the command to apply the given deltas when constructing a version. The **-x** option causes the command to ignore the given deltas when constructing a version. Both options must be followed by one or more SIDs. If multiple SIDs are given, they must be separated by commas (,). A range of SIDs may be given, by separating two SIDs with a hyphen (-). For example, the command:

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

causes deltas 1.2 and 1.3 to construct the g-file. The command:

```
get -x1.2-1.4 s.demo.c
```

causes deltas 1.2 through 1.4 to be ignored when constructing the file.

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

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

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

The **-x** option is useful if you wish to remove changes performed on a given version. For example, the command:

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

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

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

3.11.7 Listing the Deltas of a Version

You can create a table showing the deltas required to create a given version by using the **-l** option. This option causes the **get** command to create an l-file which contains the SIDs of all deltas used to create the given version.

The option is typically used to create a history of a given version's development. For example, the command:

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

creates a file named *l.demo.c*, containing the deltas required to create the most recent version of *demo.c*.

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

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

copies the list of deltas required to create version 2.3 of *demo.c* to the standard output.

Note that the **-l** option may be combined with the **-g** option to create a list of deltas without retrieving the actual version.

3.11.8 Mapping Lines to Deltas

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

3.11.9 Naming Lines

You can name each line in a given version with the current module name (i.e., the value of the **%M%** keyword) by using the **-n** option of the **get** command. This option causes each line of the retrieved file to be preceded by the value of the **%M%** keyword and a tab character.

The **-n** option is typically used to indicate that a given line is from the given file. When both the **-m** and **-n** options are specified, each line begins with the **%M%** keyword.

3.11.10 Displaying a List of Differences

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

3.11.11 Displaying File Information

You can display information about a given version by using the **-g** option of the **get** command. This option suppresses the actual retrieval of a version and causes only the information about the version, such as the SID and size, to be displayed.

The **-g** option is often used with the **-r** option to check for the existence of a given version. For example, the command:

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

displays information about version 4.3 in the s-file, *s.demo.c*. If the version does not exist, the command displays an error message.

3.11.12 Removing a Delta

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

rmdel -r*SID* *s.filename*

where -r*SID* gives the SID of the delta to be removed, and *s.filename* is the name of the s-file from which the delta is to be removed. The delta must be the most recently created delta in the s-file. Furthermore, the user must have write permission in the directory containing the s-file, and must either own the s-file or be the user who created the delta.

For example, the command:

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

removes delta 2.3 from the s-file, *s.demo.c*.

The **rmdel** command will refuse to remove a protected delta, that is, a delta whose release number is below the current floor value, above the current ceiling value, or equal to a current locked value (see the section "Protecting S-files" given earlier in this chapter). The command will also refuse to remove a delta which is currently being edited.

The **rmdel** command should be reserved for those cases in which incorrect, global changes were made to an s-file.

Note that **rmdel** changes the type indicator of the given delta from "D" to "R". A type indicator defines the type of delta. Type indicators are described in detail in the section in the XENIX *Reference*.

3.11.13 Searching for Strings

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

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

and the command:

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

is executed, then the command:

```
        what prog.c
```

displays:

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

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

3.11.14 Comparing SCCS Files

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

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

where -r*SID1* and -r*SID2* give the SIDs of the versions to be compared, and *s.filename* is the name of the s-file containing the versions. The version SIDs must be given in the order in which they were created. For example, the command:

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

displays the differences between versions 3.4 and 5.6. The differences are displayed in a form similar to the XENIX diff command.

Replace this Page
with Tab Marked:

lint

Chapter 4

lint: A C Program Checker

4.1 Introduction

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

In particular, **lint** checks for:

- Unused functions and variables
- Unknown values in local variables
- Unreachable statements and infinite loops
- Unused and misused return values
- Inconsistent types and type casts
- Mismatched types in assignments
- Nonportable and old-fashioned syntax
- Strange constructions
- Inconsistent pointer alignment and expression evaluation order

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

4.2 Invoking lint

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

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

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

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

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

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

filename (*num*): *description*

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

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

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

4.3 Options

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

Suppressive Options

– a	Suppresses complaints about assignments of long values to variables that are not long.
– b	Suppresses complaints about **break** statements that cannot be reached (programs produced by **lex** or **yacc** will often result in a large number of such complaints).
– c	Suppresses complaints about casts that have questionable portability.
– h	Does not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

−u	Suppresses complaints about functions and external variables used and not defined, or defined and not used (this option is suitable for running **lint** on a subset of files of a larger program).
−v	Suppresses complaints about unused arguments in functions.
−x	Does not report variables referred to by external declarations but never used.

Other Options

−n	Does not check compatibility against either the standard or the portable lint library.
−p	Attempts to check portability to other dialects of C.
−l*library*	Checks function definitions in the specified lint *library*. For example, **−lm** causes the library *llibm.ln* to be searched.

4.4 Checking for Unused Variables and Functions

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

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

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

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

Unused variables and functions often occur during the development of large programs. It is not uncommon for a programmer to remove all references to a variable or function from a source file, but forget to remove its declaration. Such unused variables and functions rarely cause working programs to fail, but do make programs harder to understand and change. Checking for unused variables and functions can also help you find variables or functions that you intended to use but accidentally have left out of the program.

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

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

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

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

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

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

4.5 Checking Local Variables

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

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

```
char c;

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

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

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

```
int i,total;

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

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

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

4.6 Checking for Unreachable Statements

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

```
int x,y;

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

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

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

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

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

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

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

4.7 Checking for Infinite Loops

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

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

and:

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

are both considered infinite loops. The statements:

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

and:

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

will be reported as never executed.

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

4.8 Checking Function Return Values

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

Within a function definition, the appearance of both:

```
return (expr);
```

and:

```
return ;
```

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

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

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

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

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

4.9 Checking for Unused Return Values

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

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

4.10 Checking Types

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

1. Across certain binary operators and implied assignments
2. At the structure selection operators
3. Between the definition and uses of functions
4. In the use of enumerations

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

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

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

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

4.11 Checking Type Casts

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

```
p=1;
```

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

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

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

4.12 Checking for Nonportable Character Use

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

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

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

```
        nonportable character comparison
```

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

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

4.13 Checking for Assignment of longs to ints

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

4.14 Checking for Strange Constructions

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

```
        *p++;
```

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

```
        null effect
```

The program fragment:

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

is also considered strange since the test will never succeed.

Similarly, the test

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

is equivalent to

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

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

```
degenerate unsigned comparison
```

If you use:

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

then **lint** reports

```
constant in conditional context
```

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

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

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

or:

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

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

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

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

4.15 Checking for Use of Older C Syntax

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

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

```
a=-1;
```

which could be taken as either:

```
a=- 1;
```

or:

```
a = -1;
```

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

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

```
int x 1;
```

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

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

looks somewhat like the beginning of a function declaration:

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

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

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

This form is free of any possible syntactic ambiguity.

4.16 Checking Pointer Alignment

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

```
possible pointer alignment problem
```

results from this situation.

4.17 Checking Expression Evaluation Order

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

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

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

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

will draw the comment:

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

4.18 Embedding Directives

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

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

```
/* NOTREACHED */
```

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

```
/* NOSTRICT*/
```

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

```
/* ARGSUSED */
```

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

```
/*VARARGS */
```

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

```
/* VARARGS2 */
```

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

```
/* LINTLIBRARY*/
```

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

4.19 Checking For Library Compatibility

lint accepts certain library directives, such as:

```
-ly
```

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

```
/* LINTLIBRARY */
```

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

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

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

Replace this Page
with Tab Marked:

lex

Chapter 5

lex: A Lexical Analyzer

5.1 Introduction

lex(CP) is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem-oriented specification for character string matching, and produces a C program that recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to **lex**. The **lex** code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings, program sections provided by the user are executed. The **lex** source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by **lex**, the corresponding fragment is executed.

The user supplies the additional code needed to complete his tasks, including code written by other generators. The program that recognizes the expressions is generated in the from the user's C program fragments. **lex** is not a complete language, but rather a generator representing a new language feature added on top of the C programming language.

lex turns the user's expressions and actions (called *source* in this chapter) into a C program named *yylex*. The *yylex* program recognizes expressions in a stream (called input in this chapter) and performs the specified actions for each expression as it is detected.

Consider a program to delete from the standard input all blanks or tabs at the ends of lines. The following lines:

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

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

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

The finite automaton generated for this source scans for both rules at once, checks for the termination of a string of blanks or tabs; whether or not there is a newline character; and then executes the desired rule's action.

The first rule matches all strings of blanks or tabs at the end of lines, and the second rule matches all remaining strings of blanks or tabs.

lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. **lex** can also be used with a parser generator to perform the lexical analysis phase; it is especially easy to interface **lex** and **yacc**(CP). **lex** programs recognize only regular expressions; **yacc** writes parsers that accept a large class of context-free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of **lex** and **yacc** is often appropriate. When used as a preprocessor for a later parser generator, **lex** is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. Additional programs, written by other generators or by hand, can be added easily to programs written by **lex**. **yacc** users will realize that the name *yylex* is what **yacc** expects its lexical analyzer to be named, so that the use of this name by **lex** simplifies interfacing.

lex generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a **lex** program to recognize and partition an input stream is proportional to the length of the input. The number or the complexity of **lex** rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by **lex**.

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

lex is not limited to source that can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, **lex** will recognize *ab* and leave the input pointer just before *cd*. Such backup is more costly than the processing of simpler languages.

5.2 lex Regular Expressions

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

```
integer
```

matches the string *integer* wherever it appears and the expression:

```
a57D
```

looks for the string *a57D*.

The operator characters are:

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

If any of these characters are to be used literally, they need to be quoted individually with a backslash (\), or as a group within quotation marks ("). The quotation mark operator (") indicates that whatever is contained between a pair of quotation marks is to be taken as text characters. Thus:

```
xyz"++"
```

matches the string *xyz++* when it appears. Note that part of a string may be quoted. It is harmless, but unnecessary, to quote an ordinary text character; the expression:

```
"xyz++"
```

is the same as the one above. Thus by quoting every nonalphanumeric character being used as a text character, you do not have to remember the above list of current operator characters.

An operator character may also be turned into a text character by preceding it with a backslash (\) as in:

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. The quoting mechanism can also be used to get a blank into an expression. Normally, blanks or tabs end a rule. Any blank character not contained within brackets must be quoted. Several normal C escapes with the backslash (\) are recognized:

\n	newline

\t	tab
\b	backspace
\\	backslash

Since newline is illegal in an expression, a \n must be used; it is not required to escape tab and backspace. Note that every character is always a text character. Exceptions to this are blanks, tabs, newlines and the operator characters shown in the list above.

5.3 Invoking lex

There are two steps in compiling a **lex** source program. First, the **lex** source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of **lex** subroutines. The generated program is in a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library.

The library is accessed by the loader flag *-ll*. So an appropriate set of commands is

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

The resulting program is placed in the file *a.out* for later execution. To use **lex** with **yacc** see the section "lex and yacc" in this chapter and Chapter 6, "yacc: A Compiler-Compiler"". Although the default **lex** I/O routines use the C standard library, the **lex** automata themselves do not. If private versions of *input*(), *output*(), and *unput*() are given, the library can be avoided.

5.4 Specifying Character Classes

Classes of characters can be specified using brackets: [and]. The construction:

```
[abc]
```

matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are the backslash (\), the dash (-), and the caret (^). The dash character indicates ranges. For example:

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

indicates the character class containing all the lowercase letters, the digits, the angle brackets, and underline. Ranges may be given in either order.

Using the dash between any pair of characters that are not both uppercase letters, both lowercase letters, or both digits is implementation dependent and causes a warning message. If it is desired to include the dash in a character class, it should be first or last; thus:

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

matches all the digits and the plus and minus signs.

In character classes, the caret (^) operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus:

```
[^abc]
```

matches all characters except *a*, *b*, or *c*, including all special or control characters; or:

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

is any character which is not a letter. The backslash (\) provides an escape mechanism within character class brackets, so that characters can be entered literally by preceding them with this character.

5.5 Specifying an Arbitrary Character

To match almost any character, the period (.) designates the class of all characters except a newline. Escaping into octal is possible although non-portable. For example:

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

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

5.6 Specifying Optional Expressions

The question mark (?) operator indicates an optional element of an expression. Thus:

```
ab?c
```

matches either *ac* or *abc*. Note that the meaning of the question mark here differs from its meaning in the shell.

5.7 Specifying Repeated Expressions

Repetitions of classes are indicated by the asterisk (*) and plus (+) operators. For example:

```
a*
```

matches any number of consecutive *a* characters, including zero; while *a+* matches one or more instances of *a*. For example:

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

matches all strings of lowercase letters, and:

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

matches all alphanumeric strings with a leading alphabetic character. Note that this is a typical expression for recognizing identifiers in computer languages.

5.8 Specifying Alternation and Grouping

The vertical bar (|) operator indicates alternation. For example:

```
(ab|cd)
```

matches either *ab* or *cd*. Note that parentheses are used for grouping, although they are not necessary at the outside level. For example:

```
ab|cd
```

would have sufficed in the preceding example. Parentheses should be used for more complex expressions, such as:

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

which matches such strings as *abefef*, *efefef*, *cdef*, and *cddd*, but not *abc*, *abcd*, or *abcdef*.

5.9 Specifying Context Sensitivity

lex recognizes a small amount of surrounding context. The two simplest operators for this are the caret (^) and the dollar sign ($). If the first character of an expression is a caret, then the expression is only matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of the caret, complementation of character classes, since complementation only applies within brackets. If the very last character is a dollar sign, the expression is only matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the slash (/) operator, which indicates trailing context. The expression:

```
ab/cd
```

matches the string *ab*, but only if followed by *cd*. Thus:

```
ab$
```

is the same as:

```
ab/\n
```

Left context is handled in **lex** by specifying start conditions as explained in section 5.14, "Specifying Left Context Sensitivity". If a rule is only to be executed when the **lex** automaton interpreter is in start condition *x*, the rule should be enclosed in angle brackets:

```
<x>
```

If we considered start condition ONE as being at the beginning of a line, then the caret (^) operator would be equivalent to:

```
<ONE>
```

Start conditions are explained in detail later in this chapter.

5.10 Specifying Expression Repetition

The curly braces ({ and }) specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example:

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression.

5.11 Specifying Definitions

The definitions are given in the first part of the **lex** input, before the rules. In contrast:

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

looks for 1 to 5 occurrences of the character *a*.

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

5.12 Specifying Actions

When an expression is matched by a pattern of text in the input, **lex** executes the corresponding action. This section describes some features of **lex** which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the **lex** user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When **lex** is being used with **yacc**, this is considered to be the normal situation. You may consider that actions are done instead of copying the input to the output; thus, a rule which merely copies can be omitted.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement ; as an action causes this result. A frequent rule is:

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

which causes the three spacing characters (blank, tab, and newline) to be ignored.

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

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

with the same result, although in a different style. The quotes around *\n* and *\t* are not required.

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

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

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

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

prints the string in *yytext*. The C function **printf**(S) accepts a format argument and data to be printed; in this case, the format is *printstring* where the percent sign (%) indicates data conversion, and the *s* indicates string type, and the data are the characters in *yytext*. This places the matched string on the output. This action is so common that it may be written as ECHO. For example:

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

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

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

is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence **lex** also provides a count of the number of characters matched in the variable, *yyleng*. To count both the number of words and the number of characters in words in the input, you might enter:

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

which accumulates the number of characters in the words recognized, and places the result in the variable *chars*. The last character in the matched string can be accessed with:

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

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

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

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

which, when faced with a string such as:

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

will first match the five characters:

```
"abc\
```

and then the call to *yymore*() will cause the next part of the string:

```
"def
```

to be tacked on the end. Note that the final quotation mark terminating the string should be picked up in the code labeled normal processing.

The function *yyless*() might be used to reprocess text in various circumstances. Consider the problem in the older C syntax of distinguishing the ambiguity of =−*a*. Suppose it is desired to treat this as =− *a* and to then print a message. The following rule might apply:

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

which prints a message, returns the letter after the operator to the input stream, and treats the operator as =−.

Alternatively, it might be desired to treat this as = *-a*. To do this, just return the minus sign as well as the letter to the input. The following performs the interpretation:

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

Note that the expressions for the two cases could also be written as:

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

in the first case and:

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

in the second. No backup would be required in the rule action shown above. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of =*-3*, however, makes:

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

a better rule.

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

1. *input*() which returns the next input character;

2. *output*(c) which writes the character *c* on the output;

3. *unput*(c) which pushes the character *c* back onto the input stream to be read later by *input*().

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

```
+ * ? $
```

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

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

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

5.13 Handling Ambiguous Source Rules

lex can handle ambiguous specifications. When more than one expression can match the current input, **lex** chooses the following:

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

For example, suppose the following rules are given:

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

If the input is *integers*, it is taken as an identifier, because:

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

matches 8 characters while:

```
integer
```

matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g., *int*) does not match the expression *integer*, so the identifier interpretation is used.

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

```
.*
```

For example:

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote.

Presented with the input:

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

the above expression matches:

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

which is probably not what was wanted. A better rule to follow is the form:

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

which, on the above input, stops after 'first'. The consequences of errors like this are mitigated by the fact that the dot (.) operator does not match a newline. Therefore, no more than one line is ever matched by such expressions. Do not try to defeat this with expressions like:

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

or their equivalents: the **lex** generated program will try to read the entire input file, causing internal buffer overflows.

Note that **lex** is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some of the **lex** rules to do this might be:

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides *he* and *she*. Remember that the period (.) does not include the newline. Since *she* includes *he*, **lex** will normally not recognize the instances of *he* included in *she*, since once it has passed a *she*, those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means go do the next alternative. It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of *he*. The following can then be applied:

```
she    {s++; REJECT;}
he     {h++; REJECT;}
\n     |
.      ;
```

These rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, the user could note that *she* includes *he*, but not vice versa, and omit the REJECT action on *he*; in other cases, it would not be possible to tell which input characters were in both classes.

Consider the two rules:

```
a[bc]+ { ... ; REJECT;}
a[cd]+ { ... ; REJECT;}
```

If the input is *ab*, only the first rule matches, and on *ad*, only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word "the" is considered to contain both *th* and *he*. Assuming a two-dimensional array named *digram* is to be incremented, the appropriate source is as follows:

```
%%
[a-z][a-z]     {digram[yytext[0]][yytext[1]]++; REJECT;}
.              ;
\n             ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

Remember that REJECT does not rescan the input. Instead, it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, you must not have used *unput*() to change the incoming characters from the input stream. This is the only restriction placed on the ability to manipulate the not-yet-processed input.

5.14 Specifying Left Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The caret (^) operator is a prior context operator, recognizing immediately preceding left context just as the dollar sign ($) recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments:

1. The use of flags, when only a few rules change from one environment to another.

2. The use of start conditions with rules.

3. The use of multiple lexical analyzers running together.

In each case, there are rules that recognize the need to change the environment in which the following input text is analyzed, and to set some parameter in order to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since **lex** is not involved at all. It may be more convenient, however, to have **lex** remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when **lex** is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are not similar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line that began with the letter *a*; changing *magic* to *second* on every line that began with the letter *b*; and changing *magic* to *third* on every line that began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with the following flag:

```
        int flag;
%%
^a      {flag='a'; ECHO;}
^b      {flag='b'; ECHO;}
^c      {flag='c'; ECHO;}
\n      {flag= 0 ; ECHO;}
magic {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to **lex** in the definitions section with a line reading:

```
%Start        name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with angle brackets. For example:

```
<name1>expression
```

is a rule that is only recognized when **lex** is in the start condition *name1*. To enter a start condition, execute the action statement:

```
BEGIN name1;
```

which changes the start condition to *name1*. To return to the initial state, enter:

```
BEGIN 0;
```

which resets the initial condition of the **lex** automaton interpreter. A rule may be active in several start conditions; for example:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can also be written as:

```
%START AA BB CC
%%
^a      {ECHO; BEGIN AA;}
^b      {ECHO; BEGIN BB;}
^c      {ECHO; BEGIN CC;}
\n      {ECHO; BEGIN 0;}
<AA>magic       printf("first");
<BB>magic printf("second");
<CC>magic printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but **lex** does the work rather than the user's code.

5.15 Specifying Source Definitions

Remember the format of the **lex** source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far, only the rules have been described. You will need additional options, though, to define variables for use in your program and for use by **lex**. These can go either in the definitions section or in the rules section.

Remember that **lex** is turning the rules into a program. Any source not intercepted by **lex** is copied into the generated program. There are three classes of such things:

1. Any line that is not part of a **lex** rule or action which begins with a blank or tab is copied into the **lex** generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by **lex** which contains the actions. This material must look like program fragments, and should precede the first **lex** rule.

   As a side effect of the above, lines that begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the **lex** source or the generated code. The comments should follow C language conventions.

2. Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3. Anything after the third %% delimiter, regardless of format, is copied out after the **lex** output.

Definitions intended for **lex** are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and beginning in column 1, is assumed to define **lex** substitution strings. The format of such lines is:

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. For example, using {D} for the digits and {E} for an exponent field might abbreviate rules to recognize numbers:

```
D                       [0-9]
E                       [DEde][-+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E}                 printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a FORTRAN expression such as: *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including a character set table, a list of start conditions, or adjustments to the default size of arrays within **lex** itself for larger source programs. These possibilities are discussed in the section "Source Format".

5.16 lexandyacc

If you want to use **lex** with **yacc**, note that what **lex** writes is a program named *yylex*(), the name required by **yacc** for its analyzer. Normally, the default main program on the **lex** library calls this routine, but if **yacc** is loaded, and its main program is used, **yacc** will call *yylex*(). In this case, eachlexrule shouldend with:

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to **yacc**'s names for tokens is to compile the **lex** output file as part of the **yacc** output file by placing the line:

```
#include "lex.yy.c"
```

in the last section of **yacc** input. Supposing the grammar to be named *good*, and the lexical rules to be named *better*, the XENIX command sequence can be entered as:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

The **yacc** library (-ly) should be loaded before the **lex** library, to obtain a main program which invokes the **yacc** parser. The generation of **lex** and **yacc** programs can be done in either order.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable **lex** source program to do just that:

```
%%
        int k;
[0-9]+ {
                k = atoi(yytext);
                if (k%7==0)
                  printf("%d", k+3);
                else
                  printf("%d",k);
                }
```

The rule [0-9]+ recognizes strings of digits; **atoi** (see **atof**(S)) converts the digits to binary and stores the result in *k*. The remainder operator (%) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as 49.63 or X7.

Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as shown below:

```
%%
          int k;
-?[0-9]+        {
                k = atoi(yytext);
                printf("%d", k%7==0 ? k+3 : k);
                }
-?[0-9.]+               ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a decimal point or preceded by a letter will be picked up by one of the last two rules, and not changed. The **if–else** has been replaced by a C conditional expression to save space; the form *a?b:c* means: if *a* then *b* else *c*.

For an example of statistics gathering, the following is a program which makes histograms of word lengths, where a word is defined as a string of letters.

```
          int lengs[100];
%%
[a-z]+  lengs[yyleng]++;
.       |
\n      ;
%%
yywrap()
{
int i;
printf("Length No. words\n");
for(i=0; i<100; i++)
  if (lengs[i] > 0)
     printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return*(1); indicates that **lex** is to perform wrapup. If *yywrap*() returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap*() that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written to convert double precision FORTRAN to single precision FORTRAN. Because FORTRAN does not distinguish between upper- and lowercase letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
.       .
.       .
.       .
z       [zZ]
```

An additional class recognizes white space:

```
W       [\t]*
```

The first rule changes *double precision* to *real*, or *DOUBLE PRECISION* to *REAL*.

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
  printf(yytext[0]=='d'? "real" : "REAL");
  }
```

Care is taken throughout this program to preserve the case of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"     "[^ 0]       ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of the caret (^) here. The following is a lex program that changes double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+                    |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+         |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+         {
  /* convert constants */
  for(p=yytext; *p != 0; p++)
     {
     if(*p=='d' ||*p=='D')
        *p+='e'-'d';
         ECHO;
     }
```

After the floating point constant is recognized, it is scanned by the for loop, to find the letter "d" or "D". The program then adds "'e'-'d'", which converts it to the next letter of the alphabet. The modified constant, now single precision, is written out again. The following is a series of names which must be respelled, in order to remove their initial "d". By using the array *yytext*, the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}              |
{d}{c}{o}{s}              |
{d}{s}{q}{r}{t}           |
{d}{a}{t}{a}{n}           |
...
{d}{f}{l}{o}{a}{t}printf("%s",yytext+1);
```

Another list of names must have the initial *d* changed to initial *a*:

```
{d}{l}{o}{g}              |
{d}{l}{o}{g}10            |
{d}{m}{i}{n}1             |
{d}{m}{a}{x}1             {
          yytext[0]+='a'-'d';
          ECHO;
          }
```

And one routine must have the initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h} {
          yytext[0]+='r' -'d';
          ECHO;
}
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*            |
[0-9]+                          |
\n                              |
.        ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

5.17 Specifying Character Sets

The programs generated by **lex** handle character I/O only through the *input*(), *output*(), and *unput*() routines. Thus, the character representation provided in these routines is accepted by **lex** and employed to return values in *yytext*. For internal use, a character is represented as a small integer, and, if the standard library is used, it has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented in the same form as the character constant:

```
'a'
```

If this interpretation is changed, by providing I/O routines that translate the characters, **lex** must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only *%T*. The table contains lines of the following form:

```
{integer} {character string}
```

which indicate the value associated with each character. For example:

```
%T
1     Aa
2     Bb
...
26    Zz
27    \n
28    +
29    -
30    0
31    1
...
39    9
%T
```

This table maps the lowercase and uppercase letters together into the integers 1 through 26, the newline into 27, the plus(+) and minus (−) into 28 and 29, and the digits into 30 through 39. Note the escape for a newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a larger number than the size of the hardware character set.

5.18 Source Format

The general form of a **lex** source file is:

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

The definitions section contains a combination of:

1. Definitions, in the form "name space translation".

2. Included code, in the form "space code".

3. Included code, in the form:

   ```
   %{
   code
   %}
   ```

4. Start conditions, given in the form:

   ```
   %S name1 name2 ...
   ```

5. Character set tables, in the form:

   ```
   %T
   number space character-string
   %T
   ```

6. Changes to internal array sizes, in the form:

   ```
   %x   nnn
   ```

   where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	treenodes
a	transitions
k	packed characterclasses
o	output array size

Lines in the rules section have the form:

expression action

where the action maybe continued on succeeding lines by using braces to delimit it.

Regular expressions in **lex** use the following operators:

x	The character "x".
"x"	An "x", even if x is an operator.
\x	An "x", even if x is an operator.
[xy]	The character x or y.
[x–z]	The characters x, y or z.
[^x]	Any character but x.
.	Any character but newline.
^x	An x at the beginning of a line.
<y>x	An x when lex is in start condition y.
x$	An x at the end of a line.
x?	An optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	An x or a y.
(x)	An x.
x/y	An x, but only if followed by y.
{xx}	The translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x.

Replace this Page
with Tab Marked:

yacc

Chapter 6

yacc: A Compiler-Compiler

6.1 Introduction

Computer program input generally has some structure; every computer program that accepts input can be thought of as defining an input language which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often lax about checking their inputs for validity.

yacc(CP) provides a general tool for describing the input to a computer program. The name **yacc** stands for "yet another compiler-compiler". The **yacc** user specifies the structures of his input, together with the code to be invoked as each structure is recognized. **yacc** turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have flow control in the user's application handled by this subroutine.

The input subroutine produced by **yacc** calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle peculiar features such as comment and continuation conventions, which typically defy easy grammatical specification. The class of specifications accepted is a very general one: LALR grammars with certain rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., **yacc** has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a FORTRAN debugging system.

yacc provides a general tool for imposing structure on the input to a computer program. The **yacc** user prepares a specification of the input process; this includes rules describing the input structure, the code to be invoked when these rules are recognized, and a low-level routine to do the basic input. **yacc** then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (called the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules. When one of these rules has been recognized, user code supplied for this rule is invoked. Note that actions have the ability to return values and make use of the values of other actions.

yacc is written in a portable dialect of C and the actions, and output subroutine, are also written in C. Moreover, many of the syntactic conventions of **yacc** follow the C language syntax.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be:

```
        date : month_name day ',' year ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma (,) is enclosed in single quotation marks; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input:

```
        July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. A structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules:

```
        month_name : 'J' 'a' 'n' ;
        month_name : 'F' 'e' 'b' ;
            .
            .
            .
        month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters, such as the comma, must also be passed through the lexical analyzer and are considered tokens.

Specification files are very flexible. It is relatively easy to add the following rule to the example shown above:

```
date : month '/' day '/' year ;
```

allowing:

```
7/4/1776
```

as a synonym for:

```
July 4, 1776
```

In most cases, this new rule could be slipped in to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for **yacc** to handle are also frequently difficult for you to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development stage.

The next several sections describe:

- The preparation of grammar rules.
- The preparation of the user supplied actions associated with the grammar rules.
- The preparation of lexical analyzers.
- The operation of the parser.
- Various reasons why **yacc** may be unable to produce a parser from a specification, and what to do about it.

- A simple mechanism for handling operator precedences in arithmetic expressions.
- Error detection and recovery.
- The operating environment and special features of the parsers **yacc** produces.
- Gives some suggestions which should improve the style and efficiency of the specifications.

6.2 Specifications

Names refer to either tokens or nonterminal symbols. **yacc** requires token names to be declared as such. In addition, for reasons discussed later, it is often desirable to include the lexical analyzer as part of the specification file. It may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent %% marks. (The percent sign (%) is generally used in **yacc** specifications as an escape character.)

In other words, a full specification file is shown as:

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the program section is omitted, the second %% mark may be omitted also; thus, the smallest legal **yacc** specification is:

```
%%
rules
```

Blanks, tabs, and newlines are ignored; note that they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* ... */, as in C.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and *BODY* represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of arbitrary length, and may be made up of letters, dot (.), the underscore (_), and noninitial digits. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotation marks ('). As in C, the backslash (\)is an escape character within literals, and all the C escapes are recognized. Thus:

'\n'	Newline
'\r'	Return
'\''	Single quotation mark
'\\'	Backslash
'\t'	Tab
'\b'	Backspace
'\f'	Form feed
'\xxx'	"xxx" in octal

For a number of technical reasons, the ASCII NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, then the vertical bar (|) can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules:

```
A : B C D ;
A : E F ;
A : G ;
```

can be given to **yacc** as:

```
A : B C D
  | E F
  | G
  ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, it can be indicated as follows:

```
empty : ;
```

Names representing tokens must be declared; this can be done by entering:

```
%token name1name2...
```

in the declarations section. (See Sections 3, 5, and 6 in this chapter for a detailed explanation.) Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start symbol
```

The end of the parser input is signaled by a special token, called the endmarker. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it then accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 6.3, below. Usually, the endmarker represents some reasonably obvious I/O status, such as the end of a file or the end of a record.

6.3 Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement; it can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces { and }. For example:

```
A :'('B')'
      {   hello( 1, "abc"); }
```

and:

```
XXX : YYY ZZZ
      {printf("a message\n");
       flag=25;}
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The dollar sign ($) is used as a signal to **yacc** in this context.

To return a value, the action normally sets the pseudo-variable *$$* to some value. For example, an action that does nothing but return the value 1 is:

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, ..., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is:

```
A : B C D ;
```

then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule:

```
expr : '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by:

```
expr : '(' expr ')' { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form:

```
A : B ;
```

do not need to have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. **yacc** permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the $ mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left.

Thus, in the rule:

```
A:B
  { $$=1; }
  C
  { x=$2; y=$3; }
  ;
```

the effect is to set *x* to 1, and to set *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by **yacc** by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. **yacc** actually treats the above example as if it had been written:

```
$ACT : /* empty */
        { $$=1; }
  ;

A  :B $ACT C
        { x=$2; y=$3; }
  ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call:

```
node(L, n1, n2)
```

creates a node with label L, with the descendants n1 and n2, and returns the index of the newly created node. Then, a parse tree can be built by supplying actions such as:

```
expr : expr'+'expr
         { $$=node('+', $1,$3); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks %{ and %}. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example:

```
%{intvariable=0; %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The **yacc** parser uses only names beginning in *yy*; the user should avoid such names.

In the examples shown, all the values are integers. A discussion of values of other types are found in a later section.

6.4 Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex()*. The function returns an integer, called the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by **yacc**, or chosen by the user. In either case, the *#define* mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name *DIGIT* has been defined in the declarations section of the **yacc** specification file. The relevant portion of the lexical analyzer might look like the example on the following page.

```
yylex(){
    extern int yylval;
    int c;
    ...
    c = getchar();
    ...
    switch( c ) {
     ...
    case '0':
    case '1':
     ...
    case '9':
        yylval = c-'0';
        return( DIGIT );
        ...
        }
    ...
```

The intent is to return a token number of *DIGIT*, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the program section of the specification file, the identifier, *DIGIT*, is defined as the token number associated with the token *DIGIT*.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of the token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should not be used naively.

As mentioned above, the token numbers maybe chosen by **yacc** or by the user. In the default situation, the numbers are chosen by **yacc**. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user. Hence, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is **lex**, discussed in a previous section. These lexical analyzers are designed to work in close harmony with **yacc** parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. **lex** can be easily used to produce quite complicated lexical analyzers, but there are some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

6.5 How the Parser Works

yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more understandable.

The parser produced by **yacc** consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and the lookahead token has not been read.

The machine has four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex*() to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF   shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not. In fact, the default action (represented by a .) is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action:

```
	.	reduce18
```

refers to grammar rule 18, while the action:

```
	IF	shift34
```

refers to state 34.

Suppose the rule being reduced is shown on the following page.

```
	A : x y z ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped is equal to the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a goto action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
	A	goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action turns back the clock in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side of the rule at that time. If the right hand side of the rule is empty, no states are popped off of the stack; the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable, *yylval*, is copied onto the value stack. After return from the user code, the reduction is carried out. When the goto action is done, the external variable, *yyval*, is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing; the error recovery (as opposed to the detection of error) will be discussed in a later section.

Consider the following example:

```
%token DING DONG DELL
%%
rhyme : sound place
   ;
sound : DING DONG
   ;
place : DELL
   ;
```

When **yacc** is invoked with the **-v** option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is given on the next page.

```
state 0
    $accept : _rhyme $end

    DING shift 3
    . error

    rhyme goto 1
    sound goto 2

state 1
    $accept : rhyme_$end

    $end accept
    . error

state 2
    rhyme : sound_place

    DELL shift 5
    . error

    place goto 4


state 3
    sound : DING_DONG

    DONG shift 6
    . error

state 4
    rhyme : sound place_ (1)

    . reduce 1

state 5
    place : DELL_ (3)

    . reduce 3

state 6
    sound : DING DONG_ (2)

    . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The underscore character (_) is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is:

```
        DING DONG DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is *shift 3*, so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is *shift 6*, so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
        sound : DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. It then checks the description of state 0, looking for a goto on *sound*, as shown below:

```
        sound goto 2
```

is obtained; meaning state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is *shift 5*, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme*, causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained; indicated by a *$end* in the file. When the endmarker is accepted, the action is called state 1. This successfully ends the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spent with this and the other simple examples will probably be repaid when problems arise in more complicated contexts.

6.6 Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule:

```
expr : expr '-' expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is:

```
expr - expr - expr
```

the rule allows this input to be structured as either:

```
( expr - expr ) - expr
```

or as:

```
expr - ( expr - expr )
```

(The first is called left association, the second is called right association).

yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given input such as:

```
expr - expr - expr
```

When the parser has read the second expr, the input that has been read:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying this rule, the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

```
- expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has read:

```
expr - expr
```

it could defer the immediate application of the rule, and continue reading the input until it comes across:

```
        expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving:

```
        expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read:

```
        expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a shift/reduce conflict. It may also happen that the parser has a choice of two legal reductions; this is called a reduce/reduce conflict. Note that there are never any shift/shift conflicts.

When there are shift/reduce or reduce/reduce conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguatingrule.

yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.

2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than **yacc** can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, **yacc** always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read, but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, **yacc** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an if-then-else construction:

```
stat : IF '(' cond ')' stat
  | IF '(' cond ')' stat ELSE stat
  ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the simple-if rule, and the second the if-else rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF (C1) {
    IF (C2) S1
    }
ELSE S2
```

or

```
IF (C1) {
    IF (C2) S1
    ELSE S2
    }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last *IF* immediately preceding the *ELSE*. In this example, consider the situation where the parser has seen:

```
IF( C1 ) IF( C2) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get:

```
IF ( C1 ) stat
```

and then read the remaining input:

```
        ELSE S2
```

and reduce:

```
        IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to groupings of the above listed input.

On the other hand, the *ELSE* may be shifted, *S2* read, and the right hand portion of:

```
        IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

to be reduced by the if-else rule to get:

```
        IF ( C1 ) stat
```

which can then be reduced by the simple-if rule. This leads to the second grouping of the previously listed input, which is usually desired.

Once again, the parser can do two valid things – there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs that have already been seen, such as:

```
        IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of **yacc** are best understood by examining the verbose (**−v**) option output file. The output corresponding to the above conflict state might be like the example shown on the next page.

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

    stat : IF ( cond ) stat_   (18)
    stat : IF ( cond ) stat_ELSE stat

    ELSE  shift 45
    .   reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which have been read. Thus in the example, in state 23 the parser has read input corresponding to:

```
IF ( cond ) stat
```

and the two grammar rules shown, are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line:

```
stat : IF ( cond ) stat ELSE_stat
```

Note that the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

Once again, notice that the numbers following shift commands refer to other states, while the numbers following reduce commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules that can be reduced. In most one states, there will be at most 1 reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the references previously shown can be checked; or the services of a knowledgeable user can be requested.

6.7 Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form:

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence of all the operators, and the associativity of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a **yacc** keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence. Thus:

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in FORTRAN. Thus:

```
A .LT. B .LT. C
```

is illegal in FORTRAN, and such an operator would be described with the keyword %nonassoc in **yacc**.

As an example of the behavior of these declarations, the description:

```
%right'='
%left '+''-'
%left '*''/'

%%

expr : expr '=' expr
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | NAME
    ;
```

might be used to structure the input:

```
a = b = c*d - e - f*g
```

as follows:

```
a= ( b = ( ((c*d)-e) - (f*g)))
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but have different precedences. An example is unary and binary '–'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. The %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+''-'
%left '*''/'

%%

expr : expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | '-' expr %prec '*'
    | NAME
    ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by the %token, as well.

The precedences and associativities are used by **yacc** to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.

2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by **yacc**. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially cookbook fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

6.8 Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser restarted after an error. A general class of algorithms to perform this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, **yacc** provides a simple, but reasonably general feature. The token name *error* is reserved for error handling. This name can be used in grammar rules; in effect, it suggests locations where errors are expected, and where recovery might take place. The parser pops its stack until it enters a state where the token *error* is legal. It then behaves as if the token *error* were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in an error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, the following rule:

```
        stat : error
```

would, in effect, mean that on a syntax error, the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error when there actually is no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as:

```
        stat : error ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any cleanup action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be:

```
input : error '\n' { printf( "Reenter line: "); } input
        { $$ = $4;}
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written:

```
input : error '\n'
        {yyerrok;
         printf( "Reenter last line: "); }
        input
        {$$ = $4; }
        ;
```

As mentioned above, the token seen immediately after the *error* symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like:

```
stat : error
       { resynch();
        yyerrok ;
        yyclearin ; }
       ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

6.9 The yacc Environment

When the user inputs a specification to **yacc**, the output is a file of C program, called *y.tab.c* on most systems. The function produced by **yacc** is called *yyparse*(); it is an integer valued function. When it is called, it repeatedly calls *yylex*, the lexical analyzer supplied by the user to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse*() returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse*() returns the value 0.

The user must provide a certain amount of environment for this parser, in order to obtain a working program. For example, as with every C program, a program called *main*() must be defined, that eventually calls *yyparse*() In addition, a routine called *yyerror*() prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **yacc**, a library has been provided with default versions of *main*() and *yyerror*(). The name of this library is system dependent; on many systems the library is accessed by a **-ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
   return(yyparse());
   }
```

and

```
#include <stdio.h>

yyerror(s) char *s; {
   fprintf(stderr, "%s\n", s);
   }
```

The argument to *yyerror*() is a string containing an error message, usually the string *syntax error*. The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main*() program is probably supplied by the user (to read arguments, etc.) the **yacc** library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

6.10 Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

6.11 InputStyle

It is difficult to provide rules with substantial actions and still have a readable specification file. Thus, try to be consistent with the following conventions when entering a specification file:

1. Use uppercase letters for token names, lowercase letters for nonterminal names. This rule helps you to know who to blame when things go wrong.

2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

3. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

4. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

5. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The examples in the text of this section follow this style (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the scope of action code.

6.12 Left Recursion

The algorithm used by the **yacc** parser encourages so called left recursive grammar rules of the form:

```
name : name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list : item
   | list ',' item
   ;
```

and

```
seq : item
   | seq item
   ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as:

```
seq : item
   | item seq
   ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion whenever possible.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq : /* empty */
   | seq item
   ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is asked to decide which empty sequence it has seen, when it has not seen enough to know!

6.13 Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
 int dflag;
%}
 ... other declarations ...

%%

prog  : decls stats
    ;

decls  : /* empty */
        {   dflag = 1; }
    | decls declaration
    ;

stats  : /* empty */
        {   dflag = 0; }
    | stats statement
    ;

  ... other rules ...
```

The flag dflag is now 0 when reading statements, and 1, when reading declarations, except for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of back door approach can be over done. Nevertheless, it represents a way of doing some things that are difficult to do otherwise.

6.14 Handling Reserved Words

Some programming languages permit the user to use words like *if*, which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance of "if" is a variable". The user can try this, but it is difficult. It is best that keywords be reserved; that is, be forbidden for use as variable names.

6.15 Simulating Error and Accept in Actions

The parsing actions of error and a cept can be simulated in an action by use of macros *YYACCEPT* and *YYERROR*. *YYACCEPT* causes *yyparse*() to return the value 0; *YYERROR* causes the parser to behave as if the current input symbol had been a syntax error; *yyer or*() is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

6.16 Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case, the digit may be 0 or negative. Consider:

```
sent  : adj noun verb adj noun
        { look at the sentence ... }
      ;



adj   : THE   { $$ = THE; }
      | YOUNG { $$ = YOUNG; }
      ...
      ;

noun  : DOG   { $$ = DOG; }
      | CRONE { if( $0 == YOUNG ){
                   printf( "what?\n" );
                   }
                $$ = CRONE;
              }
      ;
      ...
```

In the action following the word *CRONE*, a check is made that the preceding token shifted was not *YOUNG*. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times, this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

6.17 Supporting Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. **yacc** can also support values of other types, including structures. In addition, **yacc** keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The**yacc** value stack is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, **yacc** will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as **lint**(C), will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where **yacc** cannot easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union{
   body of union ...
   }
```

This declares the **yacc** value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If **yacc** was invoked with the **−d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable *YYSTYPE* to represent this union. Thus, the header file might also have said:

```
typedef union {
   body of union ...
   } YYSTYPE;
```

The header file must be included in the declaration section, by use of %{ and %}.

Once *YYSTYPE* is defined, the union member names must be associated with the various terminal and nonterminal names. The construction:

```
<name>
```

is used to indicate a union member name. If this follows one of the keywords, %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus:

```
%left <optype> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say:

```
%type <nodetype> expr stat
```

A couple of cases remain where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no predefined type. Similarly, reference to left context values (such as $0 – see the previous subsection) leaves **yacc** with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is:

```
rule : aaa { $<intval>$ = 3; } bbb
        { fun($<intval>2, $<other>0); }
    ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in a later section. The facilities in this subsection are not triggered until they are used. In particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the **yacc** value stack is used to hold *int*'s, as was true historically.

6.18 A Small Desk Calculator

The following example shows the complete **yacc** specification for a small desk calculator. The desk calculator has 26 registers, labeled *a* through *z*, and accepts arithmetic expressions made up of the operators +, –, *, /, % (mod operator), & (bitwise and), | (bitwise or), and = (assignment). If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules. This job is better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left'|'
%left'&'
%left'+''-'
%left'*''/''%'
%left UMINUS /* precedence for unary minus */

%% /*beginningofrules section */

list : /* empty */
   |list stat '\n'
   |list error '\n'
      {yyerrok;}
   ;

stat : expr
      {printf("%d\n", $1);}
   |  LETTER '=' expr
        {regs[$1]=$3;}
   ;
```

```
expr :'('expr')'
     { $$=$2;}
   | expr'+'expr
     { $$=$1+$3;}
   | expr'-'expr
     {$$=$1-$3;}
   | expr'*'expr
     { $$=$1*$3;}
   | expr'/'expr
     { $$=$1/$3;}
   | expr'%'expr
     { $$=$1% $3;}
   | expr'&'expr
     { $$=$1&$3;}
   | expr'|'expr
     { $$=$1|$3;}
   | '-'expr %prec UMINUS
     { $$=-$2;}
   | LETTER
     { $$=regs[$1]; }
   | number
   ;


     number : DIGIT
          { $$=$1; base=($1==0) ? 8 : 10; }
        | number DIGIT
          { $$=base * $1 + $2; }
        ;

     %%   /* start of programs */
```

```
yylex() {      /* lexical analysis routine */
               /* returns LETTER for a lowercase letter, */
               /* yylval = 0 through 25 */
               /* return DIGIT for a digit, */
               /* yylval = 0 through 9 */
               /* all other characters */
               /* are returned immediately */

   int c;

   while( (c=getchar()) == ' ') {/* skip blanks */}

               /* c is now nonblank */

   if( islower( c ) ) {
      yylval = c - 'a';
      return ( LETTER );
      }
   if( isdigit( c ) ) {
      yylval = c - '0';
      return( DIGIT );
      }
   return( c );
   }
```

6.19 yacc Input Syntax

This section has a description of the **yacc** input syntax, as a **yacc** specification. Context dependencies, etc., are not considered. Ironically, the **yacc** input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decides whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token *C_IDENTIFIER*. Otherwise, it returns *IDENTIFIER*. Literals (quoted strings) are also returned as *IDENTIFIER*, but never as part of a *C_IDENTIFIER*.

```
     /* grammar for the input to yacc */

    /* basic entities */
%token IDENTIFIER      /* includes identifiers and literals */
%token C_IDENTIFIER /* identifier followed by colon    */
%token NUMBER          /* [0-9]+  */

    /* reserved words: %type => TYPE, %left => LEFT, etc. */

%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token MARK  /* the %% mark */
%token LCURL  /* the %{ mark */
%token RCURL  /* the %} mark */

    /* ascii character literals stand for themselves */

%start spec

%%

spec   : defs MARK rules tail
    ;

tail   :MARK  {Eat up the rest of the file}
    | /* empty: the second MARK is optional */
    ;

defs   : /* empty */
    | defs def
    ;

def   : START IDENTIFIER
    | UNION { Copy union definition to output}
    | LCURL { Copy C code to output file } RCURL
    | ndefs rword tag nlist
    ;

rword  : TOKEN
    | LEFT
    | RIGHT
    | NONASSOC
    | TYPE
    ;
```

```
tag   : /* empty: union tag is optional */
    | '<' IDENTIFIER '>'
    ;

nlist  : nmno
    | nlist nmno
    | nlist ',' nmno
    ;

nmno  : IDENTIFIER      /* Literal illegal with %type */
    | IDENTIFIER NUMBER /* Illegal with %type */
    ;

    /* rules section */

rules  : C_IDENTIFIER rbody prec
    | rules rule
    ;

rule  : C_IDENTIFIER rbody prec
    | '|' rbody prec
    ;

rbody  : /* empty */
    | rbody IDENTIFIER
    | rbody act
    ;

act   : '{' { Copy action, translate $$, etc. } '}'
    ;

prec  : /* empty */
    | PREC IDENTIFIER
    | PREC IDENTIFIER act
    | prec ';'
    ;
```

6.20 An Advanced Example

This section shows an example of a grammar using some of the advanced features discussed in earlier sections. The desk calculator example is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, –, *, /, unary–, and = (assignment), and has 26 floating point variables, *a* through *z*. Moreover, it also understands intervals, written as:

```
(x,y)
```

where *x* is less than or equal to *y*. There are 26 interval-valued variables *A* through *Z* that may also be used. Assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of **yacc** and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as double precision values. This structure is given a type name, *INTERVAL*, by using **typedef**. The **yacc** value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of *YYERROR* to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of **yacc** is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g., scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through **yacc**: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

```
2.5 + ( 3.5 - 4. )
```

and

```
2.5 + ( 3.5 , 4. )
```

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the comma (,) is read. By this time, 2.5 is finished, and the parser cannot go back and change. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is circumvented by having two rules for each binary interval-valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. However, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file. In this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine **atof(S)** is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and leading to error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
   double lo, hi;
   } INTERVAL;

INTERVAL vmul(), vdiv();

double  atof();

double  dreg[26];
INTERVAL vreg[26];

%}

%start  lines

%union {
   int ival;
   double dval;
   INTERVAL vval;
   }

%token <ival> DREG VREG  /* indices into dreg, vreg arrays */

%token <dval> CONST    /* floatingpoint constant */

%type  <dval> dexp     /* expression */

%type  <vval> vexp     /* intervalexpression */
```

```
   /* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS    /* precedence for unary minus */

%%

lines :/* empty */
   |lines line
   ;

line  : dexp '\n'
       { printf( "%15.8f\n", $1 ); }
   |vexp '\n'
       { printf( "(%15.8f, %15.8f )\n", $1.lo, $1.hi); }
   |DREG '=' dexp '\n'
       { dreg[$1] = $3; }
   |VREG '=' vexp '\n'
       { vreg[$1] =$3; }
   |error '\n'
       { yyerrok; }
   ;

dexp  : CONST
   |DREG
       {$$=dreg[$1]; }
   |dexp '+' dexp
       {$$=$1+$3;}
   |dexp '-' dexp
       {$$=$1-$3;}
   |dexp '*' dexp
       {$$=$1*$3;}
   |dexp '/' dexp
       {$$=$1/$3;}
   |'-' dexp %prec UMINUS
       {$$=-$2; }
   |'(' dexp ')'
       {$$=$2;}
   ;
```

```
vexp  : dexp
        { $$.hi= $$.lo =$1; }
    | '(' dexp ',' dexp ')'
        {
         $$.lo = $2;
         $$.hi = $4;
         if( $$.lo > $$.hi ){
             printf("intervalout of order\n");
             YYERROR;
             }
           }
    | VREG
        { $$ = vreg[$1]; }
    | vexp '+' vexp
        { $$.hi = $1.hi + $3.hi;
          $$.lo = $1.lo + $3.lo; }
    | dexp '+' vexp
        { $$.hi = $1 + $3.hi;
          $$.lo = $1 + $3.lo; }
    | vexp '-' vexp
        { $$.hi = $1.hi - $3.lo;
          $$.lo = $1.lo - $3.hi;}
    | dexp '-' vexp
        { $$.hi = $1 - $3.lo;
          $$.lo = $1 -$3.hi;}
    | vexp '*' vexp
        { $$ = vmul( $1.lo, $1.hi, $3 ); }
    | dexp '*' vexp
        { $$= vmul( $1, $1, $3 ); }
    | vexp '/' vexp
        {if (dcheck( $3 ) ) YYERROR;
         $$= vdiv( $1.lo, $1.hi, $3 ); }
    | dexp '/' vexp
        {if(dcheck($3 ) ) YYERROR;
         $$ =vdiv( $1, $1, $3 ); }
    | '-' vexp %prec UMINUS
        { $$.hi= -$2.lo; $$.lo = -$2.hi; }
    | '(' vexp ')'
        {    $$ = $2; }
    ;

%%

# define BSZ 50 /* buffer size for fpnumbers */

    /* lexical analysis */
```

```
yylex(){
   register c;
            {/* skip over blanks */}
   while( ( c = getchar() ) == ' ' )

   if ( isupper(c) ){
      yylval.ival = c - 'A';
      return(VREG);
      }
   if ( islower(c) ){
      yylval.ival = c - 'a';
      return(DREG);
      }

   if( isdigit( c ) || c=='.' ){
      /* gobble up digits, points, exponents */

      char buf[BSZ+1], *cp = buf;
      int dot=0, exp=0;

      for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

         *cp = c;
         if ( isdigit(c) ) continue;
         if ( c == '.' ) {
            if ( dot++ || exp ) return('.');
                                 /* above causes syntax error */
            continue;
            }

         if (c == 'e') {
            if ( exp++ ) return('e');
                                 /* above causes syntax error */
            continue;
            }

         /* end of number */
         break;
         }
      *cp= '\0';
      if( (cp-buf) >= BSZ)
               printf( "constant too long: truncated\n");
      else ungetc( c, stdin );
            /* above pushes back last char read */
      yylval.dval= atof (buf);
      return( CONST );
      }
   return( c );
   }
```

```
INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
   /* returns the smallest interval containing a, b, c, and d */
   /* used by *, / routines */
   INTERVAL v;

   if( a>b ) { v.hi = a; v.lo = b; }
   else{ v.hi = b; v.lo = a; }

   if( c>d ) {
      if ( c>v.hi ) v.hi = c;
      if ( d<v.lo ) v.lo = d;
      }
   else {
      if ( d>v.hi ) v.hi = d;
      if ( c<v.lo ) v.lo = c;
      }
   return( v );
   }

INTERVAL vmul( a, b, v ) double a, b; INTERVAL v; {
   return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
   }

dcheck( v ) INTERVAL v; {
   if( v.hi >= 0. && v.lo <= 0. ){
      printf( "divisor interval contains 0.\n" );
      return(1);
      }
   return(0);
   }

INTERVAL vdiv( a, b, v ) double a, b; INTERVAL v; {
   return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
   }
```

6.21 Old Features

This section mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotation marks (").

2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or underscores, the type number of the literal is defined, just as if the literal did not have the quotation marks around it. Otherwise, it is difficult to find the value for such

literals. The use of multicharacter literals is likely to mislead those unfamiliar with **yacc**, since it suggests that **yacc** is doing a job that must be actually done by the lexical analyzer.

3. Most places where '%' is legal, backslash (\) may be used. In particular, the double backslash (\\) is the same as %%, *\left* the same as *%left*, etc.

4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. Actions may also have the form

   ={...}

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Replace this Page
with Tab Marked:

Signals

Chapter 7
Using Signals

7.1 Introduction

This chapter explains how to use C library functions to process signals sent to a program by the XENIX system. A signal is the system's response to an unusual condition that occurs during execution of a program such as a user pressing the INTERRUPT key or the system detecting an illegal operation. A signal interrupts normal execution of the program and initiates an action such as terminating the program or displaying an error message.

The **signal**(S) function of the standard C library lets a program define the action of a signal. The function can be used to disable a signal to prevent it from affecting the program. It can also be used to give a signal a user-defined action.

The **signal** function is often used with the **setjmp**(S) and **longjmp** (see **setime**(S)) functions to redefine and reshape the action of a signal. These functions allow programs to save and restore the execution state of a program, giving a program a means to jump from one state of execution to another without a complex assembly language interface.

To use the **signal** function, you must add the line

```
#include <signal.h>
```

to the beginning of the program. The *signal.h* file defines the various manifest constants used as arguments by the function. To use the **setjmp** and **longjmp** functions you must add the line

```
#include <setjmp.h>
```

to the beginning of the program. The *setjmp.h* file contains the declaration for the type **jmp_buf**, a template for saving a program's current execution state.

7.2 Using the signal Function

The *signal*() function changes the action of a signal from its current action to a given action. The function has the form

signal (*sigtype*,*ptr*);

where *sigtype* is an integer or a manifest constant that defines the signal to be changed, and *ptr* is a pointer to the function defining the new action or a manifest constant giving a predefined action. The function always returns a pointer value. This pointer defines the signal's previous action and may be used in subsequent calls to restore the signal to its previous value.

The *ptr* may be "SIG_IGN" to indicate no action (ignore the signal) or "SIG_DFL" to indicate the default action. The *sigtype* may be "SIGINT" for interrupt signal, caused by pressing the INTERRUPT key, "SIGQUIT" for quit signal, caused by pressing the QUIT key, or "SIGHUP" for hangup signal, caused by hanging up the line when connected to the system by modem. (Other constants for other signals are given in **signal**(S) in the XENIX *Reference*.)

For example, the function call

```
        signal(SIGINT, SIG_IGN);
```

changes the action of the interrupt signal to no action. The signal will have no effect on the program. The default action is usually to terminate the program.

The following sections show how to use the **signal** function to disable, change, and restore signals.

7.2.1 Disabling a Signal

You can disable a signal, i.e., prevent it from affecting a program, by using the "SIG_IGN" constant with **signal**. The function call has the form

```
        signal (sigtype, SIG_IGN);
```

where *sigtype* is the manifest constant of the signal you wish to disable. For example, the function call

```
        signal(SIGINT, SIG_IGN);
```

disables the interrupt signal.

The function call is typically used to prevent a signal from terminating a program executing in the background (e.g., a child process that is not using the terminal for input or output). The system passes signals generated from keystrokes at a terminal to all programs that have been invoked from that terminal. This means that pressing the INTERRUPT key to stop a program running in the foreground will also stop a program running in the background if it has not disabled that signal. For example, in the following program fragment, **signal** is used to disable the interrupt signal for the child.

```
#include <signal.h>

main()
{

        if (fork() ==0) {
                signal(SIGINT, SIG_IGN);
                /* Child process. */

        }

/* Parent process. */

}
```

This call does not affect the parent process which continues to receive interrupts as before. Note that if the parent process is interrupted, the child process continues to execute until it reaches its normal end.

7.2.2 Restoring a Signal's Default Action

You can restore a signal to its default action by using the "SIG_DFL" constant with **signal**. The function call has the form

```
signal (sigtype, SIGDFL);
```

where *sigtype* is the manifest constant defining the signal you wish to restore. For example, the function call

```
signal (SIGINT, SIG_DFL);
```

restores the interrupt signal to its default action.

The function call is typically used to restore a signal after it has been temporarily disabled to keep it from interrupting critical operations. For example, in the following program fragment the second call to **signal** restores the signal to its default action.

```
#include <signal.h>
#include <stdio.h>

main ()
{
        FILE *fp;
        charrecord[BUFSIZE], filename[100];

        signal (SIGINT, SIG_IGN);
        fp=fopen(filename, "a");
        fwrite(record, BUF, 1, fp);
        signal (SIGINT, SIG_DFL);

}
```

In this example, the interrupt signal is ignored while a record is written to the file given by "fp".

7.2.3 Catching a Signal

You can catch a signal and define your own action for it by providing a function that defines the new action and giving the function as an argument to **signal**. The function call has the form

signal (*sigtype, newptr*);

where *sigtype* is the manifest constant defining the signal to be caught, and *newptr* is a pointer to the function defining the new action. For example, the function call

```
signal(SIGINT, catch);
```

changes the action of the interrupt signal to the action defined by the function named *catch*().

The function call is typically used to let a program do additional processing before terminating. In the following program fragment, the function *catch*() defines the new action for the interrupt signal.

```
#include <signal.h>

main ()
{
        int catch ();

        printf("Press INTERRUPT key to stop.\n");
        signal (SIGINT, catch);
        while [illegible] {
                /*Body*/
        }
}

catch ()
{
        printf("Program terminated.\n");
        exit(1);
}
```

The *catch*() function prints the message "Program terminated" before stopping the program with the **exit**(S) function.

A program may redefine the action of a signal at any time. Thus, many programs define different actions for different conditions. For example, in the following program fragment the action of the interrupt signal depends on the return value of a function named *keytest*.

```
#include <signal.h>

main ()
{
        int catch1 (), catch2 ();

        if (keytest() == 1)
                signal(SIGINT, catch1);
        else
                signal(SIGINT, catch2);

}
```

Later the program may change the signal to the other action or even a third action.

When using a function pointer in the **signal** call, you must make sure that the function name is defined before the call. In the program fragment shown above, *catch1* and *catch2* are explicitly declared at the beginning of the main program function. Their formal definitions are assumed to appear after the **signal** call.

7.2.4 Restoring a Signal

You can restore a signal to its previous value by saving the return value of a **signal** call, then using this value in a subsequent call. The function call has the form:

```
signal (sigtype, oldptr);
```

where *sigtype* is the manifest constant defining the signal to be restored and *oldptr* is the pointer value returned by a previous **signal** call.

The function call is typically used to restore a signal when its previous action may be one of many possible actions. For example, in the following program fragment the previous action depends solely on the return value of a function *keytest*.

```
#include <signal.h>

main ()
{
        int catch1(), catch2();
        int (*savesig)();

        if (keytest()== 1)
                signal(SIGINT, catch1);
        else
                signal(SIGINT, catch2);

        savesig= signal (SIGINT, SIG_IGN);
        compute();
        signal(SIGINT, savesig);

}
```

In this example, the old pointer is saved in the variable "savesig". This value is restored after the function *compute* returns.

7.2.5 Program Example

This section shows how to use the **signal** function to create a modifed version of the **system**(S) function. In this version, **system** disables all interrupts in the parent process until the child process has completed its operation. It then restores the signals to their previous actions.

```
#include <stdio.h>
#include <signal.h>

system(s)      /* run command strings */
char *s;
{
        int status, pid, w;
        register int (*istat)(), (*qstat)();

        if ((pid = fork()) == 0) {
                execl("/bin/sh", "sh", "-c", s, NULL);
                exit(127);
        }
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        while ((w = wait(&status)) != pid && w != -1)
                ;
        if (w == -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        return(status);
}
```

Note that the parent uses the **while** statement to wait until the child's process ID "pid" is returned by **wait**(S). If **wait** returns the error code "-1" no more child processes are left, so the parent returns the error code as its own status.

7.3 Catching Several Signals

There are many more signals besides SIGINT, SIGQUIT, and SIGHUP. See the **signal**(S) manual page for a complete list. In the following program fragment, all signals are caught by the same function. This function makes use of the specific signal number which is passed as a parameter by the system.

```
#include <signal.h>

main()
{
        int i;
        int catch();

        for (i = 1; i <= NSIG; ++i)
                signal(i, catch);
        /*
         * Body
         */
}

catch(sig)
int sig;
{
        signal(sig, SIG_IGN);
        if (sig != SIGINT && sig != SIGQUIT && sig != SIGHUP)
                printf("Oh, oh.  Signal %d was received.\n", sig);
        unlink(tmpfile);
        exit(1);
}
```

The constant NSIG, the total number of signals, is defined in the file signal.h.

Note that the first action of the above catch function is to ignore the specific signal that was caught. This is necessary because the system automatically resets a caught signal to its default action.

7.4 Controlling Execution With Signals

Signals do not need to be used solely as a means of immediately terminating a program. Many signals can be redefined to delay their actions or even cause actions that terminate a portion of a program without terminating the entire program. The following sections describe ways that signals can be caught and used to provide control of a program.

7.4.1 Delaying a Signal's Action

You can delay the action of a signal by catching the signal and redefining its action to be nothing more than setting a globally-defined flag. Such a signal does nothing to the current execution of the program. Instead, the program continues uninterrupted until it can test the flag to see if a signal has been received. It can then respond according to the value of the flag.

The key to a delayed signal is that functions used to catch signals return execution to the exact point at which the program was interrupted. If the function returns normally the program continues execution just as if no signal occurred.

Delaying a signal is especially useful in programs that must not be stopped at an arbitrary point. If, for example, a program updates a linked list, the action of a signal can be delayed to prevent the signal from interrupting the update and destroying the list. For example, in the following program fragment the function *delay*() used to catch the interrupt signal sets the globally-defined flag "sigflag" and returns immediately to the point of interruption.

```
#include <signal.h>

int sigflag;

main ()
{
        int delay ();
        int (*savesig)();

        signal(SIGINT, delay); /* Delay the signal. */
        updatelist();
        savesig = signal(SIGINT, SIG_IGN); /* Disable the signal. */
        if (sigflag)
                /* Process delayed signals if any. */

}

delay ()
{
        signal(SIGINT, delay);
        sigflag=1;
}
```

In this example, if the signal is received while *updatelist* is executing, it is delayed until after *updatelist* returns. Note that the interrupt signal is disabled before processing the delayed signal to prevent a change to "sigflag" when it is being tested.

The first action of the delay function was to recatch the interrupt signal. This is necessary because the system resets caught signals to their default action which is normally immediate termination.

7.4.2 Using Delayed Signals With System Functions

When a delayed signal is used to interrupt the execution of a XENIX system function, such as *read*() or *wait*(), the system forces the function to stop and return an error code. This action, unlike actions taken during execution of other functions, causes all processing performed by the system function to be discarded. A serious error can occur if a program interprets a system function error caused by delayed signals as a normal error. For example, if a program receives a signal when reading the terminal, all characters read before the interruption are lost, making it appear as though no characters were typed.

Whenever a program intends to use delayed signals during calls to system functions, the program should include a check of the function return values to ensure that an error was not caused by an interruption. In the following program fragment, the program checks the current value of the interrupt flag "intflag" to make sure that the value EOF returned by *getchar* actually indicates the end of the file.

```
if (getchar() == EOF)
        if (intflag)
                /* EOF caused by interrupt */
        else
                /* true end-of-file */
```

7.4.3 Using Signals in Interactive Programs

Signals can be used in interactive programs to control the execution of the program's various commands and operations. For example, a signal may be used in a text editor to interrupt the current operation (e.g., displaying a file) and return the program to a previous operation (e.g., waiting for a command).

To provide this control, the function that redefines the signal's action must be able to return execution of the program to a meaningful location, not just the point of interruption. The standard C library provides two functions to do this: **setjmp** and **longjmp**. The **setjmp** function saves a copy of a program's execution state. The **longjmp** function changes the current execution state to a previously saved state. The functions cause a program to continue execution at an old location with old register values and status as if no operations had been performed between the time the state was saved and the time it was restored.

The **setjmp** function has the form

```
setjmp (buffer);
```

where *buffer* is the variable to receive the execution state. It must be explicitly declared with type **jmp_buf** before it is used in the call. For example, in the following program fragment **setjmp** copies the execution of the program to the variable "oldstate" defined with type **jmp_buf**.

```
jmp_buf oldstate;

setjmp(oldstate);
```

Note that after a **setjmp** call, the *buffer* variable contains values for the program counter, the data and address registers, and the process status. These values must not be modified in any way.

The **longjmp** function has the form

longjmp (*buffer*);

where *buffer* is the variable containing the execution state. It must contain values previously saved with a **setjmp** function. The function copies the values in the *buffer* variable to the program counter, data and address registers, and the process status table. Execution continues as if it had just returned from the **setjmp** function which saved the previous execution state. For example, in the following program fragment **setjmp** saves the execution state of the program at the location just before the main processing loop and **longjmp** restores it on an interrupt signal.

```
#include <signal.h>
#include <setjmp.h>

jmp_buf sjbuf;

main()
{
        int onintr();

        setjmp(sjbuf);
        signal(SIGINT, onintr);

        /* main processing loop */
}

onintr ()
{
        printf("\nInterrupt\n");
        longjmp(sjbuf);
}
```

In this example, the action of the interrupt signal as defined by *onintr* is to print the message "Interrupt" and restore the old execution state. When an interrupt signal is received in the main processing loop, execution passes to *onintr* which prints the message, then passes execution back to the main program function, making it appear as though control is returning from the **setjmp** function.

7.5 Using Signals in Multiple Processes

The XENIX system passes all signals generated at a given terminal to all programs invoked at that terminal. This means that a program has potential access to a signal even if that program is executing in the background or as a child to some other program. The following sections explain how signals may be used in multiple processes.

7.5.1 Protecting Background Processes

Any program that has been invoked using the shell's background symbol (&) is executed as a background process. Such programs usually do not use the terminal for input or output, and complete their tasks silently. Since these programs do not need additional input, the shell automatically disables the signals before executing the program. This means signals generated at the terminal do not affect execution of the program. This is how the shell protects the program from signals intended for other programs invoked from the same terminal.

In some cases, a program that has been invoked as a background process may also attempt to catch its own signals. If it succeeds, the protection from interruption given to it by the shell is defeated, and signals intended for other programs will interrupt the program. To prevent this, any program which is intended to be executed as a background process, should test the current state of a signal before redefining its action. A program should redefine a signal only if the signal has not been disabled. For example, in the following program fragment the action of the interrupt signal is changed only if the signal is not currently being ignored.

```
#include <signal.h>

main()
{
        int catch();

        if (signal(SIGINT, SIG_IGN) != SIG_IGN)
                signal(SIGINT, catch);

        /* Program body. */
}
```

This step lets a program continue to ignore signals if it is already doing so, and change the signal if it is not.

7.5.2 Protecting Parent Processes

A program can create and wait for a child process that catches its own signals if and only if the program protects itself by disabling all signals before calling the **wait** function. By disabling the signals, the parent process prevents signals intended for the child processes from terminating its call to **wait**. This prevents serious errors that may result if the parent process continues execution before the child processes are finished.

For example, in the following program fragment the interrupt signal is disabled in the parent process immediately after the child is created.

```
#include <signal.h>

main ()
{
        int (*saveintr)();

        if (fork ()==0)
                execl( ... );

        saveintr= signal (SIGINT, SIG_IGN);
        wait( &status);
        signal (SIGINT, saveintr);
}
```

The signal's action is restored after the **wait** function returns normal control to the parent.

Replace this Page
with Tab Marked:

System
Resources

Chapter 8

Using System Resources

8.1 Introduction

This chapter describes the standard 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.

In particular, this chapter explains how to

- Allocate memory for dynamically required storage
- Lock a file to ensure exclusive use by a program
- Use semaphores to control access to a resource
- Share data space to allow interaction between programs

8.2 Allocating Space

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 space, then free this space after it has finished using it.

There are four memory allocation functions: *malloc*, *calloc*, *realloc*, and *free*. The *malloc* and *calloc* functions are used to allocate space for the first time. The functions allocate a given number of bytes and return a pointer to the new space. The *realloc* function reallocates an existing space, allowing it to be used in a different way. The *free* function returns allocated space to the system.

8.2.1 Allocating Space for a Variable

The *malloc* function allocates space for a variable containing a given number of bytes. The function call has the form:

```
malloc (size)
```

where *size* is an unsigned number which gives the number of bytes to be allocated. For example, the function call

```
table = malloc (4)
```

allocates four bytes or storage. The function normally returns a pointer to the starting address of the allocated space, but will return the null pointer value if there is not enough space to allocate.

The function is typically used to allocate storage for a group of strings that vary in length. For example, in the following program fragment *malloc* is used to allocate space for ten different strings, each of different length.

```
        inti;
        char *temp, *strings[10];
        unsigned isize;

        for(i=0;i<10; i++) {
                scanf("%s", temp);
                isize = strlen(temp);
                strings[i] = malloc(isize);
                }
```

In this 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.

8.2.2 Allocating Space 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 form:

calloc (*n*, *size*)

where *n* is the number of elements in the array, and *size* is the number of bytes in each element. 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 memory. For example, the function call

```
        table = calloc (10,4)
```

allocates sufficient space for a 10 element array. Each element has 4 bytes.

The function is typically used in programs which 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.

```
        inti;
        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.

8.2.3 Reallocating Space

The *realloc* function reallocates the space at a given address without changing the contents of the memory space. The function call has the form:

realloc (*ptr, size*)

where *ptr* is a pointer to the starting address of the space to be reallocated, and *size* is an unsigned number giving the new size in bytes of the reallocated space. 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.

This function is typically used to keep storage as compact as possible. For example, in the following program fragment *realloc* is used 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 space from its current length to the length given by "i*4".

8.2.4 Freeing Unused Space

The *free* function frees unused memory space that had been previously allocated by a *malloc, calloc,* or *realloc* function call. The function call has the form:

free (*ptr*)

where *ptr* is the pointer to the starting address of the space to be freed. This pointer must be the return value of a *malloc, calloc,* or *realloc* function.

The function is used exclusively to free space which is no longer used or to free space to be used for other purposes. For example, in the following program fragment *free* frees the allocated space pointed to by "strings" if the first element is equal to zero.

```
main ()
{
char *table;

if ( table[0]== -1 )
      free (table);
```

8.3 Locking Files

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 one file locking function, the *locking* function. This function locks any given section of a file, preventing all other processes which wish to use the section from gaining access. A process may lock the entire file or only a small portion. In any case, only the locked section is protected; all other sections may be accessed by other processes as usual.

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. Before a file can be locked, however, it must be prepared using the *open* and *lseek* functions described in Chapter 2, "Using the Standard I/O Functions." To use the *locking* function, you must add the line

```
#include <sys/locking.h>
```

to the beginning of the program. The file *sys/locking.h* contains definitions for the modes used with the function.

8.3.1 Preparing a File for Locking

Before a file can be locked, it must first be opened using the *open* function, then properly positioned by using the *lseek* function to move 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 may 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);
```

8.3.2 Locking a File

The *locking* function locks one or more bytes of a given file. The function call has the form:

locking (*filedes, mode, size*)

where *filedes* is the file descriptor of the file to be locked, *mode* is an integer value which 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* may be "LOCK" for locking the given bytes, or "UNLOCK" 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, LOCK, 100L);
```

The function normally returns the number of bytes locked, but will return −1 if it encounters an error.

8.3.3 Program Example

This section shows 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 the file *data* which is opened for reading and writing. The locked portion of the file is accessed, then *locking* is used again to unlock the file.

```
#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 */

        }
}
```

8.4 Using Semaphores

The standard C library provides a group of functions, called the semaphore functions, which may be used to control the access to a given system resource. These functions create, open, and request control of "semaphores." 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. A process that wishes to take control of a semaphore away from another process must wait until that process relinquishes control. Semaphores can be used 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.

There are five semaphore functions: *creatsem*, *opensem*, *waitsem*, *nbwaitsem*, and *sigsem*. The *creatsem* function creates a semaphore. The semaphore may then be opened and used by other processes. A process can open a semaphore with the *opensem* function and request control of a semaphore with the *waitsem* or *nbwaitsem* function. Once a process has control of a semaphore it can carry out tasks using the given resource. All other processes must wait. When a process has finished accessing the resource, it can relinquish control of the semaphore with the *sigsem* function. This lets other processes get control of the semaphore and use the corresponding resource.

8.4.1 Creating a Semaphore

The *creatsem* function creates a semaphore, returning a semaphore number which may be used in subsequent semaphore functions. The function call has the form:

creatsem (*sem_name*, *mode*)

where *sem_name* is a character pointer to the name of the semaphore, and *mode* is an integer value which defines the access mode of the semaphore. Semaphore names have the same syntax as regular file names. The names must be unique. The function normally returns an integer semaphore number which may be 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.

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 preceding with its tasks.

```
main()
{
int tty1;
FILE fttyl;

tty1 = creatsem("tty1", 0777);
ftty1 = fopen("/dev/tty01", "w");
        /* Program body. */
}
```

Note that *fopen* is used immediately after *creatsem* to open the file */dev/tty01* 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 may 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 may be accessed only by using the *waitsem*, *nbwaitsem*, or *sigsem* functions. The *creatsem* function may 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.

8.4.2 Opening a Semaphore

The *opensem* function opens an existing semaphore for use by the given process. The function call has the 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 may 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.

The function is typically used by a process just 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);
```

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.

A semaphore must not be opened 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.

8.4.3 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 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 −1 if it encounters an error such as requesting a semaphore that does not exist or requesting a semaphore that is locked to a dead process.

The 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 fttyl;

waitsem(tty1);
fprintf(fttyl, "Changing tty driver\n");
```

The function waits until current controlling process relinquishes control of the semaphore before returning to the next statement.

8.4.4 Checking the Status of a Semaphore

The *nbwaitsem* function checks the current status of a semaphore. 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 form:

nbwaitsem (*sem_num*)

where *sem_num* is the semaphore number of the semaphore to be checked. The function returns −1 if it encounters an error such as requesting a semaphore that does not exist. The function also returns −1 if the process controlling the requested semaphore terminates without relinquishing control of the semaphore.

The function is typically used in place of *waitsem* to take control of a semaphore.

8.4.5 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 form:

```
sigsem (sem_num)
```

where *sem_num* is the semaphore number of the semaphore to relinquish. The semaphore must have been previously created or opened by the process. Furthermore, the process must have been previously taken control of the semaphore with the *waitsem* or *nbwaitsem* function. The function returns −1 if it encounters an error such as trying to take control of a semaphore that does not exist.

The function is typically used after a process has finished accessing the corresponding device or system resource. This allows waiting processes to take control. For example, in the following program fragment *sigsem* signals the end of control of the semaphore "tty1".

```
main()
{
int tty1;
FILE temp, ftty1;

waitsem(tty1);
while ((c=fgetc(temp)) !=EOF)
        fputc(c, ftty1);
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 signal the end of the control before terminating. In such a case, the semaphore must be reset by using the *creatsem* function.

8.4.6 Program Example

This section shows how to use the semaphore functions to control the access of a system resource. The following program creates a child process and a semaphore. The parent process sleeps while the child process decrements the semaphore to 0. When this happens, the child process exits and the parent process returns a completion message, deletes the semaphore and exits. Although the program performs no meaningful work, it clearly illustrates the use of semaphores.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include < sys/sem.h>

#define      KEY(key_t) 10
#define COUNT 2
#define BUMP 6

main()
{
        int semid;
        ushort vals[COUNT];
        struct sembuf sops[COUNT];
        int i;

        for (i=0; i < COUNT; i++)
                vals[i]=0;

        if ((semid = semget(KEY, COUNT, 0666 |IPC_CREAT)) == -1)
        {
                perror("semid");
                exit(-1);
        }

        if(semctl(semid, 0, SETALL, vals) == -1)
        {
                perror("semctl");
                semdel(semid);
                exit(-1);
        }
        sops[0].sem_num=0;
        sops[0].sem_flg=0;

        if (!fork())
        {
                /* CHILD*/d
                sleep(2);
                sops[0].sem_op = -1;
                for (i=0; i < BUMP; i++)
                {
                        puts("BUMP");
                        if(semop(semid,SOPS, 1) == -1)
                        perror("SEMOP");
                }
                exit(1);
```

```
          }

          /* PARENT */
          sops[0].sem_op= BUMP;
          if (semop(semid, sops, 1)== -1)
          {
                  perror("semop");
                  semdel(semid);
                  exit(-1);
          }

          sops[0].sem_op=0;
          puts("Entering second semop");
          if (semop(semid, sops, 1)== -1)
          {
                  perror("semop");
                  semdel(semid);
                  exit(-1);
          }
          puts("Past second semop");
          semdel(semid);
  }

  semdel(x)
  int x;
  {
          if (semctl(x, 0, IPC_RMID, o)== -1)
          {
                  perror("IPC_RMID");
                  exit(-1);
          }
  }
```

The program contains a number of global variables. The array "semf" contains the semaphore name. The name is used by the *creatsem* and *opensem* functions. The variable "sem_num" is the semaphore number. This is the value returned by *creatsem* and *opensem* and eventually used 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, it 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 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.

8.5 Using Shared Data

Shared memory is a method by which one process shares its allocated data space with another. Shared memory allows processes to pool information in a central location and directly access that information without the burden of creating pipes or temporary files.

The extended C library provides several functions to access and control shared memory. The *sdget* function creates and/or adds a shared memory segment to a given process's data space. To access a segment, a process must signal its intention with the *sdenter* function. Once a segment has completed its access, it can signal that it is finished using the the segment with the *sdleave* function. The *sdfree* function is used to remove a segment from a process's data space. The *sdgetv* and *sdwaitv* functions are used to synchronize processes when several are accessing the segment at the same time.

To use the shared data functions, you must add the line

```
#include <sd.h>
```

at the beginning of the program. The *sd.h* file contains definitions for the manifest constants and other macros used by the functions.

8.5.1 Creating a Shared Data Segment

The *sdget* function creates a shared data segment for the current process and attaches the segment to the process's data space. The function call has the form:

sdget (*path*, *flag*, *size*, *mode*)

where *path* is a character pointer to a valid pathname, *flag* is an integer value which defines how the segment should be created, *size* is a long integer value which defines the size in bytes of the segment to be created, and *mode* is an integer value which defines the access permissions to be given to the segment. The *flag* may 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 may also use SD_UNLOCK for allowing simultaneous access by multiple processes. The values can be combined by logically ORing them. The function returns the address of the segment if it has been successfully created. Otherwise, the function returns -1.

The function is typically used by just one process to create a segment that it will share with several other processes. For example, in the following fragment, the program uses *sdget* to create a segment and attach it for reading and writing. The address of the new segment is assigned to *shared*.

```
#include <sd.h>

main ()
{
char *shared;

shared=sdget("/tmp/share", SD_CREAT|SD_WRITE, 512L, 0777);
}
```

When the segment is created, the size "512" 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 that the SD_UNLOCK flag used on systems without hardware support for shared data may severely degrade the execution performance of the program.

8.5.2 Attaching a Shared Data Segment

The *sdget* function can also be used to attach an existing shared data segment to a process's data space. In this case, the function call has the form

```
sdget(path, flags)
```

where *path* is a character pointer to the pathname of a shared data segment created by some other process, and *flag* 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.

The function can be used to attach any shared data segment a process may wish to access. For example, in the following fragment, the program 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*.

```
#include <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 data segment without the necessary permissions. The segment permissions are defined when the segment is created.

8.5.3 Entering a Shared Data Segment

The *sdenter* signals a process's intention to access the contents of a shared data segment. A process cannot access the contents of the segment unless it enters the segment. The function call has the form:

sdenter (*addr*,*flag*)

where *addr* is a character pointer to the segment to be accessed, and *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 may also be combined by logically ORing them. 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 SD_UNLOCK flag and another process is currently accessing it.

Once a process has entered a segment it can examine and modify the contents of the segment. For example, in the following fragment, the program uses *sdenter* to enter the segment for reading and writing, then sets the first value in the segment to 0 if it is equal to 255.

```
#include <sd.h>

main ()
{
char*share;

share = sdget("/tmp/share", SD_WRITE);

sdenter(share, SD_WRITE);
        if (share[0]==255)
                share[0] = 0;
        .
        .
        .
}
```

In general, it is unwise to stay in a shared data 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 data 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 data segment that cannot be shared simultaneously, the program must not call the *fork* function when it is also accessing that segment.

8.5.4 Leaving a Shared Data Segment

The *sdleave* function signals a process's intention to leave a shared data segment after reading or modifying its contents. The function call has the form:

sdleave (*addr*)

where *addr* is a pointer with type **char** to the desired segment. The function returns −1 if it encounters an error, otherwise it returns 0. The return value is always an integer.

The function should be used after each access of the shared data to terminate the access. If the segment's lock flag is set, the function must be used after each access to allow other processes to access the segment. For example, in the following program fragment *sdleave* terminates each access to the segment given by "shared".

```
#include <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);
}

}
```

8.5.5 Getting the Current Version Number

The *sdgetv* function returns the current version number of the given data segment. The function call has the 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.

The function is typically used 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 <sd.h>

main()
{
char *shared;

if (sdgetv(shared) > 10)
        sdenter(shared);
```

In this example, the segment is entered if the current version number of the segment is greater than "10".

8.5.6 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 form:

sdwaitv (*addr*, *vnum*)

where *addr* is a character pointer to the desired segment, and *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.

The function is typically used to synchronize the actions of two separate processes. For example, in the following program fragment the program waits while the program corresponding to the version number "vnum" performs its operations in the segment.

```
#include <sd.h>

main ()
{
char *share;
intchange;

vnum = sdgetv( share );
i=0;
if ( sdwaitv( share, vnum ) == -1 )
        fprintf(stderr, "Cannotfind segment\n");
else
        sdenter( share );
```

If an error occurs while waiting, an error message is printed.

8.5.7 Freeing a Shared Data Segment

The *sdfree* function detaches the current process from the given shared data segment. The function call has the form:

```
sdfree(addr)
```

where *addr* is a character pointer to the segment to be set free. 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 SD_CREAT flag.

8.5.8 Program Example

This section shows how to use the shared data functions to share a single data segment between two processes. The following program attaches a data segment named */tmp/share* and then uses it to transfer information to between the child and parent processes.

```
#include <sd.h>

main()
{
 char *share, message[12];
 int i, num;

 share = sdget("/tmp/share",SD_CREAT|SD_WRITE, 12, 0777);

 if (fork()==0) {
      for (i=0; i<4; i++) {
        sdenter(share, SD_WRITE);
              strncpy(message, share, 12);
              strncpy(share,"Shared data", 12);
              vnum=sdget (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 data segment created by the parent process. Each process accesses the segment 5 times. During the access, a process copies the current contents of the segment to the variable *message* 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 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 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.

Replace this Page with Tab Marked:

m4

Appendix A
M4: A Macro Processor

A.1 Introduction

The **m4**(CP) macro processor defines and processes specially defined strings of characters called macros. By defining a set of macros to be processed by **m4**, a programming language can be enhanced to make it:

- More structured

- More readable

- More appropriate for a particular application

The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor–replacement of text by other text.

Besides the straightforward replacement of one string of text by another, **m4** provides:

- Macros witb arguments

- Conditional macro expansions

- Arithmetic expressions

- File manipulation facilities

- String processing functions

The basic operation of **m4** is copying its input to its output. As the input is read, each alphanumeric token (that is, string of letters and digits) is checked. If the token is the name of a macro, then the name of the macro is replaced by its defining text. The resulting string is reread by **m4**. Macros may also be called with arguments, in which case the arguments are collected and substituted in the right places in the defining text before **m4** rescans the text.

m4 provides a collection of about twenty built-in macros. In addition, the user can define new macros. Built-ins and user-defined macros work in exactly the same way, except that some of the built-in macros have side effects on the state of the process.

A.2 Invoking m4

The invocation syntax for **m4** is:

```
m4 [files]
```

Each file name argument is processed in order. If there are no arguments, or if an argument is a dash (-), then the standard input is read. The processed text is written to the standard output, and can be redirected as in the following example:

```
m4 file1 file2 - > outputfile
```

Note the use of the dash in the above example to indicate processing of the standard input, *after* the files and have been processed by **m4**.

A.3 Defining Macros

The primary built-in function of **m4** is **define**, which is used to define new macros. The input:

define(*name, stuff*)

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. *name* must be alphanumeric and must begin with a letter (the underscore (_) counts as a letter). *stuff* is any text, including text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example:

```
define(N, 100)
.
.
.
if (i > N)
```

defines "N" to be 100, and uses this symbolic constant in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by a left parenthesis, "(", it is assumed to have no arguments. This is the situation for "N" above; it is actually a macro with no arguments. Thus, when it is used, no parentheses are needed following its name.

You should also notice that a macro name is only recognized as such if it appears surrounded by nonalphanumerics. For example, in:

```
define(N, 100)
...
if (NNN > 100)
```

the variable "NNN" is absolutely unrelated to the defined macro "N", even though it contains three N's.

Things may be defined in terms of other things. For example:

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if "N" is redefined? Or, to say it another way, is: "M" defined as "N" or as 100? In m4, the latter is true, "M" is 100, so even if "N" subsequently changes, "M" does not.

This behavior arises because m4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string "N" is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it is just as if you had said:

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now "M" is defined to be the string "N", so when you ask for "M" later, you will always get the value of "N" at that time (because the "M" will be replaced by "N" which, in turn, will be replaced by 100).

A.4 Quoting

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by single quotation marks ` and ' is not expanded immediately, but has the quotation marks stripped off. If you say:

```
define(N, 100)
define(M, `N')
```

the quotation marks around the "N" are stripped off as the argument is being collected, but they have served their purpose, and "M" is defined as the string "N", not 100. The general rule is that **m4** always strips off one level of single quotation marks whenever it evaluates something. This is true even outside of macros. If you want the word "define" to appear in the output, you have to quote it in the input, as in:

```
'define'= 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining "N":

```
define(N, 100)
...
define(N, 200)
```

Perhaps regrettably, the "N" in the second definition is evaluated as soon as it is seen; that is, it is replaced by 100, so it's as if you had written:

```
define(100, 200)
```

This statement is ignored by **m4**, since you can only define things that look like names, but it obviously does not have the effect you wanted. To really redefine "N", you must delay the evaluation by quoting:

```
define(N, 100)
...
define('N', 200)
```

In **m4**, it is often wise to quote the first argument of a macro.

If the forward and backward quotation marks (` and ') are not convenient for some reason, the quotation marks can be changed with the built-in **changequote**. For example:

```
changequote([,])
```

makes the new quotation marks the left and right brackets. You can restore the original characters with just:

```
changequote
```

There are two additional built-ins related to **define**. The built-in **undefine** removes the definition of some macro or built-in:

```
undefine('N')
```

removes the definition of "N". Built-ins can be removed with **undefine**, as in:

```
undefine('define')
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. For instance, pretend that either the word "xenix" or "unix" is defined according to a particular implementation of a program. To perform operations according to which system you have you might say:

```
ifdef('xenix', 'define(system,1)')
ifdef('unix', 'define(system,2)')
```

Do not forget the quotation marks in the above example.

Ifdef actually permits three arguments: if the name is undefined, the value of **ifdef** is then the third argument, as in:

```
ifdef('xenix', on XENIX, not on XENIX)
```

A.5 Using Arguments

So far we have discussed the simplest form of macro processing– replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of *$n* will be replaced by the *n*th argument when the macro is actually used. Thus, the macro *bump*, defined as:

```
define(bump, $1=$1+1)
```

generates code to increment its argument by 1:

```
bump(x)
```

is:

```
x=x+1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through $1 to $9. (The macro name itself is $0.) Arguments that are not supplied are replaced by null strings, so we can define a macro *cat* which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus:

```
cat(x, y, z)
```

is equivalent to:

```
xyz
```

The arguments $4 through $9 are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus:

```
define(a, b  c)
```

defines "a" to be "b   c".

Arguments are separated by commas, but parentheses are counted properly, so a comma protected by parentheses does not terminate an argument. That is, in:

```
define(a, (b,c))
```

there are only two arguments; the second is literally "(b,c)". And of course a bare comma or parenthesis can be inserted by quoting it.

A.6 Using Arithmetic Built-ins

m4 provides two built-in functions for doing arithmetic on integers. The simplest is **incr**, which increments its numeric argument by 1. Thus, to handle the common programming situation where you want a variable to be defined as one more than N, write:

```
define(N, 100)
define(N1, `incr(N)')
```

Then "N1" is defined as one more than the current value of "N".

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the following operators (in decreasing order of precedence):

```
unary + and -
** or ^  (exponentiation)
* / %  (modulus)
+ -
== != < <= > >=
!  (not)
& or &&      (logical and)
| or ||  (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is implementation dependent.

As a simple example, suppose we want "M" to be "2**N+1". Then:

```
define(N, 3)
define(M, `eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

A.7 Manipulating Files

You can include a new file in the input at any time by the built-in function **include**:

include(*filename*)

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** (for "silent include") says nothing and continues if it cannot a cess the file.

It is also possible to divert the output of **m4** to temporary files during processing, and output the collected material upon command. **m4** maintains nine of these diversions, numbered 1 through 9.

If you say:

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as "n". Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is not the diverted stuff. Furthermore, the diverted material is not rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

A.8 Using System Commands

You can run any program in the local operating system with the **syscmd** built-in. For example,

```
syscmd(date)
```

runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique filenames, the built-in **maketemp** is provided, with specifications identical to the system function **mktemp**: a string of "XXXXX" in the argument is replaced by the process id of the current process.

A.9 Using Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

ifelse(*a*, *b*, *c*, *d*)

compares the two strings *a* and *b*. If these are identical, **ifelse** returns the string *c*; otherwise it returns *d*. Thus, we might define a macro called **compare** which compares two strings and returns "yes" or "no" if they are the same or different.

```
define(compare, `ifelse($1, $2, yes, no)')
```

Note the quotation marks, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input:

ifelse(*a*, *b*, *c*, *d*, *e*, *f*, *g*)

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise the result is *g*. If the final argument is omitted, the result is null, so:

ifelse(*a*, *b*, *c*)

is *c* if *a* matches *b*, and null otherwise.

A.10 Manipulating Strings

The built-in **len** returns the length of the string that makes up its argument. Thus:

```
len(abcdef)
```

is 6, and:

```
len((a,b))
```

is 5.

The built-in **substr** can be used to produce substrings of strings. For example:

substr(*s*,*i*,*n*)

returns the substring of *s* that starts at position *i* (origin zero), and is *n* characters long. If *n* is omitted, the rest of the string is returned, so:

```
        substr('now is the time', 1)
```

is:

```
        owisthetime
```

If *i* or *n* are out of range, various sensible things happen.

The command:

```
        index(s1,s2)
```

returns the index (position) in *s1* where the string *s2* occurs, or −1 if it does not occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

```
        translit(s,f,t)
```

modifies *s* by replacing any character found in *f* by the corresponding character of *t*. That is:

```
        translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that do not have an entry in *t* are deleted; as a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*. So:

```
        translit(s, aeiou)
```

deletes vowels from "s".

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline. It is useful mainly for throwing away empty lines that otherwise tend to clutter up **m4** output. For example, if you say:

```
        define(N, 100)
        define(M, 200)
        define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this is:

```
divert(-1)
        define(...)
        ...
divert
```

A.11 Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus, you can say:

```
errprint('fatal error')
```

Dumpdef is a debugging aid that dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Do not forget the quotation marks.

Replace this Page
with Tab Marked:

System Calls

Appendix B
XENIX System Calls

B.1 Introduction

This appendix lists some of the differences between XENIX 2.3, XENIX 3.0, UNIX V7, UNIX System 3.0 and XENIX System V. It is intended to aid users who wish to convert system calls in existing application programs for use on other systems.

B.2 Executable File Format

XENIX 3.0, UNIX System 3.0, and XENIX System V execute only those programs with the *x.out* executable file format. The format is similar to the old *a.out* format, but contains additional information about the executable file such as text and data relocation bases, target machine identification, word and byte ordering, symbol table, and relocation table format. The *x.out* file also contains the revision number of the kernel which is used during execution to control access to system functions. XENIX System V has a segmented *x.out* header which contains segmentation information, as well as relocation information. To execute existing programs in *a.out* format, you must first convert to the *x.out* format. The format is described in detail in **a.out**(F) in the XENIX *Reference*.

XENIX System V uses little-endian (low order word first in memory) word order for longs whereas some XENIX 3.0 systems use big-endian (high order word first in memory) word order. XENIX System V checks the *x.out* header for information about the word order. XENIX System V maintains full XENIX 3.0 binary compatibility. XENIX System V executes XENIX 3.0 word-swapped (big-endian) executable files as well as XENIX 3.0 and XENIX System V (little-endian) executables. Refer to the **machine**(HW) manual page in the XENIX *Reference* for a complete description of binary compatibility.

B.3 Revised System Calls

Some system calls in XENIX System V and UNIX System V have been revised and do not perform the same tasks as the corresponding calls in previous systems. To provide compatibility for old programs, XENIX System V and UNIX System V maintain both the new and the old system calls and automatically check the revision information in the *x.out* header to determine which version of a system call should be made. The following table lists the revised system calls and their previous versions.

System Call #	XENIX 2.3 function	System 3 function	System V function
35	ftime	unused	unused
38	unused	clocal	clocal
39	unused	setpgrp	setpgrp
40	unused	cxenix	cxenix
57	unused	utssys	utssys
62	clocal	fcntl	fcntl
63	cxenix	ulimit	ulimit

The *cxenix*() function provides access to system calls unique to XENIX 3.0 and/or XENIX System V. The *clocal* funtion provides access to all calls unique to an OEM.

The new XENIX System V system calls are accessed via *cxenix*() system calls with their numbers. Note that these numbers are not regular system call numbers, but *cxenix*() numbers. To use these calls, the *cxenix*() system call is made, with the high byte set to the appropriate number listed below (i.e., to call *locking*, take 40, add 256*1 to it, and pass the resulting value in **ax** when trapping into the kernel.) The XENIX 3.0 and System V system calls are listed at the below.

These calls are valid for XENIX 3.0 and XENIX System V:

cxenix Call #	Function	System Call
0	shutdown OS	shutdn
1	record locking	locking
2	create semaphore	creatsem
3	open semaphore	opensem
4	signal semaphore	sigsem
5	wait semaphore	waitsem
6	nonblocking waitsem	nbwaitsem
7	blocking read check	rdchk
8	set stack limit	stkgrow
9	extended ptrace	xptrace
10	change file size	chsize
11	XENIX 2.3 ftime call	ftime
12	sleep for short interval	nap
13	attach to shared data	sdget
14	release shared data	sdfree
15	enter critical region	sdenter
16	leave critical region	sdleave
17	get shared data version #	sdgetv
18	wait for new shared data version	sdwaitv

The following calls are found in XENIX System V only:

cxenix Call#	Function	System Call
19	change segment size	brkctl
22	message control	msgctl
23	get message queue	msgget
24	send message	msgsnd
25	receive message	msgrcv
26	semaphore control	semctl
27	get semaphore set	semget
28	semaphore ops	semop
29	sysV shared memory control	shmctl
30	sysV create shared memory	shmget
31	sysV attach shared memory	shmat

B.4 Version 7 Additions

XENIX System V maintains a number of XENIX 3.0 and UNIX V7 features that were dropped from UNIX System 3.0. In particular, XENIX System V continues to support the **dup2**(S) and **ftime**(S) functions. The **ftime** function, used with the **ctime**(S) function, provides the default value for the time zone when the TZ environment variable has not been set. This means a binary configuration program can be used to change the default time zone. No source license is required.

B.5 Changes to the ioctl Function

XENIX 3.0 and UNIX System 3.0 have a full set of XENIX 2.3-compatible **ioctl** calls. Furthermore, XENIX 3.0 and XENIX System V have resolved problems that previously hindered UNIX System 3.0 compatibility. For convenience, XENIX 2.3-compatible **ioctl** calls can be executed by a UNIX System 3.0 executable. The available XENIX 2.3 **ioctl** calls are: TIOCSETP, TIOCSETN, TIOCGETP, TIOCSETC, TIOCGETC, TIOCEXCL, TIOCNXCL, TIOCHPCL, TIOCFLUSH, TIOCGETD, and TIOCSETD.

B.6 Pathname Resolution

If a null pathname is given, XENIX 2.3 interprets the name to be the current directory, but UNIX System 3.0 considers the name to be an error. XENIX 3.0 and XENIX System V use the version number in the *x.out* header to determine what action to take. A XENIX 2.3 header causes null pathnames to be the current directory. Any other version is interpreted as an error.

If the symbol ".." is given as a pathname when in a root directory that has been defined using the **chroot**(S) function, XENIX 2.3 moves to the next higher directory. XENIX 3.0 also allows the ".." symbol to **chroot**, but restricts its use to the super-user. XENIX System V does not allow the ".." symbol to **chroot**.

B.7 Using the mount () and chown () Functions

XENIX 3.0, and UNIX System 3.0 restrict the use of the **mount**(S) system call to the super-user. XENIX System V does not restrict the use of the **mount** system call, usually however, the **mount**(C) program is only executable by the super-user. Also, XENIX System V, 3.0 and UNIX System 3.0 allow the owner of a file to use **chown**(S) function to change the file ownership.

B.8 Super-Block Format

XENIX System V, UNIX System 3.0 and UNIX System 5.0 have new super-block formats. XENIX System V and XENIX 3.0 use the System 5.0 format, but use a different magic number for each revision. The XENIX System V and XENIX 3.0 super-blocks have an additional field at the end which can be used to distinguish between XENIX 2.3, 3.0 and System V super-blocks. XENIX System V and XENIX 3.0 check this magic number at boot time and during a mount. If a XENIX 2.3 super-block is read, XENIX 3.0 converts it to the new format internally. Similarly, if a XENIX 2.3 super-block is written, XENIX 3.0 converts it back to the old format. This permits XENIX 2.3 kernels to be run on file systems also usable by UNIX System 3.0.

However, XENIX System V is word-swapped relative to XENIX-86 3.0. Even though the super-block formats are the same, the order of bytes in long words is different. XENIX System V can not **mount**(C) or **fsck**(C) XENIX 3.0 filesystems.

B.9 Separate Version Libraries

XENIX System V supports the construction of XENIX 3.0 executable files. This systems maintains both the new and old versions of system calls in separate libraries.

Replace this Page
with Tab Marked:

Index

Index

Replace this Page
with Tab Marked:

C LANGUAGE REFERENCE

XENIX® System V

Development System

C Language Reference

This document was typeset with an IMAGEN® 8/300 Laser Printer.

SCO Document Number: XG-6-21-87-4.0

Contents

Chapter 1

Introduction

1.1 Overview

The C language is a general-purpose programming language well known for its efficiency, economy, and portability. While these advantages make it a good choice for almost any kind of programming, C has proven to be especially useful in systems programming because it allows programmers to write fast and compact programs and to transport those programs to other systems. In many cases, well-written C programs are comparable in speed to assembly language programs and offer the advantages of easier maintenance and greaterreadability.

In spite of C's efficiency and power, it is 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. Instead, C programmers rely on run-time libraries to perform such tasks.

This design contributes to C's adaptability and compactness. Because the language is relatively confined, it does not assume or impose a particular programming model. Run-time routines provide support as needed, allowing the programmer to minimize their use, if desired, or to tailor run-time routines for special purposes.

The design also helps to isolate language features from processor-specific features in a particular C implementation, thus aiding programmers who want to write portable code. The strict definition of the language makes it independent of any particular operating system or machine; at the same time, programmers can easily add system-specific routines to take advantage of a particular machine's efficiencies.

Some of the significant features of the C language are as follows:

- *C provides a full set of loop, conditional,* to control program flow logically and efficiently and to encourage structured programming.
- *C offers an unusually large set of operators.* Many of C's operators correspond to common machine instructions, allowing a direct translation into machine code. The variety of operators lets the programmer specify different kinds of operations clearly and with a minimum of code.
- *C's data types include several sizes of integers,* single- and double-precision floating-point types. The programmer can design more complex data types, such as arrays and data structures, to suit specific program needs.

- *C programmers can declare "pointers" to variables and functions.* A pointer to an item corresponds to the machine address of that item. Using pointers wisely can increase program efficiency considerably, since pointers let the programmer refer to items in the same way the machine does. C also supports pointer arithmetic, allowing the programmer both to access and manipulate memory addresses directly.

- *The C preprocessor, a text processor, acts on the text of files before compilation.* Among its most useful applications for C programs are the definition of program constants, the substitution of function calls with faster macro look-alikes, and conditional compilation. The preprocessor is not limited to processing C files; it can be used on any text file.

- *C is a flexible language*, leaving much of the decision-making up to the programmer. In keeping with this attitude, C imposes few restrictions in matters such as type conversion. While this is often an asset, it is important for C programmers to be thoroughly familiar with the definition of the language in order to understand how their programs will behave.

1.2 About This Guide

The XENIX *C Language Reference* defines the C language as implemented by Microsoft. It is intended as a reference for programmers who have experience in C or in another programming language. Knowledge of programming fundamentals is assumed.

The run-time library functions available for use in Microsoft C programs are discussed in the XENIX *C Library Guide.*

Consult the XENIX *C Library Guide.* for an explanation of how to compile and link C programs on your system. The XENIX *C Library Guide.* also contains information specific to the implementation of C on your system.

This guide is organized as follows:

Chapter 1, "Introduction" summarizes the organization of the guide and the conventions used.

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 allows the programmer to declare aggregate types and pointers.

Chapter 5, "Expressions and Assignments", describes the operands and operators that make up C expressions and assignments. The type conversions and side effects that may accompany the evaluation of expressions are also discussed in this chapter.

Chapter 6, "Statements", describes C statements. Statements control the flow of program execution.

Chapter 7, "Functions", discusses features of C functions. In particular, it explains how to define, declare, and call a function and describes function parameters and return values.

Chapter 8, "Preprocessor Directives", describes the instructions recognized by the C preprocessor. The C preprocessor is a text processor automatically invoked before compilation.

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, published in 1978 by Prentice-Hall, Inc.

Appendix B, "Syntax Summary", summarizes the syntax of the Microsoft C language implementation.

The remainder of this chapter describes the notational conventions used throughout the guide.

1.3 Notational Conventions

This guide uses a number of notational conventions to describe the syntax of XENIX commands:

boldface	Boldface indicates a command, option, flag, or program name to be entered as shown. Boldface indicates the name of a library routine, global variable, standard type, constant, keyword, or identifier used by the C library. (To find more information on a given library routine consult the "Alphabetized List" in your XENIX *Reference* for the page that describes it.)

italics	Italics indicate a filename. This pertains to library include filenames (i.e. *stdio.h*), as well as, otherfilenames (i.e. */etc/ttys*). Italics indicate a placeholder for a command argument. When entering a command , a place-holder must be replaced with an appropriate filename, number, or option. Italics indicate a specific identifier, supplied for variables and functions, when mentioned in text. Italics indicate user named routines. (User named routines are followed by open and close parentheses, ().) Italics indicate emphasized words or phrases in text.
CAPITALS	Capitals indicate names of environment variables (i.e. TZ and PATH).
SMALL CAPITALS	Small capitals indicate keys and key sequences (i.e. RETURN).
[ ]	Brackets indicate that the enclosed item is optional. If you do not use the optional item, the program selects a default action to carry out.
. . .	Ellipses indicate that you can repeat the preceding item any number of times. Vertical ellipses indicate that a portion of a program example is omitted.
" "	Quotation marks indicate the first use of a technical term. Quotation marks indicate a reference to a word rather than a command.

Chapter 2

Elements of C

2.1 Introduction

This chapter describes the elements of the C programming language. The elements of the language are the names, numbers, and characters used to construct a C program. In particular, this chapter describes:

- 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 that have a specific meaning to the C compiler. C programs are constructed 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 consists of all letters, digits, and symbols that a user can represent graphically with a single character. The extent of the representable character set depends on the type of terminal, console, or character device being used.

A C program can contain only characters from the C character set, except that string literals, character constants, and comments can use any representable character. Each character in the C character set has an explicit meaning to the C compiler. The compiler generates error messages when it encounters misused characters or characters not belonging to the C character set.

The following sections describe the characters and symbols of the C character set and explain how and when to use them.

2.2.1 Letters and Digits

The C character set includes the uppercase and lowercase letters of the English alphabet and the ten decimal digits of the Arabic number system:

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

These letters and digits can be used to form the constants, identifiers, and keywords described later in this chapter.

The C compiler treats uppercase and lowercase letters as distinct characters. If a lowercase "a" is specified in a given item, you cannot substitute an uppercase "A" in its place; you must use the lowercase letter.

2.2.2 Whitespace Characters

Space, tab, linefeed, carriage return, form feed, vertical tab, and newline characters are called whitespace characters because they serve the same purpose as the spaces between words and lines on a printed page. These characters separate user-defined items, such as constants and identifiers, from other items within a program.

The C compiler ignores whitespace characters unless they are used as separators or as components of character constants or string literals. This means you can use extra whitespace characters to make a program more readable. Comments (see Section 2.6) are also treated as whitespace.

2.2.3 Punctuation and Special Characters

The punctuation and special characters in the C character set are used for a variety of purposes, from organizing the text of a program to defining the tasks to be carried out by the compiler or by the compiled program. Table 2.1 lists these characters:

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	_	Underscore
"	Double quotation	#	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	+	Plus sign
>	Right angle bracket		

These characters have special meaning to the C compiler. Their use in the C language is described throughout this guide. Punctuation characters in the representable character set that do not appear in this list can be used only in string literals, character constants, and comments.

2.2.4 Escape Sequences

Escape sequences are special character combinations that represent whitespace and nongraphic characters in strings and character constants. They are typically used to specify actions such as carriage returns and tab movements on terminals and printers and to provide literal representations of characters that normally have special meanings, such as the double quote (") character. An escape sequence consists of a backslash followed by a letter or combination of digits. Table 2.2 lists the C language escape sequences:

Table2.2

Escape Sequences

Escape Sequence	Name
\n	Newline
\t	Horizontal tab
\v	Vertical tab
\b	Backspace
\r	Carriage return
\f	Form feed
\'	Single quote
\"	Double quote
\\	Backslash
\ddd	ASCII character in octal notation
\x*dd*	ASCII character in hexadecimal notation

If the backslash precedes a character not included in the list above, the backslash is ignored and that character is represented literally. For example, the pattern "\c" represents the character "c" in a string literal or character constant.

The sequences "\ddd" and "\x*dd*" allow any character in the ASCII character set to be given as a three-digit octal or a two-digit hexadecimal character code. For example, the backspace character can be given as "\010" or "\x08". The ASCII null character can be given as "\0" or "\x0".

Only octal digits can appear in an octal escape sequence, and at least one digit must appear. However, fewer than three digits can be specified. For example, the backspace character can also be given as "\10". Similarly, a hexadecimal escape sequence must contain at least one digit, but the second digit can be omitted. The hexadecimal escape sequence for the backspace character can be given as "\x8". However, when using octal and hexadecimal escape sequences in strings, it is safer to give all three digits of the octal or hexadecimal escape sequence. Otherwise, the character following the escape sequence may be interpreted as part of the sequence, if it happens to be an octal or hexadecimal digit.

Escape sequences allow nongraphic control characters to be sent 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.

Nongraphic characters should always be represented by escape sequences. Placing a nongraphic character in a C program has unpredictable results.

The backslash character (\) used to introduce escape sequences also functions as a continuation character in strings and in preprocessor definitions. When a newline character follows the backslash, the newline is disregarded, and the next line is treated as part of the previous line.

2.2.5 Operators

Operators are special character combinations that specify how values are to be transformed and assigned. The compiler interprets each of these character combinations as a single unit, called a "token" (see Section 2.7).

Table 2.3 lists the characters that form C operators and gives the name of each operator. Operators must be specified exactly as they appear in the tables, with no whitespace between the characters of multicharacter operators. The **sizeof** operator is not included in this table; it consists of a keyword (**sizeof**) rather than a symbol.

Table 2.3

Operators

Operator	Name
!	Logical NOT
~	Bitwise complement
+	Addition
-	Subtraction, arithmetic negation
*	Multiplication, indirection
/	Division
%	Remainder
<<	Shift left
>>	Shift right
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equality
!=	Inequality
&	Bitwise AND, address-of
|	Bitwise inclusive OR
^	Bitwise exclusive OR
&&	Logical AND
||	Logical OR
,	Sequential evaluation
?:	Conditional[a]
++	Increment
--	Decrement
=	Simple assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Remainder assignment
>>=	Right shift assignment
<<=	Left shift assignment
&=	Bitwise AND assignment
|=	Bitwise inclusive OR assignment
^=	Bitwise exclusive OR assignment

[a]The conditional operator is a ternary operator, not a multicharacter operator. The form of a conditional expression is: *expression ? expression : expression*

See Chapter 5, "Expressions and Assignments," for a complete description of each operator.

2.3 Constants

A constant is a number, a character, or a string of characters that can be used as a value in a program. The value of a constant does not change from execution to execution.

The C language has four kinds of constants: integer constants, floating-point constants, character constants, and string literals. The following sections define the format and use of each.

2.3.1 Integer Constants

An integer constant is a decimal, octal, or hexadecimal number that represents an integer value. A decimal constant has the form:

digits

where *digits* are one or more decimal digits (0 through 9).

An octal constant has the form:

0*odigits*

where *odigits* are one or more octal digits (0 through 7). The leading zero is required.

A hexadecimal constant has the form:

0x*hdigits*

where *hdigits* is one or more hexadecimal digits (0 through 9 and either uppercase or lowercase "a" through "f"). The leading zero is required and must be followed by "x".

No whitespace characters can appear between the digits of an integer constant. Table 2.4 illustrates the form of integer constants:

Table 2.4

Integer Constants

Decimal Constants	Octal Constants	Hexadecimal Constants
10	012	0xa or 0xA
132	0204	0x84
32179	076663	0x7db3 or 0x7DB3

Integer constants always specify positive values. If negative values are required, the minus sign (–) can be placed in front of the constant to form a constant expression with a negative value. The minus sign is treated as an arithmetic operator.

Every integer constant is given a type based on its value. A constant's type determines what conversions must be performed when the constant is used in an expression or when the minus sign (-) is applied. 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 also given **int** type, or **long** type if the size of the value requires it. However, unlike other signed numbers, octal and hexadecimal constants are not sign-extended in type conversions.

The programmer can direct the C compiler to force any integer constant to have **long** type by appending the letter "l" or "L" to the end of the constant. Table 2.5 illustrates **long** integer constants:

Table 2.5

Long Integer Constants

Decimal Constants	**Octal Constants**	**Hexadecimal Constants**
10L	012L	0xaL or 0xAL
79l	0115l	0x4fl or 0x4Fl

Types are described in Chapter 4, "Declarations"," and conversions are described in Chapter 5, "Expressions and Assignments"."

2.3.2 Floating-Point Constants

A floating-point constant is a decimal number representing a signed real number. The value of a signed real number includes an integer portion, a fractional portion, and an exponent. Floating-point constants have the form:

[*digits*][.*digits*][E[-]*digits*]

where *digits* are one or more decimal digits (0 through 9), and E (or e) is the exponent symbol. Either the digits before the decimal point (the integer portion of the value) or the digits after the decimal point (the fractional portion) can be omitted, but not both. The exponent consists of the exponent symbol followed by a possibly negative constant integer value. The decimal point can be omitted only when an exponent is given. No whitespace characters can separate the digits or characters of the constant.

Floating-point constants always specify positive values. If negative values are required, the minus sign (-) can be placed in front of the constant to form a constant floating-point expression with a negative value. The minus sign is treated as an arithmetic operator.

The following examples illustrate some of the forms of floating-point constants and expressions:

```
15.75
1.575E1
1575e-2
-0.0025
-2.5e-3
25E-4
```

The integer portion of the floating-point constant can be omitted, as shown in the following examples:

```
.75
.0075e2
-.125
-.175E-2
```

All floating-point constants have type **double**.

2.3.3 Character Constants

A character constant is a letter, digit, punctuation character, or escape sequence enclosed in single quotation marks. The value of a character constant is the character itself. Character constants consisting of more than one character or escape sequence are not allowed.

A character constant has the form:

> '*char*'

where *char* can be any character from the representable character set (including any escape sequence) except a single quotation mark ('), a backslash (\), or a newline character. To use a single quotation mark or backslash character as a character constant, precede it with a backslash as shown in Table 2.6. To represent a newline character, use the escape sequence '\n'.

Table 2.6

Examples of Character Constants

Constant	Value
'a'	Lowercase a
'?'	Question mark
'\b'	Backspace
'\x1B'	ASCII escape character
'\''	Single quotation mark
'\\'	Backslash

Character constants have type **char** and consequently are sign-extended in type conversions (see Section 5.7 of Chapter 5, "Expressions and Assignments").

2.3.4 String Literals

A string literal is a sequence of letters, digits, and symbols enclosed in double quotation marks. A string literal is treated as an array of characters; each element of the array is a single character value.

The form of a string literal is:

"*characters*"

where *characters* are one or more characters from the representable character set, excluding the double quotation mark ("), the backslash (\), and the newline character. To use the newline character in a string, type a backslash immediately followed by a newline character. The backslash causes the newline character to be ignored. This allows the programmer to form string literals that occupy more than one line. For example, the string literal:

```
"Long strings can be bro\
ken into two pieces."
```

is identical to the string:

```
"Long strings can be broken into two pieces."
```

To use the double quotation mark or backslash character within a string literal, precede it with a backslash, as shown in the following examples:

```
"This is a string literal."
"Enter a number between 1 and 100\n Or press Return"
"First\\Second"
"\"Yes, I do,\"she said."
```

Notice that escape sequences (such as \n and \") can appear in string literals.

The characters of a string are stored in order at contiguous memory locations. A null character (\) is automatically appended to mark the end of the string. Each string in a program is considered to be a distinct item. If two identical strings appear in a program, they each receive distinct storage space.

String literals have the type **char** [ ]. This means a string is an array whose elements have type **char**. The number of elements in the array is the number of characters in the string literal plus one, since the null character stored after the last character counts as an array element.

2.4 Identifiers

Identifiers are the names you supply for the variables, functions, and labels used in a given program. You can create an identifier by declaring it with the associated variable or function. You can use the identifier in later statements within the program to refer to the given item. (Declarations are described in Chapter 4, "Declarations".")

An identifier is a sequence of one or more letters, digits, or underscores (_) that begins with a letter or underscore. Any number of characters are allowed in a given identifier, but only the first 31 characters are significant to the compiler. (Other programs that read the compiler output, such as the linker, may use fewer characters.) Use leading underscores with care. Identifiers beginning with an underscore can conflict with the names of hidden system routines and produce errors.

The following are examples of identifiers:

```
j
cnt
templ
top_of_page
skip12
```

The C compiler considers uppercase and lowercase letters to be separate and distinct characters. Thus, you can create distinct identifiers that have the same spelling but different cases for one or more of the letters. For example, each of the following identifiers is unique:

```
add
ADD
Add
aDD
```

The C compiler does not allow identifiers that have the same spelling and case as a C language keyword. Keywords are described in Section 2.5.

The linker may further restrict the number and type of characters for globally visible symbols. Furthermore, unlike the compiler, the linker may not distinguish between uppercase and lowercase letters. Consult your linker documentation for information on naming restrictions imposed by the linker.

2.5 Keywords

Keywords are predefined identifiers that have special meaning to the C compiler. They can be used only as defined. The names of program items may not conflict with the keywords listed below:

auto	**default**	**float**	**register**	**switch**
break	**do**	**for**	**return**	**typedef**
case	**double**	**goto**	**short**	**union**
char	**else**	**if**	**sizeof**	**unsigned**
const	**enum**	**int**	**static**	**void**
continue	**extern**	**long**	**struct**	**while**

Keywords cannot be redefined. However, you can specify text to be substituted for keywords before compilation by using C preprocessor directives (see Chapter 8, "Preprocessor Directives").

The **const** keyword is reserved for future use but is not yet implemented in the language.

The following identifiers may be keywords in some implementations; (see your system documentation for details):

far
fortran
huge
near
pascal

2.6 Comments

A comment is a sequence of characters that is treated as a single white-space character by the compiler but is otherwise ignored. A comment has the following form:

/ characters */*

Here *characters* can be any combination of characters from the representable character set, including newline characters but excluding the combination "*/". This means that comments can occupy more than one line, but they cannot be nested.

Comments are typically used to document the statements and actions of a C language source program. They can appear anywhere a whitespace character is allowed. Since the compiler ignores the characters of the comment, keywords can appear in comments without producing errors.

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 compiler recognizes the first "*/", after the word "nest", as the end of the comment. The compiler attempts to process the remaining text and produces an error when it cannot do so.

To suppress compilation of a large portion of a program or a program segment that contains comments, use the **#if** preprocessor directive instead of comments (see Section 8.4 of Chapter 8, "Preprocessor Directives").

2.7 Tokens

When the compiler processes a program, it breaks the program down into groups of characters known as "tokens." A token is a unit of program text that has meaning to the compiler and that cannot be broken down further. 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 whitespace characters and by other tokens, such as operators and punctuation symbols. To prevent the compiler from breaking an item down into two or more tokens, whitespace characters are prohibited between the characters of identifiers, multicharacter operators, and keywords.

When the compiler interprets tokens, it incorporates as many characters as possible into a single token before moving on to the next token. Because of this behavior, tokens not separated by whitespace may not be interpreted in the way expected.

For example, in the following expression, the compiler first makes the longest possible operator (++) from the three plus signs, and then processes the remaining plus sign as an addition operator (+):

```
i+++j
```

This expression is interpreted as "(i++) + (j)", not "(i) + (++j)". Use whitespace and parentheses to clarify your intent in such cases.

Chapter 3
Program Structure

3.1 Introduction

This chapter describes the structure of C language source programs and defines terms used later in this guide to describe the C language. It provides an overview of C language features that are described in detail in other chapters. In particular, the syntax and meaning of declarations and definitions are discussed in Chapter 4, "Declarations"," and Chapter 7, "Functions." The C preprocessor is described in Chapter 8, "Preprocessor Directives."

3.2 Source Program

A C source program is a collection of one or more directives, declarations, and/or definitions. "Directives" instruct the C preprocessor to perform specific actions on the text of the program prior to compilation. "Declarations" establish the names and attributes of variables, functions, and types used in the program.

"Definitions" are declarations that also define variables and functions. A variable definition gives the initial value of the declared variable, in addition to its name and type. The definition causes storage to be allocated for the variable. A function definition specifies the function body, a compound statement containing the declarations and statements that constitute the function. The function definition also gives the function name, formal parameters, and return type.

A source program can have any number of directives, declarations, and definitions. Each must have the appropriate syntax as described in this guide. They can appear in any order in the program, although the order affects how variables and functions can be used in the program (see Section 3.5).

A nontrivial program always contains at least one definition, a function definition. The function defines the action to be taken by the program. The following example illustrates a simple C source program.

Example

```
int x = 1;
    /* Variable definitions */
int y = 2;

extern int printf(char *,);
    /* Function declaration */

main()
    /* Function definition for main function */
{
    int z;
        /* Variable declarations */
    int w;

    z = y + x;
        /* Executable statements */
    w = y - x;
    printf("z= %d \nw= %d \n", z, w);
}
```

This source program defines the function named *main* and declares the function **printf**. The variables *x* and *y* are defined with variable definitions; the variables *z* and *w* are just declared.

3.3 Source Files

Source programs can be divided into one or more separate source files. A C source file is a text file that contains all or part of a C source program. It may, for example, contain just a few of the functions needed by the program. When the source program is compiled, the individual source files that make up the program must be compiled individually and then linked. Separate source files can also be combined to form larger source files before compilation by using the **#include** directive, discussed in Chapter 8, "Preprocessor Directives."

A source file can contain any combination of complete directives, declarations, and definitions. Items such as function definitions or large data structures cannot be split between source files.

A source file need not contain any executable statements. It is sometimes useful to place variable definitions in one source file and then declare references to these variables in other source files that use them. This makes the definitions easy to find and modify if necessary. For the same reason, manifest constants and macros (discussed in chapter 8, "Preprocessor Directives") are often organized into separate **include** files and inserted into source files where required.

Directives in a source file apply to that source file and its included files only. Moreover, each directive applies only to the portion of the file following the directive. If a common set of directives is to be applied to a source program, then all source files in the program must contain these directives.

The following is an example of a C source program contained in two source files. 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.

Example

```
/*****************************************************
    Source file 1 - main function
*****************************************************/

#define ONE   1
#define TWO   2
#define THREE 3

extern int max(int, int);
    /* Function declaration */

main ()
    /* Function definition */
{
    int w = ONE, x = TWO, y = THREE;
    int z = 0;
    z = max(x,y);
    w = max(z,w);
}

/*****************************************************
    Source file 2 - max function
*****************************************************/

int max(a, b)
    /* Function definition */
int a, b;
{
    if ( a > b )
        return (a);
    else
        return (b);
}
```

In the first source file, the function *max*() is declared without being defined. This is known as a "forward declaration". The function definition for *main*() includes function calls to *max*().

The lines beginning with a number sign (#) are preprocessor directives. These directives instruct the preprocessor to replace the identifiers *ONE*, *TWO*, and *THREE* with the specified number in the first source file. The directives do not apply to the second source file.

The second source file contains the function definition for *max*(). This definition satisfies the calls to *max*() in the first source file. Once the source files are compiled, they can be linked and executed as a single program.

3.4 Program Execution

Every program must have a primary (main) program function. This function serves as the startingpointfor program execution and usually controls execution of the program by directing the calls to other functions in the program. A program usually stops executing at the end of the main function, although it can stop at other points in the program, depending on the execution environment.

The source program usually has more than one function, each designed to perform one or more specific tasks. The *main*() function can call these functions to perform the tasks. When a function is called, execution begins at the first statement in the called function. The function returns control when a **return** statement is executed or the end of the function is encountered.

All functions, including the *main*() function, can be declared to have parameters. Functions called by other functions receive values for the parameters from the callingfunctions. Parameters of the *main*() function can be declared to receive values passed to the main function from outside the program (for example, 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 the main function. The argv parameter is declared as an array of pointers, each element of which points to a string representation of an argument passed to the *main*() function. 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 the main function. The argument-passing convention in use on a particular system is determined by the operating system rather than by the C language. See your system documentation for details.

Formal parameters to functions must be declared when the function is defined. Function definitions are described in more detail in Section 7.2 of Chapter 7, "Functions". Function declarations are discussed in Section 4.5 of Chapter 4, "Declarations".

3.5 Lifetime and Visibility

Two concepts, "lifetime" and "visibility," are important in understanding the structure of a C program. The lifetime of a variable or function can be either "global" or "local." An item with a global lifetime has storage and a defined value throughout the duration of the program. An item with a local lifetime is allocated new storage each time the "block" in which it is defined or declared is entered. When the block is exited, the local item loses its storage, and hence its value. Blocks are defined and discussed below.

An item is said to be "visible" in a block or source file if the type and name of the item are known in the block or source file. An item can also be "globally visible," which means that it is visible, or can be made visible through appropriate declarations, throughout all the source files that constitute the program. Visibility between source files (also known as "linkage") is discussed in greater detail in Section 4.6 of Chapter 4, "Declarations."

A block is a compound statement. Compound statements consist of declarations and statements, as described in Section 6.3 of Chapter 6, "Statements"." The bodies of C functions are compound statements. Blocks can be nested; function bodies frequently contain blocks, which in turn can contain blocks.

Declarations and definitions within blocks are said to occur at the "internal level." Declarations and definitions outside of all blocks occur at the "external level."

Both variables and functions can be *declared* at the external level or at the internal level. Variables can also be *defined* at the internal level, but functions can only be defined at the external level.

All functions have global lifetimes, regardless of where they are declared. Variables declared at the external level always have global lifetimes. Variables declared at the internal level usually have local lifetimes; however, the storage class specifiers **static** and **extern** can be applied to declare global variables or references to global variables within a block. See Section 4.6 of Chapter 4 "Declarations," for a discussion of these options.

Variables declared or defined at the external level are visible from the point at which they are declared or defined to the end of the source file. These variables can be made visible in other source files with appropriate declarations, as described in Section 4.6 of Chapter 4, "Declarations"." However, variables that are given **static** storage class at the external level are visible only within the source file in which they are defined.

In general, variables declared or defined at the internal level are visible from the point at which they are first declared or defined to the end of the block in which the definition or declaration appears. These variables are called local variables. If a variable declared inside a block has the same name as a variable declared at the external level, the block definition supersedes the external level definition of the variable for the duration of the block. The visibility of the external level variable is restored when the block is exited.

Block visibility can nest. This means that a block nested inside another block can contain declarations that redefine variables declared in the outer block. The redefinition of the variable holds in the inner block, but the original definition is restored when control returns to the outer block. Variables from outer blocks are visible inside all inner blocks, as long as they are not redefined in the inner blocks.

Functions with **static** storage class are visible only in the source file in which they are defined. All other functions are globally visible. See Section 4.5 of Chapter 4, "Declarations"," for more information on function declarations.

Table 3.1 summarizes the main factors that determine the lifetime and visibility of functions and variables. The table is not, however, intended to cover all cases. Refer to the above discussion and to Section 4.6 of Chapter 4, "Declarations," for more detailed information.

Table 3.1

Summary of Lifetime and Visibility

Level	Item	Storage Class Specifier	Lifetime	Visibility
External	Variable declaration	**static**	Global	Restricted to single source file
	Variable declaration	**extern**	Global	Remainder of source file
	Function declaration or definition	**static**	Global	Restricted to single source file
	Function declaration or definition	**extern**	Global	Remainder of source file
Internal	Variable definition or declaration	**extern** or **static**	Global	Block
	Variable definition or declaration	**auto** or **register**	Local	Block

The following program example illustrates blocks, nesting, and visibility of variables.

Example

```
/* 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);

  /* 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);

    /* 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);
}
```

In this 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 *main*() function prints out the values 1, 2, 3, 0, 3, 2, 1.

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 allows you to use the same identifier for more than one program item, as long as you follow the rules outlined in this section.

The compiler sets up "naming classes" to distinguish between the identifiers for different kinds of items. The names within each class must be unique to avoid conflict, but an identical name can appear in one or more naming classes. This means that you can use the same identifier for two or more different items if the items are in different naming classes. The context of a given identifier in the program allows the compiler to resolve the reference without ambiguity.

The kinds of items you can name in C programs, and the rules for naming them, are described below:

Variables and Functions	The names of variables and functions are in a naming class with formal parameters and enumeration constants. Variable and function names must, therefore, be distinct from other names in this class with the same visibility.
	However, variable names can be redefined within program blocks, as described in Section 3.5. Function names can also be redefined in this manner.
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. Redeclaring formal parameters within the function causes an error.
Enumeration Constants	Enumeration constants are in the same naming class as variable and function names. This means that 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, meaning that they can be redefined within blocks. See Section 3.5.

Tags	Enumeration, structure, and union tags are grouped together in a single naming class. Each enumeration, structure, or union tag 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 any other name in the program.
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 any other names or from label names in other functions.
Typedef Names	The names of types defined with **typedef** are treated as keywords. No other names with the same visibility are allowed to have the same spelling and case as a **typedef** keyword.

Example

```
struct student {
    char student[20];
    int class;
    int id;
    } student;
```

Structure tags, structure members, and variable names are in three different naming classes, so no conflict occurs between the three items named *student* in the above example. The compiler determines how to interpret each occurrence of *student* by its context in the program. For example, when *student* appears after the **struct** keyword, it is known to be a structure tag. When *student* appears after either of the member selection operators "." or "−>", the name refers to the structure member. In other contexts, the identifier *student* refers to the structure variable.

Chapter 4

Declarations

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*...]

where *sc-specifier* is a storage class specifier, *type-specifier* is the name of a defined type, *declarator* is an identifier that can be modified to declare a pointer, array or function, and *initializer* gives a value or sequence of values to be assigned to the variable being declared.

All C variables must be explicitly declared before they are used. C functions can be declared explicitly in a function declaration or implicitly by calling the function before it is declared or defined.

The C language defines a standard set of data types. You can add to that set by declaring new data types 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. When you declare simple variables (such as character, integer, and floating-point values), 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 portions of a program can reference it. The 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.

Although function declarations are presented in Section 4.5 of this chapter, function definitions are described in Section 7.2 of Chapter 7, "Functions"."

4.2 Type Specifiers

The C language provides definitions for a set of basic data types, called the "fundamental" types. Their names are listed in Table 4.1.

Table 4.1

Fundamental Types

Integral Types[a]	Floating-Point Types[a]	Other
char	**float**	**void**[b]
int	**double** (also called **longfloat**)	
shortint		
longint		
unsigned char		
unsigned int		
unsignedshortint		
unsigned long int		

[a]Used to declare variables and function return types.

[b]Used only to declare function return types.

Enumeration types are also considered fundamental types; type specifiers for enumeration types are discussed in Section 4.8.1. The **char**, **int**, **short int**, and **long int** types, together with their **unsigned** counterparts, are called "integral" types. The **float** and **double** type specifiers refer to "floating-point" types.

The **void** type can only be used to declare functions that return no value. Function types are discussed in Section 4.5.

You can create additional type specifiers with **typedef** declarations, discussed in Section 4.8.2.

Variable and function declarations can use any of the integral or floating-point type specifiers listed above. You can abbreviate some type specifiers, as shown in Table 4.2.

Table 4.2

Type Specifiers and Abbreviations

Type Specifier	Abbreviation
char	--
int	--
short int	short
long int	long
unsigned char	--
unsigned int	unsigned
unsigned short int	unsigned short
unsigned long int	unsigned long
float	--
long float	double

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 does not apply to variables, it is not included in the table.

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-dependent	
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-dependent	
unsigned short	2 bytes	0 to 65,535
unsigned long	4 bytes	0 to 4,294,967,295
float	4 bytes	IEEE standard notation; see discussion below.
double	8 bytes	IEEE standard notation; see discussion below.

The **char** type is used to store a letter, digit, or symbol from the representable character set. The integer value of a character is the ASCII code corresponding to that character. Since the **char** type is interpreted as a signed 1-byte integer, values in the range −128 to 127 are permitted for **char** variables, although only the values from 0 to 127 have character equivalents.

Notice that the storage and range associated with the **int** and **unsigned int** types are not defined by the C language. Instead, the size of an **int** (signed or unsigned) corresponds to the natural size of an integer on a given 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 either to the **short int** or the **long int** type, depending on the implementation. Similarly, the **unsigned int** type is equivalent either to the **unsigned short** or **unsigned long** type.

The **int** and **unsigned int** type specifiers are widely used in C programs because they allow a particular machine to handle integer values in the most efficient way for that machine. However, since the size of the **int** and **unsigned int** types varies, programs that depend on a specific **int** size may be nonportable. Expressions involving the **sizeof** operator (discussed in Section 5.3.4 of Chapter 5, "Expressions and Assignments") can be used in place of hard-coded data sizes to increase the portability of the code.

The type specifiers **int** and **unsigned int** (or simply **unsigned**) are used to define certain features of the C language (for instance, in defining the **enum** type later in Section 4.8.1). In these cases, the definition of **int** and **unsigned int** for a particular implementation determines the actual storage.

The range of values for a variable lists 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 Chapter 5, "Expressions and Assignments"), it is not always possible to use the maximum or minimum for a variable of a given type in an expression.

For example, the constant-expression −32,768 consists of the arithmetic negation operator (−) applied to the constant value 32,768. Since 32,768 is too large to represent as a **short**, it is given **long** type, and the constant-expression −32,768 consequently has **long** type. The value −32,768 can only be represented as a **short** 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 of storage space.

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, and is given **long** type because 65,000 does not fit into a **short**. This **long** value can then be cast to the **unsigned short** type without loss of information, since 65,000 will fit into 2 bytes of storage space when it is stored as an unsigned number.

Octal and hexadecimal constants are considered unsigned quantities, even though they are given **int** or **long** type, because they have the special property of representing a bit pattern. Thus, octal and hexadecimal constants are not sign-extended in type conversions.

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, a 7-bit excess 127 binary exponent, and a 24-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 an exponent range of 10 to the (+ or −) 38th power and up to seven digits of precision. The maximum value of a **float** is normally 1.701411E38.

Values with **double** type have 8 bytes. The format is similar to the **float** format, except that the exponent is 11 bits excess 1023, and the mantissa has 52 bits (plus the implied high-order 1 bit). This gives an exponent range of 10 to the (+ or −) 306th power and up to 15 digits of precision.

4.3 Declarators

Syntax

identifier
declarator[ ]
declarator[*constant-expression*]
**declarator*
declarator( )
declarator(*arg-type-list*)
(*declarator*)

C allows the programmer to declare *arrays* of values, *pointers* to values, and *function=returning* values of specified types. To declare these items, you must use a "declarator".

A declarator is an identifier possibly modified with brackets some combination of ([]), parentheses, and asterisks (*) to declare an array, pointer, or function type. Declarators appear in the pointer, array, and function declarations described in later sections of this chapter (Sections 4.4.6, 4.4.5, and 4.5, respectively). This section discusses the rules for forming and interpreting declarators.

4.3.1 Pointer, Array, and Function Declarators

When a declarator consists of an unmodified identifier, the item being declared has an unmodified type. Asterisks (*) can appear to the left of an identifier, modifying it to a *pointer* type. If the identifier is followed by brackets ([ ]), the type is modified to an *array* type. If the identifier is followed by parentheses, the type is modified to a *function-returning* type.

A declarator does not constitute a complete declaration; a type specifier must be included as well. The type specifier gives the type of the elements for an array type, the type of object addressed by a pointer type, and the return type of a function.

The sections on pointer, array, and function declarations later in this chapter discuss each type of declaration in detail (see Sections 4.4.6, 4.4.5, and 4.5, respectively). The following examples illustrate the simplest forms of declarators:

Examples

1. int list[20];

2. char *cp;

3. double func(void);

The above examples declare: 1. an array of **int** values (*list*); 2. a pointer to a **char** value (*cp*); and 3. a function with no arguments returning a **double** value (*func*).

4.3.2 Complex Declarators

Any declarator can be enclosed in parentheses. Parentheses are typically used to specify a particular interpretation of a "complex" declarator, as discussed below. A "complex" declarator is an identifier qualified by more than one array, pointer, or function modifier.

Various combinations of the array, pointer, and function modifiers can be applied to a single identifier. Some combinations are illegal. An array cannot be composed of functions, and a function cannot return an array or a function.

In interpreting complex declarators, brackets and parentheses (on the right of the identifier) take precedence over asterisks (on the left of the identifier). Brackets and parentheses have the same precedence and associate left to right. The type specifier is applied as the last step, when the declarator has been fully interpreted. Parentheses can be used to override the default association order in a way that forces a particular interpretation.

A simple rule that can be helpful in interpreting complex declarators is to read them "from the inside out." Start with the identifier and look to the right for brackets or parentheses. Interpret these (if any), then look to the left for asterisks. If you encounter a right parenthesis at any stage, go back and apply these rules to everything within the parentheses before proceeding. As the last step, apply the type specifier. To illustrate this rule, the steps are numbered in order in the following example:

```
char  *(*(*var)())[10];
^     ^ ^ ^ ^

 7    6 4 2 1  3  5
```

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.

The following examples provide further illustration and show how parentheses can affect the meaning of a declaration.

Examples:

1.
```
/* array of pointers to int values */
int  *var[5];
```

2.
```
/* pointer to array of int values */
int(*var)[5];
```

3.
```
/* function returning pointer to long */
long  *var(long,long);
```

4.
```
/* pointer to function returning long */
long  (*var)(long,long);
```

5.
```
/* array of pointers to functions
   returning structures */
struct both {
  int a;
  char b;
  } ( *var[ ])( struct both, struct both );
```

```
6. /* function returning pointer
      to an array of double values */
   double (*var(double (*)[3]))[3];

7. /* array of arrays of pointers
      to pointers to unions */
   union sign {
      int x;
      unsigned y;
      } **var[5][5];

8. /* array of pointers to arrays
      of pointers to unions */
   union sign *(*var[5])[5];
```

In the first 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; the elements are pointers to **int** values.

In the second example, parentheses alter the meaning of the declaration in the first example. Now the pointer modifier has higher priority than the array modifier, and *var* is declared to be a pointer to an array of 5 **int** values.

Function modifiers also have higher priority than pointer modifiers, so the third example declares *var* to be a pointer to a function returning a **long** value. The function is declared to take two **long** values as arguments.

The fourth example is similar to the second example. Parentheses give the pointer modifier higher priority than the function modifier, and *var* is declared to be a pointer to a function returning a **long** value. Again, the function takes two **long** arguments.

The elements of an array may not be functions, but the fifth example demonstrates how to declare an array of pointers to functions instead. In this example *var* is declared to be an array of pointers to functions returning structures with two members. The arguments to the functions are declared to be two structures with the same structure type, *both*. Notice that the parentheses surrounding "*var[ ]" are required. Without them, the declaration is an illegal attempt to declare an array of functions, as shown below:

```
        /* ILLEGAL */
   struct both *var[ ](struct both, struct both);
```

The sixth 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 3 **double** values. The function *var* takes one argument; the argument, like the return value, is a pointer to an array of 3 **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 3 pointers to **double** values. See Section 4.9, "Type Names," for a discussion and examples of abstract declarators.

A pointer can point to another pointer, and an array can contain array elements, as the seventh example shows. Here *var* is an array of 5 elements. Each element is a 5-element array of pointers to pointers to unions with two members.

The eighth example shows how the placement of parentheses alters the meaning of the declaration. In this example, *var* is a 5-element array of pointers to 5-element arrays of pointers to unions.

4.4 Variable Declarations

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 occupying the same storage space.
Arrays	Variables composed of a collection of elements with the same type.
Pointers	Variables that point to other variables. These variables contain variable locations (in the form of addresses) instead of values.

The variable declarations discussed in this section have the general form:

[*sc-specifier*] *type-specifier declarator* [, *declarator*...]

where *type-specifier* gives the data type of the variable and *declarator* is the variable's name, possibly modified to declare an array or a pointer type. More than one variable can be defined in the declaration by giving multiple declarators, separated by commas.

The *sc-specifier* gives the storage class of the variable. In some contexts, variables can be initialized when they are declared. Storage classes and initialization are discussed in Sections 4.6 and 4.7, respectively.

4.4.1 Simple Variable Declarations

Syntax

sc-specifier type-specifier identifier [, *identifier*...];

A declaration for a simple variable defines the variable's name and type. It can also define the variable's storage class, as described later in Section 4.6. The variable's name is the *identifier* given in the declaration. The *type-specifier* gives the name of a defined data type, as described below.

You can define several variables in the same declaration by giving a list of identifiers separated by commas (,). Each identifier in the list names a variable. All variables defined in the declaration have the same type.

Examples

1. int x;

2. unsigned long reply, flag;

3. double order;

The first example defines a simple variable *x*. This variable can hold any value in the set defined by the **int** type in a particular implementation.

The second example defines two variables, *reply* and *flag*. Both variables have **unsigned long** type and hold unsigned integer values.

The third example defines a variable *order* that has **double** type. Floating-point values can be assigned to this variable.

4.4.2 Enumeration Declarations

Syntax

enum [*tag*] {*enum-list*} *identifier*[, *identifier*...];
enum *tag identifier* [, *identifier*...];

An enumeration declaration gives the name of the enumeration variable and defines a set of named integer constants (the "enumeration set"). A variable declared to have enumeration type stores any 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.

Enumeration declarations begin with the **enum** keyword and have two forms, as shown above. In the first form, the values and names of the enumeration set are specified in the *enum-list*, described in detail below. The optional *tag* is an identifier that names the enumeration type defined by the *enum-list*. The *identifier* names the enumeration variable. More than one enumeration variable can be defined in the declaration.

The second form uses an enumeration *tag* to refer to an enumeration type. The *enum-list* does not appear in this type of declaration because the enumeration type is defined elsewhere. An error is generated if the given *tag* does not refer to a defined enumeration type or if the named type is not currently visible.

An *enum-list* has the following form:

> *identifier* [= *constant-expression*]
> [, *identifier* [= *constant-expression*]]
> .
> .
> .

Each *identifier* names a value of the enumeration set. By default, the first identifier is associated with the value zero, the next identifier is associated with the value one, and so on through the last identifier appearing in the declaration. The name of an enumeration constant is equivalent to its value.

The phrase "= *constant-expression*" overrides the default sequence of values. An identifier followed by the phrase "= *constant-expression*" 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 it is explicitly given another value.

An enumeration set can contain duplicate constant values, but each identifier in an enumeration list must be unique, that is, different from all other enumeration identifiers with the same visibility. For example, the value zero (0) could be given to two different identifiers, *null* and *zero*, in the same set. The identifiers in the list must also 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.

Examples

```
1. enum day {
   saturday,
   sunday=0,
   monday,
   tuesday,
   wednesday,
   thursday,
   friday
   } workday;

2. today=wednesday;

3. enum day holiday;
```

The first example defines an enumeration type named *day* and declares a variable named *workday* with that enumeration type. The value 0 is associated with *saturday* by default. The identifier *sunday* is explicitly set to 0. The remaining identifiers are given the values 1 through 5 by default.

In the second example, a value from the set is assigned to the variable *today*. Notice that the name of the enumeration constant is used to assign the value.

In the third example, a variable named *holiday* is declared to have the enumeration type *day*. Since the *day* type was previously declared, only the enumeration tag is necessary in this declaration.

4.4.3 Structure Declarations

Syntax

struct [*tag*] {*member-declaration-list*} *declarator* [, *declarator*...];
struct *tag declarator* [, *declarator*...];

A structure declaration defines the name of the structure variable and specifies a sequence of variable values (called "members" of the structure) that can have different types. A variable with structure type holds the entire sequence defined by that type.

Structure declarations begin with the **struct** keyword and have two forms, as shown above. In the first form, the types and names of the structure members are specified in the *member-declaration-list*, described in detail below. The optional *tag* is an identifier that names the structure type defined by the *member-declaration-list.*

Each *declarator* gives the name of a structure variable. The *declarator* may also modify the type of the variable to a pointer to the structure type, an array of structures, or a function returning a structure.

The second form uses a structure *tag* to refer to a structure type. The *member-declaration-list* does not appear in this type of declaration because the structure type is defined elsewhere. The structure type definition must be visible for a *tag* declaration to be used, and the definition must appear prior to the *tag* declaration, unless the *tag* is used to declare a pointer variable or a **typedef** structure type. These declarations can use a structure *tag* before the structure type is defined, as long as the structure definition is visible to the declaration.

A *member-declaration-list* is a list of 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 member declaration lists have the same form as the variable declarations discussed in this chapter, except that the declarations do not contain storage class specifiers or initializers. The structure members can have any variable type: fundamental, array, pointer, union, or structure.

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. This allows you to create linked lists of structures.

Bitfields

A bitfield declaration has the following form:

type-specifier [*identifier*] : *constant-expression*;

The bitfield consists of the number of bits specified by *constant-expression.* The *type-specifier* for a bitfield declaration must specify an unsigned integral type, and the *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. An unnamed bitfield whose width is specified as zero (0) has a special function: it guarantees that storage for the member following it in the declaration list begins on an **int** boundary.

The identifiers in a structure declaration list must be unique within that list. It is not necessary for the identifiers in the list to be distinct from ordinary variable names or from identifiers in other structure declaration lists. Structure tags must be distinct from other structure, union, and enumeration tags having the same visibility.

Structure members are stored sequentially in the same order in which they are declared. The first member has the lowest memory address and the last member the highest. The storage for each member begins on a memory boundary appropriate to its type. Therefore, unnamed blanks can occur between the members of a structure in memory.

Bitfields are not stored across boundaries of their declared type. For example, a bitfield declared with **unsigned int** type is either packed into the space remaining in the previous **int** or it begins a new **int**.

Examples

```
1. struct {
       float x,y;
   } complex;

2. struct employee {
       char name[20];
       int id;
       long class;
   } temp;

3. struct employee student, faculty, staff;

4. struct sample {
       char c;
       float *pf;
       struct sample *next;
   } x;

5. struct {
       unsigned icon : 8;
       unsigned color : 4;
       unsigned underline : 1;
       unsigned blink : 1;
   } screen[25][80];
```

The first example defines a structure variable named *complex*. This structure has two members with **float** type, *x* and *y*. The structure type is not named.

The second 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.

The third 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 the previous example.

The fourth 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*).

The fifth example defines a two-dimensional array of structures named *screen*. The array contains 2,000 elements. Each element is an individual structure containing four bitfield members: *icon*, *color*, *underline*, and *blink*.

4.4.4 Union Declarations

Syntax

> **union** [*tag*] {*member-declaration-list*} *declarator* [, *declarator*...];
> **union** *tag declarator*[, *declarator*...];

A union declaration defines the name of the 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 any single value defined by that type.

Union declarations have the same forms 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.

The storage associated with a union variable is the storage required for the longest member of the union. When a smaller member is used, 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.

Examples

```
1. union sign {
     int svar;
     unsigned uvar;
   } number;

2. union {
     char *a, b;
     float f[20];
   } jack;

3. union {
     struct {
         char icon;
         unsigned color : 4;
     } window1, window2, window3, window4;
   } screen[25][80];
```

The first example defines a union 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 union type is named *sign*.

The second example defines a union variable named *jack*. The members of the union are, in order, 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. The union type is unnamed.

The third example defines a two-dimensional array of unions named *screen*. The array contains 2,000 elements. Each element is an individual union with four members: *window1*, *window2*, *window3*, and *window4*, where each member is a structure. Each union element holds one of the four possible structure members at any given time. Thus, the *screen* variable is a composite of up to four different "windows."

4.4.5 Array Declarations

Syntax

type-specifier declarator [*constant-expression*];
type specifier declarator [ ];

A declaration for an array defines the name of the array and the type of each element. It can 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 Section 5.2.2 of Chapter 5, "Expressions and Assignments".

Array declarations have two forms, as shown above. The *declarator* gives the variable name, and may modify the variable's type. The brackets ([]) following the *declarator* modify the declarator to array type. The *constant-expression* inside the brackets defines the number of elements in the array. Each element has the type given by the *type-specifier*. The *type-specifier* can specify any type except **void** and function types.

The second form omits the *constant-expression* in brackets. This form can be used only if the array is initialized, declared as a formal parameter, or declared as a reference to an array explicitly defined elsewhere in the program.

Arrays of arrays ("multidimensional" arrays) are defined by giving a list of bracketed *constant-expressions* following the array declarator.

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 a multidimensional array is declared within a function, the first *constant-expression* can be omitted if the array is initialized, declared as a formal parameter, or declared as a reference to an array explicitly defined elsewhere in the program.

Arrays of pointers to various types can be defined by using complex declarators, as described earlier in Section 4.3.2.

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 occur between the elements of an array in storage.

Arrays are stored by row. For example, the following array consists of 2 rows with 3 columns each:

```
char A[2][3];
```

The 3 columns of the first row are stored first, followed by the 3 columns of the second row.

To refer to an individual element of an array, use a subscript expression, discussed in Section 5.2.5 of Chapter 5, "Expressions and Assignments".

Examples

```
1. int scores[10], game;

2. float matrix[10][15];

3. struct {
       float x,y;
       }complex[100];

4. char *name[20];
```

The first example defines 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.

The second example defines a two-dimensional array named *matrix*. The array has 150 elements, each having **float** type.

The third example defines an array of structures. This array has 100 elements. Each element is a structure containing two members.

The fourth example defines an array of pointers. The array has 20 elements. Each element is a pointer to a **char** value.

4.4.6 Pointer Declarations

Syntax

type-specifier **declarator*;

A pointer declaration defines the name of the pointer variable and the type of the object to which the variable points. The *declarator* defines the variable's name, and may modify its type. The *type-specifier* gives the type of the object. The type can be any fundamental, structure, or union type.

Pointer variables can also point to functions, arrays, and other pointers. To declare more complex pointer types, refer to Section 4.3.2.

A pointer to a structure or union type can be declared before the structure or union type is defined, as long as the structure or union type definition is visible at the time of the declaration. 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. The pointer is declared by using the structure or union tag. See the fourth example below.

A variable declared as a pointer holds a memory address. The amount of storage required for an address and the meaning of the address depends on the given implementation of the compiler. Pointers to different types are not guaranteed to have the same length.

In some implementations the special keywords **near** and **far** are available to modify the size of a pointer. See your system documentation for more information.

Examples

```
1. char *message;

2. int *pointers[10];

3. int(*pointer)[10];

4. struct list *next, *previous;

5. struct list {
     char *token;
     int count;
     struct list *next;
   } line;
```

The first example defines a pointer variable named *message*. It points to a variable with **char** type.

The second example defines an array of pointers named *pointers*. The array has 10 elements. Each element is a pointer to a variable with **int** type.

The third example defines a pointer variable named *pointer*. It points to an array with 10 elements. Each element in this array has **int** type.

The fourth example defines two pointer variables that point to the structure type *list*. This declaration can appear before the definition of the *list* structure type (see the next example), as long as the *list* type definition has the same visibility as the declaration.

The fifth example declares the variable *line* to have the structure type named *list*. The *list* structure type is defined to have 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.

4.5 Function Declarations

Syntax

[*type-specifier*] *declarator*([*arg-type-list*]) [, *declarator*...];

A function declaration defines the name and return type of a function, and possibly establishes the types and number of arguments to the function. Function declarations, also called forward declarations, do not define the function body or parameters. Instead they permit the attributes of the function to be known before the function is defined. Function definitions are described in detail in Section 7.2 of Chapter 7, "Functions"."

The *declarator* of the function declaration names the function, and the *type-specifier* gives the function's return type. If the *type-specifier* is omitted from a function declaration, the return type of the function is assumed to be **int**.

Function declarations may include either the **extern** or the **static** storage class specifier. Storage class specifiers are discussed in Section 4.6.

Argument Type List

The *arg-type-list* establishes the number and types of the arguments to the function. It has the following form:

[*type-name*][, *type-name*...][,]

The first *type-name* gives the type of the first argument to the function, the second *type-name* gives the type of the second argument, and so on. Each *type-name* is separated from the next by a comma. If the *arg-type-list* ends with a comma, the number of arguments to the function is variable. However, the function is expected to have at least as many arguments as there are *type-names* before the terminating comma. If the *arg-type-list* contains only a single comma, the number of arguments to the function is variable and may be zero.

A *type-name* for a fundamental, structure, or union type consists of the type specifier for that type (such as **int**). The *type-names* for pointers, arrays, and functions are formed by combining a type specifier with an "abstract declarator," that is, a declarator without an identifier. Section 4.9 explains how to form and interpret abstract declarators.

The special keyword **void** can be used in place of the *arg-type-list* to declare a function that has no arguments. The compiler displays a warning message if a call to the function or the function definition specifies arguments.

One other special construction is allowed in the *arg-type-list*. The phrase **void *** specifies an argument of any pointer type. This phrase can be used in the *arg-type-list* as if it were a *type-name*.

The *arg-type-list* may be omitted altogether. The parentheses after the function identifier are still required, but they are empty. In this form, the function declaration establishes neither the number nor the types of arguments to the function. When this information is omitted, the compiler does not perform any type-checking between the actual arguments in a function call and the formal parameters of the function definition. See Section 7.4 of Chapter 7, "Functions", for details.

Return Type

Functions can return values of any type except arrays and functions. Thus, the *type-specifier* of a function declaration can specify any fundamental, structure, or union type. The function identifier can be modified with one or more asterisks (*) to declare a pointer return type.

Although functions are not allowed to return arrays and functions, they can return pointers to arrays and functions. Functions that return pointers to array or function types are declared by modifying the function identifier with asterisks (*), brackets ([ ]), and parentheses to form a complex declarator. Forming and interpreting complex declarators is discussed in Section 4.3.2.

Examples

```
1. int add(int, int);

2. char *strfind(char *,);

3. void draw(void);

4. double (*sum(double, double))[3];

5. int (*(*select)(void))(int)
```

```
6. char *p;
   short *q;
   int prt(void *);
```

The first example declares a function named *add* that takes two **int** arguments and returns an **int** value.

The second example declares a function named *strfind*, which returns a pointer to a **char** value. The function takes at least one argument, a pointer to a **char** value. The argument type list ends with a comma, indicating that the function may take more arguments.

The third example declares a function with **void** return type (returning no value). The *argument-type-list* is also **void**, meaning no arguments are expected for this function.

In the fourth example, *sum* is declared as a function returning a pointer to an array of 3 **double** values. The *sum* function takes two arguments, each a **double** value.

In the fifth example, the function named *select* is declared to return a pointer to a function taking no arguments and returning a pointer. The pointer return value points to a function taking one **int** argument and returning an **int** value.

In the sixth example, the function *prt* is declared to take a pointer argument of any type and to return an **int**. Either the **char** pointer *p* or the **short** pointer *q* could be passed as an argument to *prt* without producing a type mismatch warning.

4.6 Storage Classes

The storage class of a variable determines whether the item has a "global" or "local" lifetime. An item with a global lifetime exists and has a value througbout the duration of the program. All functions have global lifetimes.

Variables with local lifetimes are allocated new storage each time execution control passes to the block in which they are defined. When execution passes out of the block, the variables no longer have meaningful values.

Althougb C defines only two types of storage classes, four storage class specifiers are available. They are:

auto
register
static
extern

Items with **auto** and **register** class have local lifetimes. The **static** and **extern** specifiers refer to items with global lifetimes.

The four storage class specifiers have distinct meanings because storage class specifiers 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. 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 Section 3.5 of Chapter 3, "Program Structure."

The placement of variable or function declarations within source files also affects storage class and visibility. Declarations outside of all function definitions are said to occur at the "external level;" declarations within function definitions occur at the "internal level."

The exact meaning of each storage class specifier depends on whether the declaration occurs at the external or the internal level and whether the item declared is a variable or a function. The following sections describe the meaning of storage class specifiers in each kind of declaration. They also explain the default behavior when the storage class specifier is omitted from a variable or function declaration.

4.6.1 Variable Declarations at the External Level

Variable declarations at the external level use the **static** and **extern** storage class specifiers or omit the storage class specifier entirely. The **auto** and **register** storage class specifiers are not allowed at the external level.

Variable declarations at the external level are either *definitions* of variables or *references* to variables defined elsewhere. An external variable declaration that also initializes the variable (implicitly or explicitly) is a definition of the variable. Definitions at the external level can take several forms:

1. A variable can be defined at the external level by declaring it with the **static** storage class specifier. The **static** variable can be explicitly initialized, as described in Section 4.7. If the initializer is omitted, the variable is automatically initialized to zero at compile time. Thus, "static int k = 16;" and static int k; are both considered definitions.
2. A variable is defined when it is explicitly initialized at the external level. For example, "int j = 3;" is a variable definition.

Once a variable is defined at the external level, it is visible throughout the remainder of the source file in which it appears. The variable is not visible above its definition in the same source file, nor is it visible in other source files of the program, unless a *reference* is declared to make it visible, as described below.

A variable can be defined at the external level only once within a source file. If the **static** storage class specifier is given, another variable with the same name can be defined with the **static** storage class specifier in a different source file. Since each **static** definition is visible only in its own source file, no conflict occurs.

The **extern** storage class specifier is used to declare a *reference* to a variable defined elsewhere. These declarations can be used to make a definition in another source file visible or to make a variable visible above its definition in the same source file. Once a reference to the variable is declared at the external 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 are not allowed to contain initializers, since they refer to variables whose values are already defined.

For an **extern** reference to be valid, the variable to which it refers must be defined once, and only once, at the external level. The definition can be in any of the source files that make up the program.

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 external level. For example, the declaration "int n;" is a valid external declaration. This declaration can have one of two different meanings, depending on the context:

1. If a variable by the same name is *defined* at the external level elsewhere in the program, the declaration is taken to be a reference to that variable, exactly as if the **extern** storage class specifier had been used in the declaration.
2. If no such definition is present, the declared variable is allocated storage at link time and initialized to zero. 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 external level, "int i;" and "char i;", storage space for an **int** is allocated for *i* at link time.

Example

```
/***************************************************
      SOURCE FILE ONE
***************************************************/

extern int  i;
      /* reference to i, defined below */

main()
{
   i++;
      /* i equals 4 */
   next();
}

int  i=3;
      /* definition of i */

next()
{
   i++;
   printf("%d\n", i);
         /* i equals 5 */
   other();
}

/***************************************************
      SOURCE FILE TWO
***************************************************/

extern int  i;
      /* reference to i in first source file */

other()
{
   i++;
   printf("%d\n", i);
      /* i equals 6 */
}
```

The two source files contain a total of three external 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 makes the global variable visible in that source file.

All three functions perform the same task: they increase *i* and print it. (Assume that the **printf** function is defined elsewhere in the program.) The values printed are 4, 5, and 6.

If the variable *i* had not been initialized, it would have been automatically set to zero at link time. The values printed in this case would be 1, 2, and 3.

4.6.2 Variable Declarations at the Internal Level

Any of the four storage class specifiers can be used for variable declarations at the internal level. When the storage class specifier is omitted from a variable declaration at the internal level, the default storage class is **auto.**

The **auto** storage class specifier declares a variable with a local lifetime. The variable is visible only in the block in which it is declared. Declarations of **auto** variables can include initializers, as discussed later in this chapter. Variables with **auto** storage class are not initialized automatically, so they should be explicitly initialized when declared or assigned initial values in statements within the block. If not initialized, the values of **auto** variables are undefined.

The **register** storage class specifier tells the compiler to give the variable storage in a register, if possible. Register storage usually results in faster access time and smaller 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 the **register** declaration, the variable is given **auto** storage class and stored in memory. The compiler assigns register storage to variables in exactly the same order in which the declarations appear in the source file. Register storage (if available) is only guaranteed for **int** and pointer types.

A variable declared at the internal level with the **static** storage class specifier has a global lifetime. The variable is visible only within the block in which it is declared. Unlike **auto** variables, variables declared as **static** retain their values when the block is exited.

Declarations of **static** variables can include initializers. If not explicitly initialized, a **static** variable is automatically set to zero. Initialization is performed once, at compile time; the **static** variable is *not* reinitialized each time the block is entered.

A variable declared with the **extern** storage class specifier is a reference to a variable with the same name defined at the external level in any of the source files of the program. The purpose of the internal **extern** declaration is to make the external-level variable definition visible within the block.

The internal **extern** declaration does not change the visibility of the global variable in any other part of the program.

Example

```
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()
{
    /* i is redefined */
    int  i = 16;

    /* this a is visible only within other */
    static int  a = 2;

    a += 2;
    /* values printed are 16, 4 */
    printf("%d\n%d\n", i, a);
}
```

The variable *i* is defined at the external level with initial value 1. A reference to the external-level *i* is declared in the *main*() function with an **extern** declaration. The **static** variable *a* is automatically set to zero, since the initializer is omitted. The call to **printf** (assuming the **printf** function os defined elsewhere in the source program) prints out the values 1, 0, 0, 0.

In the *other* function, the variable *i* is redefined as a local variable with initial value 16. This does not affect the value of the external-level *i*. The variable *a* is declared as a **static** variable and initialized to 2. This *a* does not conflict with the *a* in *main*, since the visibility of **static** variables at the internal level is restricted to 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. Internal **static** variables retain their values when the block in which they are declared is exited and reentered.

4.6.3 Function Declarations

Function declarations can use either the **static** or the **extern** storage class specifier. Functions always have global lifetimes.

The visibility rules for functions are slightly different from the rules for variables. Function declarations at the internal level have the same meaning as function declarations at the external level. This means that functions cannot have block visibility, and the visibility of functions cannot be nested. A function declared to be **static** is visible only within the source file in which it is defined. Any function in the same source file can call the **static** function, but functions in other source files cannot. Another **static** function by the same name can be declared in a different source file without conflict.

Functions declared as **extern** are visible throughout all the source files that constitute the program (unless they are later redeclared as **static**). Any function can call an **extern** function.

Function declarations that omit the storage class specifier default to **extern**.

4.7 Initialization

A variable can be set 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. The initializer is preceded by an equal sign (=), as shown below:

= *initializer*

Variables of any type can be initialized, with the restrictions outlined below. Functions do not take initializers.

Declarations that use the **extern** storage class specifier cannot contain initializers.

Variables declared at the external level can be initialized; if not explicitly initialized, they are set to zero at compile time. Any variable declared with the **static** storage class specifier can be initialized. Initializations of **static** variables are performed once, at compile time. If not explicitly initialized, **static** variables are automatically set to zero.

Initializations of **auto** and **register** variables are performed each time execution control passes to the block in which they are declared. If the initializer is omitted from the declaration of an **auto** or **register** variable, the initial value of the variable is undefined.

Initializations of **auto** aggregate types (arrays, structures, and unions) are prohibited. Only **static** aggregates and aggregates declared at the external level can be initialized.

The initial values for external variable declarations and for all **static** variables, whether external or internal, must be constant-expressions. Constant-expressions are described in Section 5.2.10 of Chapter 5, "Expressions and Assignments." Automatic and register variables can be initialized with either constant or variable values.

Sections 4.7.1 and 4.7.2 describe how to initialize variables of fundamental, pointer, and aggregate types.

4.7.1 Fundamental and Pointer Types

Syntax

var= *expression*

The value of *expression* is assigned to the variable. The conversion rules for assignment apply.

Examples

1. int x=10;

2. register int *px=0;

3. `int c = (3 * 1024);`

4. `int *b = &x;`

In the first example, *x* is initialized to the constant-expression 10. In the second example, the pointer *px* is initialized to zero, producing a "null" pointer. The third example uses a constant-expression to initialize *c*. The fourth example initializes the pointer *b* with the address of another variable, *x*.

4.7.2 Aggregate Types

Syntax

var= {*initializer-list*}

An *initializer-list* is a list of initializers separated by commas. Each initializer in the list is either a constant-expression or an *initializer-list*. Thus, a brace-enclosed list can appear within another *initializer-list*. This is useful for initializing aggregate members of an aggregate, as shown in the examples below.

For each *initializer-list*, the values of the constant-expressions are assigned in order to the members of the aggregate variable. When a union is initialized, the *initializer-list* must be a single constant-expression. The value of the constant-expression is assigned to the first member of the union.

If there are fewer values in an *initializer-list* than there are in the aggregate type, the remaining members or elements are initialized to zero. Giving too many initial values for the aggregate type causes an error. These rules apply to each embedded *initializer-list*, as well as to the aggregate as a whole.

For example:

```
int P[4][3]={
    {1,1,1},
    {2,2,2},
    {3,3,3,},
    {4,4,4,},
};
```

declares *P* as a 4x 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. Notice 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 are not required; the required commas are those that separate constant-expressions and *initializer-lists.*

If there is no embedded initializer list for an aggregate member, values are simply assigned in order to each member of the subaggregate. Thus, the above initialization is equivalent to:

```
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.

Examples

```
1. struct list {
   int  i, j, k;
   float m[2][3];
   } x = {
       1,
       2,
       3,
       {4.0, 4.0, 4.0}
   };

2. union {
   char  x[2][3];
   int  i, j, k;
   } y = {
       {'1'},
       {'4'}
   };
```

In the first 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 zero by default.

In the second 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 "{'1'}" gives values to the first row of the array. Since only one value appears in the list, the element in the first column is initialized to the character "1", and the remaining two elements in the row are initialized to zero (the null character), 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 zero.

4.7.3 String Initializers

An array can be initialized with a string literal. For example:

```
    char code[ ]= "abc";
```

initializes *code* as a four-element array of characters. The fourth element is the null character that terminates all string literals.

If the array size is specified and the string is longer than the specified size of the array, the extra characters are simply discarded. 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 null character are discarded. Note that some compilers return a warning message when this happens.

If the string is shorter than the specified size of the array, the remaining elements of the array are initialized to zero (the null character).

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. The name of a declared type can be used 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. These declarations are used to set up shorter or more meaningful names for types already defined by C or for types declared by the user.

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. In type declarations, the variable identifier is omitted, since no variable is declared. The *tag* is mandatory; 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 a *tag* refers to a type defined elsewhere, is not legal for type declarations.

Examples:

1.
   ```
   enum status{
   loss = -1,
   bye,
   tie=0,
   win
   };
   ```

2.
   ```
   struct student{
   char name[20];
   int id, class;
   };
   ```

The first 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.

The second example declares a structure type named *student*. A structure variable can be declared to have *student* type with a declaration such as "struct student employee;".

4.8.2 typedef Declarations

Syntax

typedef *type-specifier declarator* [, *declarator*...];

A **typedef** declaration is analogous to a variable declaration except that the **typedef** keyword appears in place of a storage class specifier. The declaration is interpreted in the same way as variable and function declarations, but the identifier, instead of taking on the type specified by the declaration, becomes a new keyword for the type.

typedef does not create types. It creates synonyms for existing types or names for types that could be specified in other ways. Any type can be declared with **typedef**, including pointer, function, and array types. A

typedef name for a pointer to a structure or union type can be declared before the structure or union type is defined, as long as the definition has the same visibility as the declaration.

Examples

```
1. typedef int WHOLE;

2. typedef struct club {
   char name[30];
   int size, year;
   } GROUP;

3. typedef GROUP *PG;

4. typedef void DRAWF(int, int);
```

The first example declares *WHOLE* to be a synonym for **int**.

The second 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.

The third 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.

The final example 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);".

4.9 Type Names

A "type name" specifies a particular data type. Type names are used in three contexts: in the argument type lists of function declarations, in type casts, and in **sizeof** operations. Argument type lists are discussed in Section 4.5. Type casts and **sizeof** operations are discussed in Section 5.7.2 and 5.3.4, respectively, of Chapter 5, "Expressions and Assignments".

The type names for fundamental, enumeration, structure, and union types are simply the type specifiers for those types.

A type name for a pointer, array, or function type has the form:

type-specifier abstract-declarator

An *abstract declarator* is a declarator without an identifier, consisting solely of one or more pointer, array, or function modifiers. The pointer modifier (*) always appears before the identifier in a declarator, while array ([]) and function (()) modifiers appear after the identifier. It is thus possible to determine where the identifier would appear in an abstract declarator and to interpret the declarator accordingly.

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.

When you give a function type with an abstract declarator, you can include the function's argument type list, which also consists of type names. See the second and fourth examples below.

The abstract declarator "()" alone 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 (a function type).

The type specifiers established through **typedef** declarations also qualify as type names.

Examples:

1. long*

2. double *(double, double)

3. int (*)[5]

4. int (*)(void)

The first example gives the type name for "pointer to **long**" type.

The second example is the type name for a function that takes two **double** arguments and returns a pointer to a **double** value.

The third and fourth examples show how parentheses modify complex abstract declarators. Example 3 gives the name for a pointer to an array of five **int** values. Example 4 names a pointer to a function taking no arguments and returning an **int**.

Chapter 5

Expressions and Assignments

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. Operators specify how the operand or operands of the expression are manipulated.

In C, assignments are considered expressions. 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.

The value resulting from an expression's evaluation depends on the relative precedence of operators in the expression and side effects, if present. The precedence of operators determines the grouping of operands in an expression. 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.

The value represented by each operand in an expression has a type, which may be converted to a different type in certain contexts. Type conversions take place in assignments, type casts, function calls, and operations.

5.2 Operands

A C operand is a constant, an identifier, a string, a function call, a subscript expression, a member selection expression, or a more complex expression 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 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.

5.2.1 Constants

A constant operand has the value and type of the constant value it represents. A character constant has **char** type. An integer constant can have either **int** or **long** type, depending on the integer's size and how the value was 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, which is established when the identifier is declared. The value of an identifier depends upon its type, as follows:

- Identifiers of integral and floating-point types represent values of the corresponding type.
- An identifier of **enum** type represents one constant value of 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 the specified type.
- An identifier declared as an array represents a pointer whose value is the address of the first element of the array. The type addressed by the pointer is the type of the first element of the array. For example, if *series* is declared to be a ten-element integer array, the identifier *series* expresses the address of the array, while the subscript expression "series[*n*]" (where *n* is an integer in the range zero to nine) refers to a variable integer element of *series*. Subscript expressions are discussed in Section 5.2.5.

  The address of an array does not change during the execution of the program, although the values of the individual elements can change. The pointer value represented by an array identifier is not a variable, and 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 type addressed by the pointer is a function returning a value of a specified type. The address of a function does not change during the execution of a program; only the return value varies. Thus, function identifiers cannot be left-hand operands in assignment operations.

5.2.3 Strings

A string literal consists of a list of characters enclosed in double quotes, as shown below:

"*string*"

A string literal is stored as an array of elements with **char** type. The string literal represents the address of the first element of the array. 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 contexts that allow pointer values, and they are subject to the same restrictions as pointers. String literals have one additional restriction: they are not variables and cannot be left-hand operands in assignment operations.

The last character of a string is always the null character, "\0". The null character is not visible in the string expression, but it is added as the last element when the string is stored. Thus, the string abc actually has four characters rather than three.

5.2.4 Function Calls

Syntax

expression (*expression-list*)

A function call consists of an *expression* followed by an *expression-list* in parentheses, where *expression* evaluates to a function address (for example, a function identifier), and *expression-list* is a list of expressions whose values, the actual arguments, are passed to the function. The *expression-list* 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**, the function call expression also has **void** type. If control returns from the called function without execution of a **return** statement, the value of the function call is undefined.

See Section 7.4 of Chapter 7, "Functions," for a detailed discussion of 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*. *expression1* is any pointer value (such as an array identifier) and *expression2* is an integer value. *expression2* must be enclosed in brackets ([ ]).

Subscript expressions are generally used to refer to array elements, but a subscript can be applied to any pointer.

The subscript expression is evaluated by adding the integer value (*expression2*) to the pointer value (*expression1*), 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 of tbe addition operator (see Section 5.3.6), the integer 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. To evaluate the expression *line*[*i*], the integer value *i* is multiplied by the length of an **int**. The converted value of *i* represents *i* **int** positions. This converted value is added to the original pointer value (*line*) to yield an address that is offset *i* **int** positions from *line*.

As the last step in evaluating the subscript expression, the indirection operator is applied to the new address. The result is the value of the array element at that position (intuitively, *line*[*i*]).

Notice that the subscript expression:

```
line[0]
```

yields the value of the first element of a *line*, since the offset from the address represented by a *line* is zero. Similarly, an expression such as:

```
line[5]
```

refers to the element offset five positions from *line*, or the sixth element of the array.

Multidimensional Array References

A subscript expression can be subscripted, as follows:

expression1 [*expression2*] [*expression3*]...

Subscript expressions associate left to right. The leftmost subscript expression, *expression1*[*expression2*], is evaluated first. The address that results from adding *expression1* and *expression2* forms the pointer expression to which *expression3* is added. The indirection operator (*) is applied after the last subscripted expression is evaluated. However, the indirection operator is not applied at all if the final pointer value addresses an array type. See the third example below.

Expressions with multiple subscripts refer to elements of multidimensional arrays. A multidimensional array is an array whose elements are arrays. The first element of a three-dimensional array, for example, is an array with two dimensions.

Examples:

```
int prop[3][4][6];
int  i, *ip;

1. i = prop[0][0][1];

2. i = prop[2][1][3];

3. ip = prop[2][1];
```

The array named *prop* has 3 elements, each of which is a 4-by-6 array of **int** values.

Example 1 shows how to refer to the second individual **int** element of *prop*. Arrays are stored by row, so the last subscript varies fastest.

The second example shows a more complex reference to an individual element of *prop*. To evaluate the expression, 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 a 6-element array, the third element of the selected 4-by-6 array.

Next, the second subscript, 1, is multiplied by the size of the 6-element **int** array and added to the address represented by *prop*[2].

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.

The last step in evaluating the expression *prop*[2] [*1*] [*3*] applies the indirection operator to the pointer value. The result is the **int** element at that address.

Example 3 shows a case where the indirection operator is not applied. The expression *prop*[2][*1*] is a valid reference to the 3-dimensional array *prop*; the result of the expression is a pointer value that addresses an array with 1 dimension. Since the pointer value addresses an array type, the indirection operator is not applied.

5.2.6 Member Selection Expressions

Syntax

expression.identifier
expression->identifier

Member selection expressions refer to members of structures and unions. A member selection expression has the value and type of the selected member.

In the first form, "*expression.identifier*", *expression* represents a value of **struct** or **union** type. The *identifier* names a member of the specified structure or union.

In the second form, "*expression->identifier*", *expression* represents a pointer to a structure or union. The *identifier* names a member of the specified structure or union.

The two forms of member selection expressions have a similar effect. In fact, expressions involving the pointer selection operator (->) are shorthand versions of expressions using the period (.) in cases where the expression before the period consists of the indirection operator (*) applied to a pointer value. (The indirection operator is discussed in Section 5.3.3.) Thus:

expression->identifier

is equivalent to:

(**expression*).*identifier*

when *expression* is a pointer value.

Examples

```
struct pair {
    int a;
    int b;
    struct pair *sp;
    } item, list[10];
```

1. item.sp = &item;

2. (item.sp)->a=24;

3. list[8].b = 12;

In the first 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.

In the second example, the pointer expression "item.sp" is used with the pointer selection operator (->) to assign a value to the member *a*.

The third example shows how to select an individual structure member from an array of structures.

5.2.7 Expressions with Operators

Expressions with operators can be unary, binary, or ternary expressions. A unary expression consists of an operand prefixed by an unary operator ("unop") or an operand enclosed in parentheses and preceded by the **sizeof** keyword:

unop operand
sizeof (*operand*)

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

Assignment expressions 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. The forms of assignment expressions are:

operand++
operand−−
++operand
−−operand
operand = operand
operand compound-assignment-op operand

5.2.8 Expressions in Parentheses

Any operand can be enclosed in parentheses. The parentheses have no effect on the type or value of the enclosed expression. For example, in the expression:

```
(10+5)/5
```

the parentheses around "10 + 5" mean that the value of "10 + 5" is the left operand of the "/" (division) operator. 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 for the expression.

5.2.9 Type-Cast Expressions

A type-cast expression has the following form:

(type-name)operand

Type-cast conversions are discussed in Section 5.7.2; type names are discussed in Section 4.9 of Chapter 4, "Declarations"."

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-point constants, enumeration constants, type casts to integral and floating-point types, and other constant-expressions. The operands can be combined and modified using operators, as described in Section 5.2.7, with some restrictions.

Constant-expressions may not use assignment operators (see Section 5.4) or the binary sequential evaluation operator (,). The unary address-of operator (&) can be used only in certain initializations (see the last paragraph of Section 5.2.10).

Constant-expressions used in preprocessor directives are subject to additional restrictions, and are consequently known as *restricted-constant-expressions.* A *restricted-constant-expression* cannot contain **sizeof** expressions, enumeration constants, or type casts to any type. It can, however, contain the special constant-expression "defined(*identifier*)". See Section 8.2.1 of Chapter 8, "Preprocessor Directives", for details.

These additional restrictions also apply to constant-expressions used to initialize variables at the external level. However, such expressions are allowed to apply the unary address-of operator (&) to other external-level variables with fundamental, structure, and union types and to external-level arrays subscripted with a constant-expression. In these expressions, a constant-expression not involving the address-of operator can be added to or subtracted from the address expression.

5.3 Operators

C operators take one operand (unary operators), two operands (binary operators), or three operands (the ternary operator).

Unary operators prefix their operand and associate right to left. C's unary operators are:

- ~ !	Complement operators
* &	Indirection and address-of operators
sizeof	Size operator

Binary operators associate left to right. The binary operators are:

* / %	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 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-point types. These conversions are known as "arithmetic" conversions because they apply to the types of values ordinarily used in arithmetic.

The arithmetic conversions summarized below are called the "usual arithmetic conversions." The discussion of each operator in the following sections specifies whether the operator performs the usual arithmetic conversions and also specifies the additional conversions, if any, the operator performs.

The specific path of each type of conversion is outlined in Section 5.7.

The usual arithmetic conversions proceed in order as follows:

1. Any operands of **float** type are converted to **double** type.
2. If one operand has **double** type, the other operand is converted to **double**.
3. Any operands of **char** or **short** type are converted to **int**.
4. Any operands of **unsigned char** or **unsigned short type** are converted to **unsigned int** type.
5. If one operand is of type **unsigned long**, the other operand is converted to **unsigned long**.
6. If one operand is of type **long**, the other operand is converted to **long**.
7. If one operand is of type **unsigned int**, the other operand is converted to **unsigned int**.

5.3.2 Complement Operators

Arithmetic Negation (-)

The arithmetic negation operator (-) produces the negative (two's complement) of its operand.

The operand must be an integral or floating-point value.

The usual arithmetic conversions are performed.

Bitwise Complement (~)

The bitwise complement operator (~) produces the bitwise complement of its operand. The operand must be of integral type. The usual arithmetic conversions are performed. The result has the type of the operand after conversion.

Logical NOT (!)

The logical NOT operator (!) produces the value zero if its operand is true (nonzero) and the value one if its operand is false (zero). The result has **int** type. The operand must be an integral, floating-point, or pointer value.

Examples:

```
1. short x = 987;
   x = -x;

2. unsigned short y = 0xaaaa;
   y = ~y;

3. if( !(x < y));
```

In the first example, the new value of *x* is the negative of 987, or -987.

In the second example, the new value assigned to *y* is the one's complement of the unsigned value 0xaaaa, or 0x5555.

In the third example, if *x* is greater than or equal to *y*, the result of the expression is one (true). If *x* is less than *y*, the result is zero (false).

5.3.3 Indirection and Address-of Operators

Indirection (*)

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 to which the operand points. The result type is the type addressed by the pointer operand. If the pointer value is null, the result is unpredictable.

Address-of (&)

The address-of operator (&) takes the address of its operand. The operand can be any value that can appear as the left-hand value of an assignment operation. (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.

The address-of operator cannot be applied to a bitfield member of a structure, nor can it be applied to an identifier declared with the **register** storage class specifier.

Examples:

```
int *pa, x;
int  a[20];

1. pa=&a[5];

2. x=*pa;
```

In the first 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*.

The indirection operator (*) is used in the second example to access the **int** value at the address stored in *pa*. The value is assigned to the integer variable *x*.

5.3.4 Sizeof Operator

The **sizeof** operator determines the amount of storage associated with an identifier or a type. A **sizeof** expression has the form:

sizeof(*name*)

where *name* is either an identifier or a type name. The type name may not be **void.** The value of a **sizeof** expression is the amount of storage, in bytes, associated with the named identifier or type.

When the **sizeof** operator is applied to an array identifier, the result is the size of the entire array in bytes rather than the size of the pointer represented by the array identifier.

When the **sizeof** operator is applied to a structure or union type name, or to an identifier of structure or union type, the result is the actual size in bytes of the structure or union, which 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 members.

Example:

```
buffer=calloc(100, sizeof(int));
```

With the **sizeof** operator you can avoid specifying machine-dependent data sizes in your program. The above example uses the **sizeof** operator to pass the size of an **int**, which varies across machines, as an argument to a function named *calloc*. The value returned by the function is stored in *buffer*.

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 take integral and floating-point 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.

The conversions performed by the multiplicative operators make no provision for overflow or underflow conditions. Information is lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

Multiplication (*)

The multiplication operator (*) specifies that its two operands are to be multiplied.

Division (/)

The division operator (/) specifies that its first operand is to be divided by the second. When two integers are divided, the result, if not an integer, is truncated. If both operands are positive or unsigned, the result is truncated toward zero. The direction of truncation when either operand is negative may be either toward or away from zero, depending on the implementation. Division by zero gives unpredictable results.

Remainder (%)

The result of the remainder operator (%) is the remainder when the first operand is divided by the second.

Examples:

```
int i=10, j=3, n;
double x=2.0, y;

1. y=x*i;
```

2. `n=i/j;`

3. `n=i%j;`

In the first example, *x* is multiplied by *i* to give the value 20.0. The result has **double** type.

In the second example, 10 is divided by 3. The result is truncated toward zero, yielding the integer value 3.

In the third example, *n* is assigned the integer remainder 1 when 10 is divided by 3.

5.3.6 Additive Operators

The additive operators perform addition (+) and subtraction (–). The operands can be integral or floating-point values; some additive operations can also be performed on pointer values, as outlined under the discussion of each operator. The usual arithmetic conversions are performed on integral and floating-point operands. The type of the result is the type of the operands after conversion.

The conversions performed by the additive operators make no provision for overflow or underflow conditions. Information is lost if the result of an additive operation cannot be represented in the type of the operands after conversion.

Addition (+)

The addition operator (+) specifies addition of its two operands. The operands can have integral or floating-point types, as described above, 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 length of the value addressed by the pointer. 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 added to the pointer value, the result is a new pointer value expressing the address *i* positions from the original address. The new pointer value addresses the same type as the original pointer value.

Subtraction (–)

The subtraction operator (–) subtracts its second operand from the first. The operands can be integral or floating-point values, as described above. The subtraction operator also allows the subtraction of an integer from a pointer value and the subtraction of two pointer values.

When an integer value is subtracted from a pointer value, the same conversions take place as with addition of a pointer and integer. The subtraction operator converts the integer value with respect to the type addressed by the pointer value. The result is the memory address *i* positions before the original address, where *i* is the integer value and each position is the length of the type addressed by the pointer value. The new pointer points to the type addressed by the original pointer value.

Two pointer values can be subtracted if they point to the same type. The difference between the two pointers is converted to a signed integer value by dividing the difference by the length of the type 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 below.

Pointer Arithmetic

Additive operations involving a pointer and an integer generally give meaningful results only when the pointer operand addresses an array member and the integer value produces an offset within the bounds of the same array. The conversion of the integer value to an address offset assumes that only memory positions of the same size lie between the original address and the address plus offset.

This assumption is valid for array members. An array is by definition a series of values of the same type; its elements reside in contiguous memory locations. Storage of any types except array elements is not guaranteed to be completely filled. That is, blanks can occur between memory positions, even positions of the same type. Adding to or subtracting from addresses referring to any values but array elements gives unpredictable results.

Similarly, the conversion involved in the subtraction of two pointer values assumes that only values of the same type, with no blanks, lie between the two addresses given by the operands.

Additive operations between pointer and integer values on machines with segmented architecture must take the segment addressing conventions into account. In some cases these operations may not he valid. See your system documentation for more information.

Examples:

```
int i = 4, j;
float x[10];
float *px;
```

```
1. px=&x[4]+i;

2. j=&x[i]-&x[i-2];
```

In the first example, the integer operand *i* is added to the address of the fifth element of *x*. 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]".

In the second example, the address of the third element of *x* ("x[i-2]") is subtracted from the address of the fifth element of *x* ("x[i]"). The difference is divided by the length of a **float**. The result is the integer value −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. The usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.

For leftward shifts, the vacated right bits are filled with zeros. In a rightward shift, the method of filling left bits depends on the type (after conversion) of the first operand. If it is **unsigned**, vacated left bits will be filled with zeros. Otherwise, vacated left bits are filled with copies of the sign bit.

The result of a shift operation is undefined if the second operand is negative.

The conversions performed by the shift operators make no provision for overflow or underflow conditions. Information is 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 the above example, *x* is shifted left by 8 positions and *y* is shifted right 8 positions. The shifted values are added, giving 0xaa55, and assigned to *z*.

5.3.8 Relational Operators

The binary relational operators test their first operand against the second to determine if the relation specified by the operator holds true. The result of a relational expression is either one (if the tested relation holds) or zero (if it does not). 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-point, or pointer type. The types of the operands can be different. The usual arithmetic conversions are performed on integral and floating-point operands.

One or both operands of the equality (==) and inequality (!=) operators can have **enum** type. An **enum** value is converted in the same manner as an **int** value.

The operands of any relational operator can be two pointers to the same type. For the equality (==) and inequality (!=) operators the result of the comparison reflects whether the two pointers address the same memory location. The result of pointer comparisons involving the other operators (<, >, <=, >=) reflects 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. Comparisons between the addresses of different elements of the same array can be useful, however, 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 for equality (==) or inequality (!=) to the constant value zero (0). A pointer with a value of zero does not point to a memory location: it is called a “null” pointer. A pointer value is equal to zero only if it is explicitly given that value through assignment or initialization.

Examples:

```
int x=0,y=0;

1. x<y

2. x>y

3. x<=y

4. x>=y

5. x==y

6. x!=y
```

When *x* and *y* are equal, expressions 3, 4, and 5 have the value one and expressions 1, 2, and 6 have the value zero.

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 type, but their types can be different. The usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.

Bitwise AND (&)

The bitwise AND (&) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are ones, the corresponding bit of the result is set to one. Otherwise, the corresponding result bit is set to zero.

Bitwise Inclusive OR (|)

The bitwise inclusive OR (|) operator compares each bit of its first operand to the corresponding bit of the second operand. If either of the compared bits is a one, the corresponding bit of the result is set to one. Otherwise, both bits are zeros, and the corresponding result bit is set to zero.

Bitwise Exclusive OR (^)

The bitwise exclusive OR (^) operator compares each bit of its first operand to the corresponding bit of the second operand. If one of the compared bits is a zero and the other bit is a one, the corresponding bit of the result is set to one. Otherwise, the corresponding result bit is set to zero.

Examples

```
short  i=0xab00;
short  j=0xabcd;
short  n;

1. n=i&j;

2. n=i|j;

3. n=i^j;
```

The result assigned to *n* in the first example is the same as *i*, 0xab00. The bitwise inclusive OR in the second example results in the value 0xabcd, while the bitwise exclusive OR in the third example produces 0x00cd.

5.3.10 Logical Operators

The logical operators perform logical AND (&&) and OR (||) operations. The operands of the logical operators must have integral, floating-point, or pointer type. The types of the operands can be different.

The operands of logical AND and OR expressions are evaluated 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.

These operators do not perform the standard arithmetic conversions. Instead, they evaluate each operand in terms of its equivalence to zero. A pointer has a value of zero only if it is explicitly set to zero through assignment or initialization.

The result of a logical operation is either zero or one, as described below. The type of the result is **int**.

Logical AND (&&)

The logical AND operator (&&) produces the value one if both operands have nonzero values. If either operand is equal to zero, the result is zero. If the first operand of a logical AND operation has a value of zero, the second operand is not evaluated.

Logical OR (||)

The logical OR operator (||) performs an inclusive OR on its operands. It produces the value zero if both operands have zero values. If either operand has a nonzero value, the result is one. If the first operand of a logical OR operation has a nonzero value, the second operand is not evaluated.

Examples:

```
    int x, y;

    1. if(x < y && y < z)
         printf ("x is less than z\n");

    2. if (x==y || x==z)
         printf ("x is equal to either y or z\n");
```

In the first 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*, "y < z" is not evaluated and nothing is printed.

In the second example, a message is printed if *x* is equal to either *y* or *z*. If *x* is equal to *y*, "x==z" is not evaluated.

5.3.11 Sequential Evaluation Operator

The sequential evaluation operator (,) evaluates its two operands sequentially from left to right. The result of the operation has the value and type of the right operand. The types of the operands are unrestricted. No conversions are performed.

This operator (also called the "comma" operator) is typically used to evaluate two or more expressions in contexts that allow only one expression to appear.

Examples:

```
    1. for(i=j=1;i+j<20;i+=i, j--);

    2. f(x, y+2, z);
       f((x--, y+2), z);
```

In the first example, each operand of the **for** statement's third expression is evaluated independently. The left operand, "i += i," is evaluated first, then "j--" is evaluated.

As shown in the second example, the comma character is used in other contexts as a separator. In the first function call, three arguments, separated by commas, are passed to the called function: *x*, "y + 2", and *z*. The use of the comma character as a separator must not be confused with its use as an operator; the two functions are completely different.

In the second function call, parentheses force the compiler to interpret the first comma as the sequential evaluation operator. This function call passes two arguments to *f*. 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*.

5.3.12 Conditional Operator

C has one ternary operator, the conditional operator (? :). Its form is:

operand1 ? operand2 : operand3

operand1 is evaluated in terms of its equivalence to zero. It must have integral, floating-point, or pointer type. If *operand1* has a nonzero value, *operand2* is evaluated and the result of the expression is the value of *operand2*. If *operand1* evaluates to zero, *operand3* is evaluated, and the result of the expression is the value of *operand3*. Notice that either *operand2* or *operand3* is evaluated, but not both.

The type of the result depends on the types of the second and third operands, as follows:

1. If both the second and third operands have integral or floating-point type (their types can be different), the usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.
2. Both the second and third operands can have the same structure, union, or pointer type. The type of the result is the same structure, union, or pointer type.
3. One of the second or third operands can be a pointer and the other a constant-expression with the value zero. The type of the result is the pointer type.

Example:

```
j = (i < 0) ? (-i) : (i);
```

The above example assigns the absolute value of *i* to *j*. If *i* is less than zero, −*i* is assigned to *j*. If *i* is greater than or equal to zero, *i* is assigned to *j*.

5.4 Assignment Operators

C's assignment operators can both transform and assign values in a single operation. Using a compound assignment operator to replace two separate operations can reduce code size and improve program efficiency. The assignment operators are listed and described below:

++	Unary increment operator
--	Unary decrement operator
=	Simple assignment operator
*=	Multiplication assignment operator
/=	Division assignment operator
%=	Remainder assignment operator
+=	Addition assignment operator
-=	Subtraction assignment operator
<<=	Left shift assignment operator
>>=	Right shift assignment operator
&=	Bitwise AND assignment operator
|=	Bitwise inclusive OR assignment operator
^=	Bitwise exclusive OR assignment operator

In assignment, the type of the right-hand value is converted to the type of the left-hand value. The specific path of the conversion depends on the two types and is outlined in detail in Section 5.7.

5.4.1 Lvalue Expressions

An assignment operation specifies that the value of the right-hand operand is to be assigned to the storage location named by the left-hand operand. Thus, the left-hand operand of an assignment operation (or the single operand of a unary assignment expression) must be an expression referring to a memory location. Expressions that refer to memory locations are called "lvalue" expressions. A variable name is such an expression: the name of the variable denotes a storage location, while the value of the variable is the value residing at that location.

The C expressions that may be lvalue expressions are:

- identifiers of character, integer, floating-point, pointer, enumeration, structure, or union type
- subscript ([]) expressions, except when a subscript expression evaluates to a pointer to an array
- member selection expressions (-> and .), if the selected member is one of the above expressions

- unary indirection (*) expressions, except when such expressions refer to arrays
- type casts to pointer types
- an lvalue expression in parentheses

5.4.2 Unary Increment and Decrement

The unary assignment operators (++ and --) increase and decrease their operand, respectively. The operand must have integral, floating-point, or pointer type, and must be an lvalue expression.

Operands of integral or floating-point type are increased or decreased by the integer value 1. The type of the result is the type of the operand. An operand of pointer type is increased or decreased by the size of the object it addresses. An increased pointer points to the next object; a decreased pointer points to the previous object.

An increment (++) or decrement (--) operator can appear either before or after its operand. When the operator prefixes its operand, the result of the expression is the increased or decreased value of the operand. When the operator postfixes its operand, the immediate result of the expression is the value of the operand *before* it is increased or decreased. After that result is noted in context, the operand is increased or decreased.

Examples:

```
1. if (pos++ > 0)
       *ptt = *qtt;

2. if (line[--i] != '\n')
       return;
```

In the first example, the variable *pos* is compared to zero, then increased by one.

In the second example, the variable *i* is decreased before it is used as a subscript to *line*.

5.4.3 Simple Assignment

The simple assignment operator (=) performs assignment. The right operand is assigned to the left operand; the conversion rules for assignment (discussed in Section 5.7.1) apply.

Example:

```
doublex;
int y;

x=y;
```

The value of *y* is converted to **double** type and assigned to *x*.

5.4.4 Compound Assignment

The compound assignment operators consist of the simple assignment operator combined with another binary operator. Compound assignment operators perform the operation specified by the additional operator, then assign the result to the left operand. A compound assignment expression such as:

expression1 += *expression2*

can be understood as:

expression1 = *expression1* + *expression2*

However, the compound assignment expression is not equivalent to the expanded version because the compound assignment expression evaluates *expression1* only once, while in the expanded version *expression1* is evaluated twice: in the addition operation and in the assignment operation.

Each compound assignment operator performs the conversions that the corresponding binary operator performs, and restricts the types of its operands accordingly. The result of a compound assignment operation has the value and type of the left operand.

Example:

```
#defineMASK 0xffff

n |= MASK;
```

In this example, a bitwise inclusive OR 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, discussed in Section 8.2.1 of Chapter 8, "Preprocessor Directives"."

5.5 Precedence and Order of Evaluation

The precedence and associativity of C operators affect the grouping and evaluation of operands in an expression. An operator's precedence is meaningful only in the presence of other operators having higher or lower precedence. Expressions involving higher precedence operators are evaluated first.

Table 5.1 summarizes the precedence and associativity of C operators. The operators are listed in order of precedence from the highest to the lowest. Where several operators appear together in a line or large brace, they have equal precedence and are evaluated according to their associativity, that is, either left to right or right to left.

Table 5.1

Precedence and Associativity of C Operators

Operator[a] Symbol	Type of Operation	Associativity
() [] . –>	Expression	Left to right
– ~ ! * & ++ –– sizeof *casts*	Unary[b]	Right to left
* / %	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 assignment[c]	Right to left
,	Sequential evaluation	Left to right

[a] Operators are listed in descending order of precedence. Where 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 highest precedence and associate 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 right to left or left to right. The result of expressions involving multiple occurrences of multiplication (*), addition (+), or binary bitwise (&, |, ^) operators at the same level is indifferent to the direction of evaluation. 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 operator (,) and the logical AND (&&) and OR (||) operators guarantee a particular order of evaluation for the operands. The sequential evaluation operator (,) is guaranteed to evaluate its operands from left to right.

The logical operators also guarantee to evaluate their operands left to right. However, the logical operators evaluate the minimum number of operands necessary 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 increased when *x* is false (zero).

The examples below show the default grouping for several expressions:

Examples:

Expression	**Default Grouping**
1. a & b || c	(a & b) || c
2. a = b || c	a = (b || c)
3. q && r || s--	(q && r) || s--

In the first example, the bitwise AND operator (&) has higher precedence than the logical OR operator (||), so "a & b" forms the first operand of the logical OR operation.

In the second example, the logical OR operator (||) has higher precedence than the simple assignment operator (=), so "b || c" is grouped as the right-hand operand in the assignment. Notice that the value assigned to *a* is either zero or one.

The third example shows a correctly formed expression that may produce an unexpected result. The logical AND operator (&&) has higher precedence than the logical OR operator (||), 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 decreased. To correct this problem, "s--" should appear as the first operand of the expression or should be decreased in a separate operation.

The following example shows an illegal expression that produces a program error:

Example:

IllegalExpression	**Default Grouping**
p == 0 ? p += 1 : p += 2	(p == 0 ? p += 1 : p) += 2

In this example, 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.

To prevent errors of this kind, and to produce more readable code, the use of parentheses is recommended. The above example can be corrected and clarified through the use of parentheses, as shown here:

```
    (p==0) ? (p+=1) : (p+=2)
```

5.6 Side Effects

"Side effects" are changes in the state of the machine that take place as a result of evaluating an expression. Side effects occur whenever the value of a variable is changed. Any assignment operation has side effects, and any call to a function that contains assignment operations has side effects.

The order of evaluation of side effects is implementation-dependent, except where the compiler 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" may be evaluated before "i = j + 2", or vice versa, with different results in each case.

Unary increment and decrement operations involve assignment and can cause side effects, as shown in the following example:

```
    d = 0;
    a = b++ = c++ = d++;
```

The value of *a* is unpredictable. The initial value of *d* (zero) could be assigned to *c*, then to *b*, and then to *a* before any of the variables are increased. In this case *a* would be equal to zero.

A second method of evaluating this expression begins by evaluating the operand "c++ = d++". The initial value of *d* (zero) is assigned to *c*, and then both *d* and *c* are increased. Next, the increased value of *c* (one) is assigned to *b* and *b* is increased. Finally, the increased value of *b* is assigned to *a*. In this case, the final value of *a* is two.

Since the C language does not define the order of evaluation of side effects, both of these evaluation methods are correct and either can be implemented. Statements that depend on a particular order of evaluation for side effcts produce nonportable and unclear code.

5.7 Type Conversions

Type conversions take place when a value is assigned to a variable of a different type, when a value is explicitly cast to another type, when an operator converts the type of its operand or operands before performing an operation, and when a value is passed as an argument to a function. The rules governing each kind of conversion are outlined below.

5.7.1 Assignment Conversions

In assignment operations, the type of the value being assigned is converted to the type of the variable receiving the assignment. C allows conversions by assignment between integral and floating-point types, even when the conversion entails loss of information. The methods of carrying out the conversions depend upon the type, as follows:

Conversions from Signed Integral Types

A signed integer is converted to a shorter signed integer by truncating the high-order bits and is converted to a longer signed integer by sign-extension. Conversion of signed integers to floating-point values takes place without loss of information, except that some precision can be lost when a **long** value is converted to a **float**. To convert a signed integer 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.

Conversions from signed integral types are summarized in Table 5.2.

Table 5.2

Conversions from Signed Integral Types

From	To	Method
char	**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**.
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**.
long	**char**	Preserve low-order byte.
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 a **float**; if the **long** cannot be represented exactly, some loss of precision occurs.
long	**double**	Represent as a **double**; if the **long** cannot be represented exactly as a **double**, some loss of precision occurs.

Note: The **int** type is equivalent either to the **short** type or to the **long** type, depending on the implementation. Conversion of an **int** value proceeds 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. An unsigned integer is converted to a longer unsigned or signed integer by zero-extending. Unsigned values are converted to floating-point values by first converting to a signed integer of the same size, then converting that signed value to a floating-point value.

When an unsigned integer is converted to a signed integer of the same size, no change in the bit pattern occurs. However, the value represented changes if the sign bit is set.

Conversions from unsigned integral types are summarized in Table 5.3.

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 short	char	Preserve low-order byte.
unsigned short	short	Preserve bit pattern; high-order bit becomes sign bit.
unsigned short	long	Zero-extend.
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 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**.

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 as for an **unsigned short** or an **unsigned long**, whichever is appropriate.

Conversions from Floating-Point Types

A **float** value converted to a **double** undergoes no change in value. A **double** converted to a **float** is represented exactly, if possible. If the value is too large to fit into a **float**, precision is lost.

A floating-point value is converted to an integer value by converting to a **long**. Conversions to other integer types take place as for a **long**. The decimal portion of the floating-point 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.

Conversions from floating-point types are summarized in Table 5.4:

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 a 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.
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 a long, result is undefined.
double	unsigned short	Convert to long; convert long to unsigned short.
double	unsigned long	Convert to long; convert long to unsigned long.
double	float	Represent as a float; if the double value cannot be represented exactly as a float, loss of precision occurs; if the value is too large to be represented in a float, the result is undefined.

Conversions from Other Types

An **enum** value is an **int** value, by definition of the **enum** type. Conversions to and from an **enum** value proceed as 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.

A pointer value behaves like an unsigned integer value in conversions, with the size of the pointer determined by the implementation. Conversions to and from a pointer type proceed as for an unsigned integer of the appropriate size, except that pointers cannot be converted to floating-point types.

A pointer to one type of value can be converted to a pointer to a different type. The result may be undefined, however, because of the alignment requirements and sizes of different types in storage. In some implementations the **near** and **far** keywords modify pointer sizes. Conversions between **near** and **far** pointers may produce meaningless addresses.

The **void** type has no value, by definition. Therefore, it cannot be converted to any other type, nor can any value be converted to **void** by assignment. However, a value can be explicitly cast to **void**, as discussed in Section 5.7.2.

5.7.2 Type-Cast Conversions

Explicit type conversions can be made by means of a type cast. A type cast has the form:

(type-name)operand

where *type-name* specifies a particular type and *operand* is a value to be converted to the specified type. (Type names are discussed in Section 4.9 of Chapter 4, "Declarations".")

The conversion of *operand* takes place as though it had been assigned to a variable of the named type. The conversion rules for assignments (outlined in Section 5.7.1) apply to type casts as well. The type name **void** can be used in a cast operation, but the resulting expression cannot be assigned to any item.

5.7.3 Operator Conversions

The conversions performed by C operators depend on the operator and on the type of the operand and operands. Many operators perform the "usual arithmetic conversions," which are outlined in Section 5.3.1.

C permits some arithmetic with pointers. In pointer arithmetic, integer values are converted to express memory positions. See the discussions of additive operators (Section 5.3.6) and subscript expressions (Section 5.2.5) for details.

5.7.4 Function-Call Conversions

The type of conversion performed on the arguments in a function call depends on whether a forward declaration with declared argument types is present for the called function.

If a forward declaration is present, and it includes declared argument types, the compiler performs type-checking. The type-checking process is outlined in detail in Section 7.4.1 of Chapter 7, "Functions"."

If no forward declaration is present, or if the forward declaration omits the argument type list, the only conversions performed on the arguments in the function call are the usual arithmetic conversions. 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**.

Chapter 6 Statements

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
- compound statement
- **continue** statement
- **do** statement
- expression statement
- **for** statement
- **goto** statement
- **if** statement
- null statement
- **return** statement
- **switch** statement
- **while** statement

C statements consist of keywords, expressions, and other statements. The keywords that appear in C statements are:

break	**do**	**if**
case	**else**	**return**
continue	**for**	**switch**
default	**goto**	**while**

The expressions in C statements are the expressions discussed in Chapter 5, "Expressions and Assignments"." 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 (i.e., a single statement), is 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 be prefixed with an identifying label consisting of a name and a colon. Statement labels are recognized only by the **goto** statement and are therefore discussed with the **goto** statement.

When a C program is executed, its effect is that of the execution of the statements in order of their appearance in the program, except where a statement explicitly transfers control to another location.

6.2 Break Statement

Syntax

break;

Execution

The **break** statement terminates the execution of the smallest enclosing **do, break, switch,** or **while** statement in which it appears. Control passes to the statement following the terminated statement. A **break** statement appearing outside any **do, for, switch,** or **while** statement causes an error.

Within nested statements, the **break** statement terminates only the **do, for, switch,** or **while** statement immediately enclosing it. To transfer control out of the nested structure altogether, a **return** or **goto** statement can be used.

Example:

```
for (i=0;i < LENGTH - 1; i++) {
    for(j=0;j < WIDTH - 1;j++) {
        if (lines[i][j]=='\0') {
            lengths[i]=j;
            break;
        }
    }
}
```

The above 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 stored in *lengths*[*i*]. Control then returns to the outer **for** loop. The variable *i* is increased and the process is repeated until *i* is greater than or equal to *LENGTH-1*.

6.3 Compound Statement

Syntax

```
{
[declaration]
  .
  .
  .
statement
[statement]
  .
  .
  .
}
```

Execution

All statements within a compound statement are executed in the order of their appearance. The single exception to this is where one of the statements causes a logical branch.

Example:

```
if (i > 0) {
    line[i] = x;
    x++;
    i--;
}
```

A compound statement typically appears as the body of another statement such as the **if** statement. In the above example, if *i* is greater than zero, all of the statements in the compound statement are executed in order.

Labeled Statements

Like other C statements, any of the statements in a compound statement may carry a label. Transfer into the compound statement by means of a **goto** is therefore possible. However, transferring into a compound statement is dangerous when the compound statement includes declarations that initialize variables. Declarations in a compound statement precede the executable statements, so transferring directly to an executable statement within the compound statement bypasses the initializations. The results are unpredictable.

6.4 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. Within a **do** or a **while** statement, the next iteration begins with the reevaluation of the **do** or **while** statement's expression. Within a **for** statement, the next iteration starts with the evaluation of the **for** statement's *loop-expression*. It proceeds with the evaluation of the conditional expression and subsequent termination or reiteration of the statement body.

Example:

```
while (i-- > 0) {
    x=f(i);
    if(x==1)
        continue;
    y=x*x;
}
```

The statement body is executed if *i* is greater than zero. 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".

6.5 Do Statement

Syntax

do
statement
while (*expression*);

Execution

The body of a **do** statement is executed one or more times until *expression* becomes false. First, the statement body is executed. Then *expression* is evaluated. If *expression* is false (zero), the **do** statement terminates and control passes to the next statement in the program. If *expression* is true (nonzero), the statement body is executed again, and *expression* is tested again. The statement body is executed repeatedly until *expression* becomes false.

The **do** statement may also terminate with the execution of a **break, goto,** or **return** statement within the statement body.

Example:

```
do{
    y=f(x);
    x--;
}while (x> 0);
```

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 zero, the statement body is executed again and "x > 0" is reevaluated. The statement body is executed repeatedly so long as *x* remains greater than zero. Execution of the **do** statement terminates when *x* becomes zero or negative.

6.6 Expression Statement

Syntax

expression;

Execution

The expression is evaluated, according to the rules outlined in Chapter 5, "Expressions and Assignments"."

Examples:

1. x=(y+3);
2. x++;
3. f(x);

In C, assignments are expressions; the value of the expression is the value being assigned (sometimes called the "right-hand value"). In the first example, *x* is assigned the value of "y + 3". In the second example, *x* is increased by 1.

The third example shows a function call expression. The value of the expression is the value, if any, returned by the function. If a function returns a value, the expression statement usually incorporates an assignment to store the returned value when the function is called. If the return value is not assigned, as in the example, the function call is executed but the return value, if any, is not used.

6.7 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. The *init-expression* and *loop-expression* are optional expressions that can be used to initialize and modify values during the **for** statement's execution.

The first step in the execution of the **for** statement is the evaluation of *init-expression*, if present. Next, *cond-expression* is evaluated, with three possible results:

1. If the conditional expression is true (nonzero), the statement body is executed; then *loop-expression*, if present, is evaluated; then the process begins again with the evaluation of *cond-expression*.
2. If the conditional expression is omitted, the conditional expression is considered true; execution proceeds exactly as described above. A **for** statement lacking *cond-expression* terminates only upon the execution of a **break, goto,** or **return** statement within the statement body.
3. If the conditional expression is false, execution of the **for** statement terminates and control passes to the next statement in the program.

A **for** statement may also terminate with the execution of a **break, return,** or **goto** statement within the statement body.

Example:

```
for (i = space = tab = 0; i < MAX; i++) {
    if (line[i] == '\0x20')
        space++;
    if (line[i] == '\t') {
        tab++;
        line[i] = '\0x20';
    }
}
```

The above example counts space (\0x20) 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 zero. Then *i* is compared to 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 increased and tested against *MAX*. The statement body is executed repeatedly as long as *i* is less than *MAX*.

6.8 Goto and Labeled Statements

Syntax

```
goto name;
  .
  .
  .
name: statement
```

Execution

The **goto** statement transfers control directly to the statement specified by *name*. The labeled statement is executed immediately after the **goto** statement is executed. An error results if no statement with the given label resides in the same function or if an identical label appears before more than one statement in the same function.

A statement label is meaningful only to a **goto** statement. When a labeled statement is encountered in any other context, the statement is executed without regard to the label.

Example:

```
if (errorcode > 0)
    goto exit;
      .
      .
      .
exit:
    return (errorcode);
```

In the example, a **goto** statement transfers control to the point labeled *exit* when an error occurs.

Forming Labels

A label name is simply an identifier, formed by following the same rules that govern the construction of identifiers (see Section 2.4 of Chapter 2, "Elements of C"). Each statement label must be distinct from other statement labels and identifiers in the same function.

6.9 If Statement

Syntax

> **if** (*expression*)
> *statement1*
> [**else**
> *statement2*]

Execution

The body of an **if** statement is executed selectively, depending on the value of *expression*. First, *expression* is evaluated. If *expression* is true (nonzero), the statement immediately following it is executed. If *expression* is false, the statement following the **else** keyword is executed. If *expression* is false and the **else** clause is omitted, the statement following *expression* is ignored. Control then passes from the **if** statement to the next statement in the program.

Example:

```
if(i > 0)
    y = x/i;
else{
    x = i;
    y = f(x);
}
```

In the example, the statement "y = x/i;" is executed if *i* is greater than zero. If *i* is less than or equal to zero, *i* is assigned to *x* and "f(x)" is assigned to *y*. Notice that the statement forming the **if** clause ends with a semicolon.

Nesting

C does not offer an "else if" statement, but the same effect is achieved by nesting **if** statements. An **if** statement may be nested in 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. In the absence of braces, the compiler resolves ambiguities by pairing each **else** with the most recent **if** lacking an **else**.

Examples:

```
1. if (i > 0)        /* Without braces */
    if(j > i)
        x = j;
    else
        x = i;


2. if (i > 0){      /* With braces */
    if (j > i)
        x = j;
}
else
    x = i;
```

In the first example, the **else** is associated with the inner **if** statement. If *i* is less than or equal to zero, no value is assigned to *x*.

In the second version, 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*.

6.10 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.

Example:

```
    for (i=0; i < 10; line[i++] = 0)
        ;
```

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. In the above example, the third expression of the **for** statement initializes the first ten elements of *line* to zero. The statement body is a null statement, since no further statements are necessary.

Labeling a Null Statement

The null statement, like any other C statement, may be prefixed by an identifying label. To label an item that is not a statement, such as the closing brace of a compound statement, you can insert and label a null statement immediately before the item to get the same effect.

6.11 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 just after 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.

Example:

```
    main()
    {
         .
         .
         .
       y=sq(x);
       draw(x, y);
         .
         .
         .
    }

    sq(x)
    int x;
    {
       return (x * x);
    }

    void draw(x,y)
    int x, y;
    {
         .
         .
         .
       return;
    }
```

The *main*() function calls two functions, *sq*() and *draw.*() The *sq*() function returns the value of "x * x" to *main.*() 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 an error.

By convention, parentheses enclose the *expression* of the **return** statement, as shown above. The language does not require the parentheses.

Omitting the Return Statement

If no **return** statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function. The return value of the called function is undefined. If a return value is not required, the function should be declared to have **void** return type.

6.12 Switch Statement

Syntax

```
switch (expression) {
[declaration]
  .
  .
  .
[case constant-expression :]
  .
  .
  .
[statement]
  .
  .
  .
[default:
statement]
[case constant-expression :]
  .
  .
  .
[statement]
  .
  .
  .
}
```

Execution

The **switch** statement transfers control to a statement within its body. The statement receiving control is the statement whose **case** *constant-expression* matches the value of the *expression* in parentheses. Execution of the statement body begins at the selected statement and proceeds through the end of the body or until a statement transfers control out of the body.

The **default** statement is executed if no **case** *constant-expression* is equal to the value of the **switch** *expression*. If the **default** statement is omitted, and no **case** match is found, none of the statements in the **switch** body are executed.

The **switch** expression must be an integral or **enum** value. If the *expression* is shorter than an **int**, it is widened to an **int** value. Each **case** *constant-expression* is then cast to the type of the **switch** expression. The value of each **case** *constant-expression* must be unique within the statement body.

The **case** and **default** labels of the **switch** statement body are significant only in the initial test that determines the starting point for execution of the statement body. All statements appearing 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.

Declarations may appear at the head of the compound statement forming the **switch** body, but initializations included in the declarations are not performed. The effect of the **switch** statement is to transfer control directly to an executable statement within the body, bypassing the lines that contain initializations.

Examples:

```
1. switch (c){
     case 'A':
         capa++;
     case 'a':
         lettera++;
     default:
         total++;
   }

2. switch (i){
     case -1:
         n++;
         break;
     case 0:
         z++;
         break;
     case 1:
         p++;
         break;
   }
```

In the first example, all three statements of the **switch** body are executed if *c* is equal to *'A'*. Execution control is transferred to the first statement ("capa++;") and continues in order through the rest of the body. If *c* is equal to *'a'*, *lettera* and *total* are increased. Only *total* is increased if *c* is not equal to *'A'* or *'a'*.

In the second example, a **break** statement follows each statement of the **switch** body. The **break** statement forces an exit from the **switch** after one statement in the body is executed. If *i* is equal to -1, only *n* is increased. The **break** following the statement "n++;" causes execution control to pass out of the **switch** body, bypassing the remaining statements. Similarly, if *i* is equal to 0, only *z* is increased; if *i* is equal to 1, only *p* is increased. The final **break** statement is not strictly necessary, since control will pass out of the body at the end of thecompound statement, but it is included for consistency.

Multiple Labels

A statement may carry multiple **case** labels, as the following sample shows:

```
    case'a':
    case'b':
    case'c':
    case'd':
    case'e':
    case'f': hexcvt(c);
```

Although any statement within the body of the **switch** statement may be labeled, no statement is required to carry a label. Statements without labels may be freely intermingled with labeled statements. Keep in mind, however, that once the **switch** statement passes control to a statement within the body, all succeeding statements in the block are executed, regardless of their labels.

6.13 While Statement

Syntax

while (*expression*)
statement

Execution

The body of a **while** statement is executed zero or more times until *expression* becomes false. First, *expression* is evaluated. If the *expression* is initially false (zero), the body of the **while** statement is never executed, and control passes from the **while** statement to the next statement in the program. If *expression* is true (nonzero), the body of the statement is executed. Following each execution of the statement body, *expression* is reevaluated. The body is executed repeatedly as long as *expression* remains true.

The **while** statement may also terminate with the execution of a **break**, **goto**, or **return** within the statement body.

Example:

```
while (i >= 0){
    string1[i] = string2[i];
    i--;
}
```

The above example copies characters from *string2* to *string1*. If *i* is greater than or equal to zero, *string2[i]* is assigned to *string1[i]* and *i* is decreased. When *i* reaches or falls below 0, execution of the **while** statement terminates.

Chapter 7

Functions

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 main function and may have other functions. The sections of this chapter describe how to define, declare, and call C functions.

A function *definition* specifies the name of the function, its formal parameters, and the declarations and statements that define its action. The function definition can also give the return type of the function and its storage class.

A function *declaration* establishes the name, return type, and storage class of a function whose explicit definition is given at another point in the program. The number and types of arguments to the function can also be specified in the function declaration. This allows the compiler to compare the types of the actual arguments and the formal parameters of a function. Function declarations are optional for functions whose return type is **int**. To ensure correct behavior, functions with other return types must be declared before they are called.

A function *call* passes execution control from the calling function to the called function. The actual arguments, if any, are passed by value to the called function. Execution of a **return** statement in the called function returns control and possibly a value to the calling function.

7.2 Function Definitions

A function definition specifies the name, formal parameters, and body of a function. It may also define the function's return type and storage class. A function definition has the following form:

> [*sc-specifier*][*type-specifier*] *declarator* ([*parameter-list*])
> [*parameter-declarations*]
> *function-body*

The *sc-specifier* gives the function's storage class, which must be either **static** or **extern**. The *type-specifier* and *declarator* together specify the function's return type and name. The *parameter-list* is a list (possibly empty) of formal parameters to be used by the function. The *parameter-declarations* establish the types of the formal parameters. The *function-body* is a compound statement containing local variable declarations and statements. The following sections describe the parts of the function definition in detail.

7.2.1 Storage Class

The storage class specifier in a function definition gives the function either **static** or **extern** storage class. 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 constitute the program.

The storage class specifier is required in a function definition in only one case: when the function is declared elsewhere in the same source file with the **static** storage class specifier.

The **static** storage class specifier can also be used when defining a function previously declared in the same source file without a storage class specifier. Normally, a function declared without a storage class specifier defaults to the **extern** class. However, if the function definition explicitly specifies the **static** class, the function is given **static** class instead.

When the storage class specifier is omitted from a function definition, the storage class defaults to **extern**. The **extern** storage class specifier can be explicitly specified in the function definition, but it is not required.

7.2.2 Return Type

The return type of a function defines the size and type of value returned by the function. The type declaration has the form:

[*type-specifier*] *declarator*

where *type-specifier*, together with the *declarator*, define the function's return type and name. If no *type-specifier* is given, the return type **int** is assumed.

The *type-specifier* can specify any fundamental, structure, or union type. The *declarator* consists of the function identifier, possibly modified to declare a pointer type. 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. Functions with **int** return type do not have to be declared before they are called. Functions with other return types cannot be called before they are either defined or declared.

A function's return value 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 of call. If no **return** statement is executed, or if the executed **return** statement does not contain an expression, the return value of the function is undefined. If the calling function expects a return value, the behavior of your program is also undefined.

Examples:

```
1. /* return type is int*/
  static add (x, y)
  int x, y;
  {
    return (x+y);
  }

2. typedef struct {
    char name[20];
    int id;
    long class;
  } STUDENT;

    /*return type is STUDENT */
    STUDENT sortstu (a, b)
    STUDENT a, b;
  {
    return ( (a.id < b.id) ? a : b );
  }
```

```
3. /* 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);
  }
```

In the first example, the return type of *add* is int by default. The function has static storage class, which means it can be called only by functions in the same source file.

The second 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.

The third example defines 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**.

7.2.3 Formal Parameters

Formal parameters are variables that receive values passed to a function by a function call. The formal parameters are declared in a parameter list at the beginning of the function declaration. The parameter list defines the names of the parameters and the order in which they take on values in the function call.

The parameter list has the form:

> ([*identifier*[, *identifier*]]...)

where each *identifier* names a parameter. The parentheses are required.

Parameter declarations define the type and size of values stored in the formal parameters. These declarations have the same form as other variable declarations (see Section 4.4 of Chapter 4, "Declarations"). A formal parameter can have any fundamental, structure, union, pointer, or array type.

A parameter can only have **auto** or **register** storage class. If no storage class is given, **auto** storage is assumed. If a formal parameter is named in the parameter list but is not declared, the parameter is assumed to have **int** type. Formal parameters can be declared 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 used for variable declarations within the function body.

The type of the formal parameter should correspond to the type of the actual argument and to the type of the corresponding argument in the argument type list for the function, if present. If the function has a variable number of arguments, the user is responsible for determining the number of arguments passed and for retrieving additional arguments from the stack within the body of the function.

The compiler performs the usual arithmetic conversions independently on each formal parameter and on each actual argument, if necessary. After 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 an **int**.

The converted type of each formal parameter determines how the arguments placed on the stack by the function call are interpreted. A type mismatch between an actual 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 stored in the **long** formal parameter. This error creates problems not only with the **long** formal parameter, but with any formal parameters that follow it. Errors of this kind can be detected through diligent use of argument type lists in function declarations.

Example:

```
struct student {
    char name[20];
    int id;
    long class;
    struct student *nextstu;
}student;

main()
{
    int match ( struct student *, char * );
        .
        .
        .
    if (match (student.nextstu,
         student.name) > 0) {
        .
        .
        .
    }
}

match ( r, n )
struct student *r;
char *n;
{
    int i = 0;

    while ( r->name[i]==n[i] )
        if ( r->name[i++]=='\0' )
            return (r->id);
    return(0);
}
```

The example contains a structure type declaration, a forward declaration of the function *match*(), a call to *match*, and the definition of the *match*() function. Notice 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 a pointer to the *student* structure type and the second a pointer to a **char** type.

The two formal parameters of the *match*() function are *r* and *n*. The parameter *r* is declared as a pointer to the *student* structure type. The parameter *n* is declared as a pointer to a **char** type.

The function is called with two arguments, both members of the *student* structure. Because there is a forward declaration of *match*, the compiler performs type-checking between the actual arguments and the argument type list and between the actual arguments and the formal parameters. 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 keep track of the current position in the array. The function returns the *id* structure member if the *name* member matches the array *n*. Otherwise, it returns zero.

7.2.4 Function Body

The function body is simply a compound statement. The compound statement contains the statements that define the function's action and can also contain declarations of variables used by these statements. See Section 6.3 of Chapter 6, "Statements", for a discussion of compound statements.

All variables declared in the function body have **auto** storage type unless otherwise specified. When the function is called, storage space for the local variables is created and local initializations are performed. Execution control passes to the first statement in the compound statement and continues sequentially until a **return** statement or the end of the function body is encountered. Control then passes back to the point of call.

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 the optional expression.

7.3 Function Declarations

A function declaration defines the name, return type, and storage class of a given function, and may establish the type of some or all of the function's arguments. See Chapter 4, "Declarations", for a detailed description of the syntax of function declarations.

Functions can be declared implicitly or with forward declarations. The return type of a function declared either implicitly or with a forward declaration must agree with the return type specified in the function definition.

An implicit declaration occurs whenever a function is called without being previously defined or declared. The C compiler implicitly declares the function to have **int** return type. By default, the function is declared to have **extern** storage class. The function definition can redefine the storage class to **static**, provided the function definition is given later in the same source file.

A forward declaration establishes the attributes of a function, allowing the declared function to be called before it is defined or to be called from another source file. If the storage class specifier **static** is given in a forward declaration, the function has **static** class. The function definition must also specify the **static** class. If the storage class specifier is **extern** or is omitted, the function has **extern** class. However, the function definition can redefine the storage class as **static**, provided the function definition appears below the declaration in the same source file.

Forward declarations have several important uses. They establish the return type for functions that return any type of value but **int**. (Functions that return **int** values can also have forward declarations, but do not require them.) Functions with non-**int** return types cannot be called before they are either declared or defined; the compiler assumes that the called function has **int** return type.

Forward declarations can be used to establish the types of arguments expected in a function call. The optional argument type list of a forward declaration gives the type and number of arguments expected. (The number of arguments can be variable.) The argument type list is a list of type names corresponding to the expression list in the function call.

If no argument type list is supplied, no type-checking is performed. Type mismatches between actual arguments and formal parameters are silently accepted. Type-checking is discussed further in Section 7.4.1.

Forward declarations are also used to declare pointers to functions before the functions are defined.

Example:

```
    main()
    {
       int a=0, b=1;
       float x=2.0, y=3.0;
       double realadd(double, double);

       a=intadd (a, b);
       x=realadd(x, y);
    }

    intadd(a, b)
    int a, b;
    {
       return (a+ b);
    }

    double realadd(x, y)
    double x, y;
    {
       return (x+y);
    }
```

In the example, the function *intadd*() is implicitly declared to return an **int** value, since it is called before it is defined. The compiler does not check the types of the arguments in the call because no argument type list is available.

The function *realadd*() returns a **double** value instead of an **int**. The forward declaration of *realadd* in the *main*() function allows the *realadd*() function to be called before it is defined. Notice that the definition of *realadd*() matches the forward declaration by specifying the **double** return type.

The forward declaration of *realadd*() also establishes the type of its two arguments. The actual arguments match the types given in the forward declaration and also match the types of the formal parameters.

7.4 Function Calls

A function call is an expression that passes control and zero or more actual arguments to a function. A function call has the form:

expression (*expression- list*)

where *expression* evaluates to a function address and *expression- list* is a list of expressions whose values, the actual arguments, are passed to the function. The *expression-list* can be empty.

When the function call is executed, the expressions in the function expression list are copied, converted as necessary, and then passed to 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 end of the list. Since the called function works with copies of the actual arguments, any changes it makes to the arguments are not reflected in the original values from which the copies were made.

Execution control then passes to the first statement in the function. 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. The return value is undefined.

The expressions in the function call's expression list can be evaluated in any order, so expressions with side effects have unpredictable results. The only guarantee the compiler makes is that all side effects in the expression list are evaluated before control passes to the called function.

The only requirement in calling a function is for the expression before the parentheses to evaluate to a function address. This means that a function can be called through any function pointer expression. It may be helpful to remember that a function is called in the same manner it is declared. For instance, when declaring a function, the name of the function is given, followed by an argument type list in parentheses. To call the function, only the name of the function is required, followed by an expression list in parentheses. The indirection operator (*) is not required to call the function; the name of the function evaluates to the function address, which is used to call the function.

The same principle applies when calling a function through a pointer. For example, suppose a function pointer is declared as follows:

```
int (*fpointer)(int, int);
```

The identifier *fpointer* is declared to point to a function taking two **int** arguments and returning an **int** value. A function call through *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.

Examples:

```
1. double *realcomp(double, double);
   double a, b, *rp;
          .
          .
          .
   rp = realcomp(a, b);

2. main()
   {
     longlift(int), step(int), drop(int);
     void work (int, long (*)(int));
     int select, count;
          .
          .
          .
     select = 1;
     switch ( select ) {
         case 1: work(count, lift);
             break;

         case2: work(count, step);
             break;

         case3: work(count, drop);

         default:
             break;
     }
   }
```

```
void work (n, func)
int n;
long (*func)(int);
{
 int i;
 long j;

 for (i=j=0; i < n; i++)
     j += (*func)(i);
}
```

In the first example, the *realcomp*() function is called in the statement rp = realcomp(a, b);. Two **double** arguments are passed to the *realcomp*() function. The return value, a pointer to a **double**, is assigned to *rp*.

In the second example, the function call:

```
work (count, lift);
```

in *main*() passes an integer variable and the address of the function *lift*() to the function *work*(). Notice 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 forward declaration for *work*() is given at the beginning of the *main*() function.

The formal parameter *func* in *work*() is declared to be a pointer to a function taking one **int** argument and returning a **long**. The parentheses around the parameter name are required; without them, the declaration would specify a function returning a pointer to a **long**.

The function *work*() calls the selected function by using the function call:

```
(*func)(i);
```

One argument, *i*, is passed to the called function.

7.4.1 Actual Arguments

An actual argument can be any value with fundamental, structure, union, or pointer type. Although arrays and functions cannot be passed as parameters, pointers to these items can be passed.

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 variables from which it was originally derived.

Pointers provide a way to access a value by reference from a function. 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.

Each expression in a function call is evaluated and converted as follows. If an argument type list for the called function is available, the usual arithmetic conversions are performed independently on each expression in the expression list and on each type in the argument type list. Each expression in the expression list is then compared with the type name that occupies the corresponding position in the argument type list. The value of the expression is converted (if necessary) to the named type as if by assignment.

Next, the converted expression is compared with the type of the formal parameter that has the same place in the parameter list as the expression has in the expression list. (The formal parameters also undergo the usual arithmetic conversions before the comparison.) No conversions are performed, but the compiler produces warning messages as if the expressions were assigned to the formal parameters.

The number of expressions given in the expression list must match the number of formal parameters, unless the function's forward declaration explicitly specifies a variable number of arguments. In this case, the compiler checks as many arguments as there are type names in the argument type list and converts them, if necessary, as described above. If there are additional actual arguments in the function call, each additional argument undergoes the usual arithmetic conversions, but is not otherwise converted or checked.

If the argument type list contains the special type name **void**, the compiler expects zero actual arguments in the function call and zero formal parameters. It produces a warning message if it finds otherwise.

If the argument type list is empty (omitted) or the called function has no forward declaration, the compiler performs no type-checking, either for type or for number of arguments. In this case, the actual arguments in the function call, if any, undergo the usual arithmetic conversions independently before they are placed on the stack.

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 can be misinterpreted.

Type mismatches between actual and formal parameters can produce serious errors, particularly when the mismatches entail size differences. Keep in mind that these errors are not detected unless an argument type list is given in the forward declaration of the function.

Example:

```
main()
{
   void swap(int *, int *);
   int x, y;
   .
   .
   .
   swap(&x, &y);
}

void swap(a, b)
int *a, *b;
{
   int t;

   t = *a;
   *a = *b;
   *b = t;
}
```

In the above example, the *swap*() function is declared in *main*() to have two arguments, both pointers to integers. The formal parameters *a* and *b* are also declared as pointers to integer variables. In the function call:

```
swap (&x, &y)
```

the address of *x* is stored in *a* and the address of *y* is stored in *b*. There are two names, or aliases, for the same location. References to ***a* and ***b* in *swap* are effectively references to *x* and *y* in *main*(). The assignments within *swap* change the contents of *x* and *y*.

The compiler performs type-checking on the arguments to *swap* because an argument type list is present in the forward declaration of *swap*. The types of the actual arguments match both the argument type list and the formal parameters.

7.4.2 Calls with a Variable Number of Arguments

To call a function with a variable number of arguments, the programmer simply gives any number of arguments in the function call. In the forward declaration of the function (if there is one), a variable number of arguments is specified by placing a comma at the end of the argument type list (see Section 4.5 of Chapter 4, "Declarations"). One argument must be present in the function call for each type name specified in the argument type list. If only a comma (but no type names) is given, no arguments are required when calling the function. Refer to **varargs**(F) in the XENIX *Reference* for information on using variable arguments lists.

All the arguments given 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. The programmer is responsible for retrieving any additional arguments from the stack and for determining how many arguments are present.

Example:

```
main()
{
   int scores(int, );
   int count, average, i;
    .
    .
    .
   average = scores(count, 14, 96, 82);
    .
    .
    .
}

scores(number)
int number;
{
   int *ip, total = 0, i;
   ip = &number + 1;

   for (i=1;i <= number;i++, ip++)
      total += *ip;

   if (number > 0)
      return (total/number);
   return (-1);
}
```

The above example shows a function named *scores*() that takes a variable number of arguments. The forward declaration of *scores* in *main*() establishes that *scores*() has at least one argument, an **int**. The comma at the end of the argument type list means that there may be more undeclared arguments.

In the call to *scores*, four actual arguments are passed. The first argument is checked for compatibility with the argument type list and the formal parameter of *scores*. Since the types match, no conversions or warning messages are necessary.

In the definition of the *scores*() function, one formal parameter is declared. The additional arguments are retrieved by taking the address of the previous argument (*number* in the first case), increasing it, and retrieving the value at that address. This procedure works because arguments passed to a function are stored in order on the stack.

The *number* argument is assumed to hold the number of additional arguments, so its value controls how many additional arguments are retrieved from the stack. When *number* arguments have been retrieved and added to the total, the average of the scores is returned to the *main* function. The value −1 is returned if *number* is zero.

7.4.3 Recursive Calls

Any function in a C program can be called recursively. A function can therefore call itself. The C compiler allows any number of recursive calls to a function. On each call, 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. Previous parameters are inaccessible to all versions of the function except the version in which they were created.

Notice that variables declared with global 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 defines no limit on 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

8.1 Introduction

The C preprocessor is a text processor used to manipulate the text of a source file before compilation. The compiler ordinarily invokes the preprocessor in its first pass, but the preprocessor can also be invoked separately to process text without compiling. This chapter explains the main tasks performed by preprocessor directives and describes each directive in detail.

Preprocessor directives are typically used to make source programs easy to modify and to compile in different execution environments. Directives in the source file instruct 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, and suppress compilation of a portion of the file by removing blocks of text.

The C preprocessor recognizes the following directives:

```
#define     #ifdef
#elif       #ifndef
#else       #include
#endif      #line
#if         #undef
```

The number sign (#) must be the first nonwhitespace character on the line containing the directive. Whitespace characters can appear between the number sign and the first letter of the directive. Some directives are followed by arguments or values, as described below. Directives can appear anywhere in a source file, but they apply only to the remainder of the source file in which they appear.

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 an identifier is defined, it cannot be redefined to a different value without first removing the definition. However, the identifier can be redefined with exactly the same definition. Thus, a program is allowed to contain more than one occurrence of the same definition.

The **#undef** directive removes the definition of an identifier. Once the definition has been removed, the identifier can be redefined to a different value. Sections 8.2.1 and 8.2.2 discuss the **#define** and **#undef** directives respectively.

Macros can be defined to look and act like function calls. Because macros do not generate actual function calls, replacing function calls with macros can improve execution time. However, macros create problems if they are not defined and used with care. Macro definitions with arguments may require the use of parentheses to preserve the proper precedence in an expression. In addition, macros may not handle expressions with side effects correctly. See the examples in Section 8.2.1 for details.

8.2.1 Define Directive

Syntax

#define *identifier text*
#define *identifier* (*parameter-list*) *text*

The **#define** directive substitutes the given *text* for subsequent occurrences of the specified *identifier* in the source file. The *identifier* is replaced only when it forms a token. (Tokens are described in Chapter 2, "Elements of C," and in Appendix B.) For instance, the *identifier* is not replaced when it occurs within strings or as part of a longer identifier.

If a *parameter-list* appears after the *identifier*, the **#define** directive replaces each occurrence of *identifier*(*argument-list*) with a version of *text* modified by substituting actual arguments for formal parameters.

The *text* consists of a series of tokens, such as keywords, constants, or complete statements. One or more whitespace characters must separate the *text* from the *identifier* (or from the closing parenthesis of the *parameter-list*). If the text is longer than one line, it can be continued onto the next line by preceding the newline character with a backslash (\).

The *text* can also be empty. The effect of this option is to remove instances of the given *identifier* from the source file. The *identifier* is still considered defined, however, and yields the value 1 when tested with the **#if** directive (discussed later in this chapter).

The *parameter-list*, when given, 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 between the *identifier* and the opening parenthesis are allowed.

Formal parameter names appear in *text* to mark the places where actual values will be substituted. Each parameter name can occur more than once in the *text*, and the names can appear in any order.

The actual arguments following an instance of the *identifier* in the source file are matched to the formal parameters of the *parameter-list*, and the *text* is modified by replacing each formal parameter with the corresponding actual argument. The actual *argument-list* and the formal *parameter-list* must have the same number of arguments.

Arguments with side effects sometimes cause macros to produce unexpected results. A macro definition may contain more than one occurrence of a given formal parameter. If that formal parameter is replaced by an expression with side effects, the expression, with its side effects, is evaluated more than once (see Example 4 below).

Examples:

```
1. #define WIDTH    80
   #define LENGTH   (WIDTH+10)

2. #define FILEMESSAGE "Attempt to create \
   file failed because of insufficient space"

3. #define REG1    register
   #define REG2    register
   #define REG3

4. #define MAX(x,y)  ((x) > (y)) ? (x) : (y)

5. #define MULT(a,b)  ((a) * (b))
```

The first 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 with "(WIDTH + 10)," which is in turn replaced with the expression "(80 + 10)." The parentheses around "WIDTH + 10" are important because they control the interpretation in a statement such as the following:

```
var = LENGTH * 20;
```

After the preprocessing stage the statement becomes:

```
var = (80+ 10) * 20;
```

or 1800.

Without parentheses, the result is:

```
    var=80+10*20;
```

which evaluates to 280 because the multiplication operator (*) has higher precedence than the addition operator (+).

The second example defines the identifier *FILEMESSAGE*. The definition is extended to a second line by using the backslash escape character (\).

The third 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. See the discussion of the **#if** directive later in Section 8.4.1 for an expanded version of this example.

The fourth example defines a macro named *MAX*. Each occurrence of the identifier *MAX* following 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 with:

```
    ((1) > (2)) ? (1) : (2)
```

and the occurrence:

```
    MAX(i,s[i])
```

is replaced with:

```
    ((i) > (s[i])) ? (i) : (s[i])
```

This macro is easier to read than the corresponding expression, making the source program easier to understand.

Notice that arguments with side effects may cause this macro to produce unexpected results. For example, the occurrence "MAX(i, s[i++])" is replaced with "((i) > (s[i++])) ? (i) : (s[i++])". The expression "(s[i++])" is evaluated twice, so by the time the ternary expression has been fully evaluated, *i* has increased by two. The result of the ternary expression is unpredictable, since the operands of the ternary expression can be evaluated in any order, and the value of *i* varies depending on the evaluation order.

The fifth 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 is "3 + 4 * 5 + 6," which evaluates to 29 because the multiplication operator (*) has higher precedence than the addition operator (+).

8.2.2 Undefine Directive

Syntax

#undef *identifier*

The **#undef** directive removes the current definition of *identifier*. The preprocessor ignores subsequent occurrences of *identifier*. To remove a macro definition using **#undef**, give only the macro *identifier*. Do not give a parameter list.

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 compilation of portions of the source program.

Example:

```
#define WIDTH       80
#define ADD(X, Y)    (X) + (Y)
 .
 .
 .
#undef WIDTH
#undef ADD
```

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. The **#undef** directive can also be applied to an identifier that has no previous definition. This ensures that the identifier is undefined.

8.3 Include Files

Syntax

> **#include** *pathname*
> **#include** <*pathname*>

The **#include** directive adds the contents of a given "include file" to another file. Constant and macro definitions can be organized into include files and added to any source file by using **#include** directives. Include files are also useful for incorporating declarations of external variables and complex data types. The types need only be defined and named 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 of the directive. 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 *pathname* is a filename optionally preceded by a directory specification. It must name an existing file. The syntax of the file specification depends on the specific operating system on which the program is compiled.

The preprocessor uses the concept of a "standard" directory or directories to search for included files. The location of the standard directories for include files depends on the implementation and the operating system. See your system documentation for a definition of the standard directories.

The preprocessor stops searching as soon as it finds a file with the given name. If a complete, unambiguous pathname for the include file is given, either in double quotation marks (" ") or in angle brackets (< >), the preprocessor searches only that pathname and ignores the standard directories.

If the file specification does not give a complete pathname, and the file specification is enclosed in double quotation marks, the preprocessor searches for the file in the same directory as the including file first (the "current working directory"). It then searches directories specified in the compiler command line and finally searches the standard directories.

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 directories specified in the compiler command line and then searches the standard directories.

An **#include** directive can be nested. In other words, the directive can appear in a file named by another **#include** directive. When the preprocessor encounters the nested **#include** directive, it processes the named file and inserts it into the current file. The preprocessor uses the same search procedures outlined above in searching for nested include files.

The new file can also contain **#include** directives. Nesting can continue up to ten levels. Once the nested **#include** is processed, the preprocessor continues to insert the enclosing include file into the original source file.

Examples:

1. #include <stdio.h>

2. #include "defs.h"

The first example adds the contents of the file named *stdio.h* to the source program. The angle brackets cause the preprocessor to search the standard directories for *stdio.h*, after searching directories specified in the command line.

The second example adds the contents of the file specified by *defs.h* to the source program. The double quotation marks mean that the directory containing the current source file is searched first.

8.4 Conditional Compilation

This section describes the syntax and use of directives that control "conditional compilation." These directives allow for suppressing compilation of portions of a source file. They test a constant-expression or an identifier to determine which text blocks are passed on to the compiler and which are removed from the source file in the preprocessing stage.

8.4.1 If, Elif, Else, and Endif Directives

Syntax

#if *restricted-constant-expression*
  [*text*]
[**#elif** *restricted-constant-expression*
  text]
[**#elif** *restricted-constant-expression*
  text]
   .
   .
   .
   .
[**#else**
  text]
#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. Zero or more **#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 blocks of *text* for further processing. A *text* block is any sequence of text. It can occupy more than one line. Usually the *text* block is program text that has meaning to the compiler or the preprocessor. However, this is not a requirement; the preprocessor can be used to process any kind of text.

The selected *text* is processed by the preprocessor and passed to the compiler. If the *text* contains preprocessor directives, those directives are carried out.

Any text blocks not selected by the preprocessor are removed from the file in the preprocessing stage and are therefore not compiled.

The preprocessor selects a single *text* block by evaluating the *restricted-constant-expressions* following each **#if** or **#elif** directive until a true (nonzero) *restricted-constant-expression* is found. All *text* between the first true *restricted-constant-expression* and the next number sign (#) is selected.

If no *restricted-constant-expression* is true, or if there are no **#elif** directives, the preprocessor selects the text after the **#else** clause. If the **#else** clause is omitted, and no *restricted-constant-expression* in the **#if** block is true, no text is selected.

Each *restricted-constant-expression* follows the rules for restricted-constant-expressions discussed in Section 5.2.10 of Chapter 5, "Expressions and Assignments". Such expressions cannot contain **sizeof** expressions, type casts, or enumeration constants, but they can contain the special constant-expression "defined(*identifier*)." This constant-expression is considered true (nonzero) if the given *identifier* is currently defined. Otherwise, the condition is false (zero). An identifier defined as emptytext is considered defined.

The **#if**, **#elif**, **#else**, and **#endif** directives can nest in the text portions of other **#if** directives. When nested, each **#else**, **#elif**, and **#endif** directive belongs to the closest preceding **#if** directive.

Examples:

```
1. #if defined(CREDIT)
       credit();
   #elif defined(DEBIT)
       debit();
   #else
       printerror();
   #endif

2. #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
```

```
3. #if DLEVEL==0
      #define STACK0
   #elif DLEVEL==1
      #define STACK100
   #elif DLEVEL > 5
      display( debugptr );
   #else
      #define STACK200
   #endif
4. #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
      #endif
   #endif
```

In the first 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.

The next two examples assume a previously defined manifest constant, *DLEVEL*. The second example 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.

In the third example, **#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 not defined, "display(debugptr);" is compiled and *STACK* is not defined.

The fourth example uses preprocessor directives to control the meaning of **register** declarations in a portable source file. The compiler assigns **register** storage to variables in the same order in which the **register** declarations appear in the source file. If a program contains more **register** declarations than the machine can accommodate, the compiler honors earlier declarations over later ones. Loss of efficiency can occur if the variables declared later are more heavily used.

The definitions listed above 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 Ifdef and Ifndef Directives

Syntax

#ifdef *identifier*
#ifndef *identifier*

The **#ifdef** and **#ifndef** directives accomplish the same task as the **#if** directive used with "defined(*identifier*)." These directives can be used 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 (zero).

The **#ifndef** directive checks for exactly the opposite 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 (zero).

8.5 Line Control

Syntax

#line *constant* ["*filename*"]

The **#line** directive instructs the preprocessor to change the compiler's internally stored line number and filename to a given line number and filename. The compiler uses the internally stored line number and filename to refer to errors encountered during compilation. The line number normally refers to the current input line; the filename refers to the current input file. The line number is increased after each line is processed.

Changing the line number and filename causes the compiler to ignore the previous values and to continue 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 the generated program.

The *constant* value in the **#line** directive is any integer constant. The *filename* can be any combination of characters. It must be enclosed in double quotation marks (""). If *filename* is omitted, the previous filename remains unchanged.

The current line number and filename are always available through the predefined identifiers __LINE__ and __FILE__. The __LINE__ and __FILE__ identifiers can be used to insert self-descriptive error messages into the program text.

Examples:

1. #line 151 "copy.c"

2.
```
#define ASSERT(cond)   if(!cond)\
 {printf("assertion error line %d, file(%s)\n", \
  __LINE__,__FILE__);}else;
```

In the first example, the internally stored line number is set to 151 and the filename is changed to *copy.c*.

In the second example, the macro *ASSERT* uses the predefined identifiers "__LINE__" and "__FILE__" to display an error message about the source file if a given "assertion" is not true.

Appendix A
Differences

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 differences are listed 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**, **enum** and **void**. (The **const** keyword is not yet implemented but is reserved for future use.) The identifiers **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	Hexadecimal and octal constants are treated as unsigned values and are not sign-extended in type conversions.
2.4.3	Hexadecimal **bit** patterns consisting of a backslash (\), the letter "x," and up to two hexadecimal digits are permitted as character constants (for example, \x12). Microsoft C defines two additional escape sequences: the sequence \v represents a vertical tab (VT), and the sequence \" represents the double quote character. Character constants always have type **char**, with the result that they are sign-extended in type conversions.
2.6	The **short** type is always 16 bits in length, the **long** type 32 bits. The size of an **int** is machine dependent. On 8086/80286 processors an **int** is 16 bits long, and on 68000 and 80386 machines it is 32 bits.
4	The **char** type is signed, with the result that a **char** value is sign-extended in type conversions.

Two additional unsigned types are supported: **unsigned char** and **unsigned long**.

Microsoft C offers an additional fundamental type, the **enum** (enumeration) type. The **void** type is defined as the return type of functions that do not return a value.

6.5 The keyword **unsigned** can be applied as an adjective to any integer type (**char**, **int**, **short**, or **long**). When **unsigned** stands alone, it is taken to mean **unsigned int**.

6.6 The arithmetic conversions carried out by the Microsoft C Compiler are outlined in Sections 5.3.1 and 5.7 of Chapter 5, "Expressions and Assignments". Although compatible with the Kernighan and Ritchie conversions, the Microsoft C conversions are spelled out in greater detail, including the specific path for each type of conversion.

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** and **void** are additional type specifiers. Additional acceptable combinations are **unsigned char**, **unsigned short**, **unsigned short int**, **unsigned long**, and **unsigned long int**.

8.4 Optional 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 must be declared **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** statement has **enum** or integral type. Each of the **case** constant-expressions is cast to the type of the *expression*.

12 The number sign (#) introducing the preprocessor directive can be preceded by any combi-

	nation of whitespace characters. Whitespace can also occur between the number sign and the preprocessor keyword.
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 expression before the arrow must have the same type (or 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

keyword
identifier
constant
string
operator
separator

B.1.1 Keywords

auto	**default**	**float**	**register**	**switch**
break	**do**	**for**	**return**	**typedef**
case	**double**	**goto**	**short**	**union**
char	**else**	**if**	**sizeof**	**unsigned**
const†	**enum**	**int**	**static**	**void**
continue	**extern**	**long**	**struct**	**while**

The following identifiers may be keywords, depending on whether the corresponding option is enabled when the program is compiled. See your system documentation for details.

far
fortran
huge
near
pascal

B.1.2 Identifiers

identifier:
 letter
 underscore
 identifier letter
 identifier underscore
 identifier digit

†Not yet implemented.

letter: one of:
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:
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:
1 2 3 4 5 6 7 8 9

octal-constant:
0*octal-digit*
octal-constant octal-digit

octal-digit: one of:
0 1 2 3 4 5 6 7

hexadecimal-constant:
  0x*hexadecimal-digit*
  0X*hexadecimal-digit*
  hexadecimal-constant hexadecimal-digit

hexadecimal-digit: one of:
  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

floating-point-constant:
  fractional-constant exponent
  fractional-constant
  digit-seq exponent

fractional-constant:
  digit-seq . *digit-seq*
  . *digit-seq*
  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 quote ('), backslash (\), or newline character.

escape-sequence: one of:

```
\'    \"    \\    \ddd    \xddd    \b
\f    \n    \r    \t      \v
```

enum-constant:
  identifier

B.1.4 Strings

string-literal:

  char-seq

char-seq:
  char
  char-seq char

B.1.5 Operators

operator: one of:

```
!      ~      ++     --     +
-      *      /      %      <<
>>     <      <=     >      >=
==     !=     |      &      ^
&&     ||     =      +=     -=
*=     /=     %=     >>=    <<=
&=     ^=     |=     ?:     ,
[]     ()     .      ->
```

B.1.6 Separators

separator: one of:

```
[  ]  (  )  {  }
*  ,  :  =  ;  #
```

B.2 Expressions

In this section (B.2), the brackets shown are part of the syntax for the language and should be interpreted literally.

expression:
  identifier
  constant
  string
  expression(*expression-list*)
  expression()
  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:
  - ~ ! * &

lvalue:
identifier
expression[*expression*]
expression.expression
expression->expression
**expression*
(*type-name*)*expression*
(*lvalue*)

type-name:
See Section B.3, “Declarations.”

binary-expression:
expression binop expression

binop: one of:

*	/	%	+	−
<<	>>	<	>	<=
>=	==	!=	&	|
^	&&	||	,	

ternary-expression:
expression ? expression : expression

assignment-expression:
lvalue++
lvalue−−
++*lvalue*
−−*lvalue*
lvalue assignment-op expression

assignment-op: one of:

=	*=	/=	%=	+=	−=
<<=	>>=	&=	|=	^=	

constant-expression:
  identifier
  constant
  (*type-name*)*constant-expression*
  unary-expression
  binary-expression
  ternary-expression
  (*constant-expression*)

B.3 Declarations

In this section (B.3), the brackets shown are part of the syntax for the language and should be interpreted literally.

declaration:
  sc-specifier type-specifier declarator-list;
  type-specifier declarator-list;
  sc-specifier declarator-list;
  type-specifier;
  typedef *type-specifier declarator-list*;

sc-specifier:
  auto
  extern
  register
  static

type-specifier:
  char
  double
  enum-specifier
  float
  int
  long
  longint
  short
  shortint
  struct-specifier
  typedef-name
  union-specifier
  unsigned
  unsigned char
  unsigned int
  unsigned long
  unsigned long int
  unsigned short
  unsigned shortint

enum-specifier:
  enum *tag* {*enum-list*}
  enum {*enum-list*}
  enum*tag*

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-listmember-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
    declarator[ ]
    declarator[*constant-expression*]
    **declarator*
    declarator( )
    declarator(*arg-type-list*)
    (*declarator*)

arg-type-list:
    type-name
    arg-type-list, *type-name*
    arg-type-list,
    void
    ,
    void*

type-name:
    type-specifier
    type-specifier abstract-declarator

abstract-declarator:
    *
    [ ]
    (*arg-type-list*)
    **abstract-declarator*
    abstract-declarator *
    abstract-declarator[ ]
    abstract-declarator[*constant-expression*]
    [ ]*abstract-declarator*
    [*constant-expression*]*abstract-declarator*
    abstract-declarator( )
    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*

B.4 Statements

In this section (B.4), brackets enclose optional portions of the syntax.

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

B.5 Definitions

In this section (B.5), brackets enclose optional portions of the syntax.

definition:
    function-definition
    data-definition

function-definition:
    [*sc-specifier*] [*type-specifier*] *declarator* ([*parameter-list*])
        [*parameter-decs*] *compound-statement*

parameter-list:
    identifier
    parameter-list , *identifier*

parameter-decs:
    declaration
    declaration-list declaration

data-definition:
    declaration

B.6 Preprocessor Directives

In this section (B.6), brackets enclose optional portions of the syntax.

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 for **sizeof** expressions, casts, and enumeration constants.

C User's Guide

Contents and Introduction Missing

Chapter 2

CC: A C Compiler

2.1 Overview

This chapter describes how to use the C compiler, **cc**. It describes filename conventions, the command line, the many options, the compiling and linking processes, the different memory model configurations, and the special keywords.

It is assumed that you are familiar with the C language and that you know how to create C language programs using a text editor. The structure and syntax of C are explained in the *XENIX C Language Reference*. If you need help with a text editor, see Chapter 2 of the *XENIX User's Guide*. For a complete list and explanation of the messages produced by the C compiler and link editor, see Appendix B of the *XENIX C User's Guide*.

2.2 Filename Conventions

There are several kinds of files that can be put on the **cc** command line and several kinds of files that are produced as output. **cc** assumes that the suffix of a file's name identifies that file. A file whose name ends in '*.c*' is assumed to be C source, and conversely, if a file contains C source, its name must end in '*.c*'. Here are the filenaming conventions that **cc** follows:

Type of File	Suffix
C Source	.c
Include File	.h
Masm Source	.s
Object File	.o
Library File	.a
Preprocessed C Source	.i
Assembler Listing	.L
C Source Listing	.S
Link Map Listing	.map

Include files are never put on the **cc** command line but the '*.h*' is still a conventional suffix. It stands for "header" because include files are generally put at the top (head) of C source files.

2.3 The cc Command Line

The arguments to **cc** consist of files and options. The simplest command would be:

```
cc t.c
```

cc translates the C source in the file *t.c* into object code, links it together with the standard C library and produces an executable output file, named *a.out* by default.

To execute the program one simply enters:

```
        a.out
```

cc invokes the compiler passes for each C source file and the XENIX assembler, **masm**, for each assembly language source file. Files are processed in the order that they are found on the command line. The **cc** and **masm** commands ignore all object (*.o*) and library (*.a*) files until all source files have been compiled or assembled.

Both the compiler and the assembler generate files with '*.o*' suffixes. These files contain relocatable object code. The object files have the same names as the source files but different suffixes. When the compiler and the assembler have successfully completed their code generation, they pass control to the link editor, **ld**. The link editor joins the object files with the library files, resolves external references and outputs an executable file.

Another example:

```
        cc -O figure.c mach.s util.o -o calc -lm
```

The **−O** option tells **cc** to "optimize" the generated code. The C source *figure.c* will be compiled and the assembler source *mach.s* will be assembled. Assuming there are no syntax errors, this invocation of **cc** will produce two new object files named *figure.o* and *mach.o*. These two object files are linked together with *util.o* (from a previous **cc**) and the math library (**−lm**) to produce an executable output file. The **−o** option causes this file to be named *calc* instead of *a.out*.

Following normal Unix conventions, if a single '*.c*' file is compiled and linked at once, the '*.o*' file will be removed. If there is an environment variable named QUIETCC that is set to a non-null value, then **cc** will not echo the filename when compiling a single '*.c*' file. If there are multiple source files, the filename *will* still be echoed before each compilation is done.

2.4 Command Line Options

There are many options available to control the actions of **cc**. These are presented in two formats. First, a table arranged in alphabetical order is presented with a brief explanation of each option. The options are then arranged in functional groups with complete descriptions of their syntax and use.

2.4.1 Alphabetical List of Options

–c	Creates object modules but does not call the linker. (2.4.2)
–C	Preserves comments when preprocessing a file (only valid when used with **–P**, **–E**, or **–EP**). (2.4.5)
–compat	Makes the executable output binary compatible across several machines and systems. (2.4.4)
–CSOFF	When used with **–O**, turns off common subexpression optimization. (2.4.6)
–CSON	When used with **–O**, turns on common subexpression optimization. (2.4.6)
–d	Shows the passes as they are executed. (2.4.10)
–D*name*[=*string*]	Defines *name* to the preprocessor. The value is either *string* or 1 if "=*string*" is not given. (2.4.5)
–dos	Makes a program to run on DOS, not XENIX. (2.4.8)
–E	Preprocesses each source file sending the result to the standard output. Prepends a #line directive. (2.4.5)
–EP	Same as **–E** but without the #line directive. (2.4.5)
–F*num*	Sets the size of the program stack. *num* is in hexadecimal. (2.4.4)
–Fa[*name*]	Makes an assembly source listing in *source.s*. Continues with the link. (2.4.3) In all of the **–Fx** listing options the name of the listing file can be controlled with *name*.
–Fc[*name*]	Makes a merged C and assembly listing in *source.L*. (2.4.3)
–Fe*name*	Names the executable program file *name*.
–Fl[*name*]	Makes an assembly and object code listing in *source.L*. (2.4.3)
–Fm[*name*]	Makes a link map listing in *a.map*. (2.4.3)
–Fo*name*	Renames the object file to *name*. Note that *name* is required with this option. (2.4.11)
–FPxx	Controls floating point operations when used with **–dos**. (2.4.8)
–Fs[*name*]	Makes a C source listing in *source.S*. (2.4.3)
–g	Includes symbol information for **sdb**(CP). (This is equivalent to the **–Zi** option.)
–i	Creates separate instruction and data spaces in small model programs. (2.4.4)
–I*pathname*	Adds *pathname* to the list of directories to be searched for include files. (2.4.5)
–K	Removes stack probes. (2.4.11)
–l*name*	Searches library *name* for unresolved function references. (2.4.4)
–L	Makes an assembly and object code listing in *source.L*. (2.4.3)

−LARGE	Uses the large model passes. Available only on machiues that support large model programs. (2.4.2)
−M*string*	Sets the memory model, word order, data threshold, machine type, enables special keywords and non-ANSI extensions. *string* is a series of letters from "**smlh0123ebdt**". (2.4.9)
−n	Sets pure text model. This is equivalent to the **−i** option. (2.5.2)
−ND*name*	Renamesthe data segmentto *name*. (2.4.9)
−nl *num*	Restricts the length of external symbols to *num*. (2.4.4)
−NM*name*	Renames the module to *name*. (2.4.9)
−NT *name*	Renames the text segment to *name*. (2.4.9)
−o *name*	Renames the executableoutputfile to *name*. Default is *a.out*. (2.4.2)
−O[*string*]	Optimizesthe generated code. (2.4.6)
−p	Generates code for profiling. (2.4.11)
−P	Preprocesses the C source putting the result in *source.i*. (2.4.5)
−pack	Packs structure members. (2.4.6)
−s	Strips the symbol table from the executable output. (2.4.4)
−S	Makes an assemblylistingin *source.s*. (2.4.3)
−SEG	Sets the maximum number of segments the linker can handle to *num*. (2.4.4)
−u	Removesall manifest defines. (2.4.5)
−U*definition*	Removes the given manifest definition. (2.4.5)
−V*string*	Copies the *string* to the object file. Used for version control. (2.4.11)
−w	Suppresses warning messages from the compiler. (2.4.11)
−W*num*	Sets the output level for compiler warningmessages. (2.4.11)
−X	Removes */usr/include* from the list of files to be searched for #include files. (2.4.5)
−z	Shows the passes but does not execute them. (2.4.10)
−*Zi*	Includes information used by the symbolic debugger (sdb) in the output file.
−Zp1, 2, 4	Options for data alignment consistent with different processors (applies to 80386 processors only). (2.4.7)

2.4.2 Everyday Options −c, −o, −LARGE

The **−c** option creates a linkable object file for each source file but does not link these files. No executable program is created. This option suppresses the invocation of the link editor.

```
        cc -c subr.c
```

The **-o** option allows you to specify an executable filename other than the default filename, **a.out**.

```
        cc -o calc arith.c sqrt.c util.o
```

The new name may not end in '*.o*' or '*.c*'.

If **cc** gives the error message "out of heap space", using the **-LARGE** option may help. **-LARGE** tells the driver to use a different set of large model compiler passes which may enable you to compile larger source files. On machines that do not support large model programs the only alternative is to split the source files into smaller pieces.

```
        cc -LARGE monster.c
```

2.4.3 Listing Options -S,-L,-Fa,-Fc,-Fl,-Fm,-Fs

Assembly language listing files are used by programmers who wish to debug their program with **adb**(CP). Since **adb** recognizes machine instructions instead of the C source statements, an assembly language listing is helpful for debugging.

The **-S** option creates an assembly source listing of the compiled C source file and copies the listing to a file with a name the same as the source file but with a '*.s*' suffix.

```
        cc -S bug.c
```

The **-L** option creates an assembler listing file containing object code and assembly source instructions. The listing is copied to a file with a name the same as the source file but with a '*.L*' suffix.

```
        cc -L hardbug.c
```

The **-L** option does not produce code suitable for assembly. It is provided so the compilation can be stopped and the intermediate assembly source studied.

The **-F***x* options (where *x* = **a**, **c**, **l**, **m** or **s**) produce a variety of listings.

x	Type of Listing	Default Suffix
a	Assembly	.s
c	Merged Assembly and C Source	.L
l	Assembly and Object Code	.L
m	Link map	.map
s	C Source	.S

All of the **−F***x* options can be called in the following ways to control the name of the listing file:

Syntax	Listing File Name
−Fa	source.s
−Faname	name.s
−Faname.lst	name.lst

For example,

```
cc −Fclist confusing.c
```

produces a listing file named "*list.L*" containing the generated assembly code interleaved with the C source.

−Fa and **−Fl** produce the same listings as **−S** and **−L** respectively. The difference is that **−S** and **−L** stop after producing the listings and do *not* call the linker. **−Fa** and **−Fl** *will* continue on to call the linker.

In all the listing files, the names of globally visible functions and variables begin with an underscore. This is important to keep in mind when writing assembler programs that interface with C code.

If you request optimization with **−O**, the listing files will reflect the optimized code. Since optimization involves rearrangement of code, the correspondence between your C source and the generated code may not be clear. This is especially true with the **−Fc** option where the C source is interleaved with the generated assembly code.

2.4.4 Linker Options −l,−s,−F,−i,−SEG,−nl,−compat

The options in this section have no effect unless **cc** will be calling the linker. If the **−c** option is given, the linker is not called and these options will be ignored.

The **−l** option causes the specified library to be searched for unresolved references to functions. A library is a convenient way to store a large collection of object files. The XENIX system provides several libraries, the most important of which is the standard C library. Functions in this library are automatically linked to your program whenever you invoke the **cc** compiler without the **−c** option.

Library files in the command line are examined only if there are unresolved external references encountered from previous object files. Library files must have been processed with **ranlib**(CP). For information on the library format, see **ar**(F) and the XENIX *C Library Guide*.

The **cc** command does not search a library until it encounters the **-l** option, so the placement of the option is important. The option must follow the names of any source files containing calls to functions in the given library. Note that the *name* provided with the **-l** option comprises only a part of the actual library name. Thus,

```
        cc backgammon.c -lcurses
```

would cause the library */lib/Slibcurses.a* to be searched. The '*S*' stands for "small" and denotes the memory model. It would be '*M*' or '*L*' if the **-Mm** or **-Ml** options were used. **cc** first looks in */lib* for the requested library. If it is not there it checks */usr/lib*. If it is not there either, **cc** stops and does not call the linker.

The size of an executable file can be reduced by using the **-s** option to strip the symbol table leaving only object code. The symbol table contains information about code relocation and program symbols. The debugger **adb** uses the symbol table to look up symbolic references to variables and functions when debugging. The information in this table is not required for normal execution and can be removed when the program has been completely debugged.

The command **strip**(CP) may also be used to strip the symbol table.

```
        cc -s secret.c
```

The **-F** option allows you to set the size of the program stack to *num* bytes. The program stack is used for storage of function parameters and automatic variables. This option is passed to the link editor. Note that the stack size is given in hexadecimal. This option does not apply to the 80386, which has a variable stack.

```
        cc -F 2000 okay.c
```

On 286 machines the stack size defaults to 1000 (hexadecimal). On 8086 machines the default is a *variable stack*. This means that the stack size starts at the top of a full 64 Kbyte data segment and grows down until it reaches data. This is useful for program development but once a program is debugged, performance can be enhanced by fixing the stack size. A sufficient stack size must be anticipated before the compilation of the program.

The **-i** option creates separate instruction and data spaces for small model programs. When the output file is executed, the program text and data areas are allocated separate physical segments. The text portion is read-only and may be shared by all users executing the file. This option is implied when creating middle or large model programs. This option is passed to the link editor.

```
        cc -i small.o
```

The **-SEG** option on **cc** sets the maximum number of segments that the linker can handle. This number defaults to 128 and can be as large as 1,024. If 1,024 is too small use the **-NT** option to reduce the number of different segment names.

```
        cc -SEG 800 aa.oab.o ... zz.o
```

The **-nl** option sets the maximum length of external symbols to *num*. Names longer than *num* are truncated before being copied to the external symboltable.

```
        cc -nl 10 trunk.c
```

The **-compat** flag is used to create a program that is *binary* compatible – one which can be run, unchanged, on any of these systems:

```
        XENIX-286 System V
        XENIX-286 3.0
        XENIX-8086 System V
```

It is a linker flag because it effects the values in the *x.out* header and the searching of special libraries. Note tbat using any options other than **-M0** will guarantee incompatibility on 8086 processors.

2.4.5 Preprocessor Options -I, -D, -P, -E, -EP, -C, -X, -u, -U

The C compiler invokes the C preprocessor in its first pass. The preprocessor manipulates the contents of a source file prior to compilation.

The preprocessor recognizes a number of directives embedded in the text of the source file. These directives are frequently used to make a program capable of compiling in a number of different execution environments.

Directives in the source file instruct 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 before compilation, and suppress compilation of a portion of the file. For a detailed description of preprocessor directives (and instructions on their uses), see the *XENIX C Language Reference*.

The **-I** option adds the specified *pathname* to a list of directories to be searched when a #include directive is processed. If an included file cannot be found in the directories in this list, directories in the standard list (*/usr/include, /usr/lib, current directory*) are searched.

```
        cc -I/usr/stefanis charlie.c
```

The **-D** option defines a *name* to the preprocessor (as if defined by a **#define** directive in the source). This sets the value of *name* to 1. It is also possible to set the value of *name* to a given *string*.

```
        cc -DWHITE=345 colour.c
```

The **-P** option preprocesses the source file and copies the result to a file with the same name but with a "*.i*" suffix.

The **-E** option is similar to the **-P** option except that the result is directed to the standard output. In addition, a **#line** directive (with the current input line number and source file name) is placed at the beginning of the output for each file. If the output is recompiled, the **#line** directive ensures that the line numbers and filenames are correct in error messages.

The **-EP** option works in the same fashion as the **-E** option but does not insert **#line** directives.

The **-C** option instructs the preprocessor to preserve comments in the source. It may only be used in conjunction with the **-P**, **-E** and **-EP** options.

The **-X** option removes the standard directories from the list of directories to be searched for **#include** files.

There are several "manifest defines" which the preprocessor defines for you. They relate to the type of cpu, the operating system, and the model configuration. They all begin with "M_" for "manifest":

M_I86	This is an Intel processor.
M_XENIX	This is XENIX.
M_SYS3 and M_SYSIII	Unix System III compatible.
M_SYS5 and M_SYSV	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_I8086, M_I186, M_I286, or M_I386	Depending on -M0, -M1, -M2 or -M3.
M_I86SM, M_I86MM, or M_I86LM	
M_SDATA or M_LDATA	
M_STEXT or M_LTEXT	

The last 3 lines depend on the memory model set by -M[sml].

The **-u** option excludes (undefines) all of these manifest defines. To undefine a specific one, use the **-U** option:

```
        cc -UM_XENIX dos.c
```

-U will only undefine manifest constants. It cannot be used to counteract a **#define** within the source.

2.4.6 Optimization Options -O, -CSON, -CSOFF

The **-O** option causes the compiler to reduce the size of the object file by deleting, moving or simplifying instruction sequences. The resulting object file is usually smaller and faster.

This option applies to source files only; existing object files (*.o* files) cannot be optimized with this option. The **-O** option must appear before the names of the files that you wish to optimize.

The syntax of the option is as follows:

-O*string*

The following *string* arguments are available with the **-O** option:

a	Reduces restrictions on aliases.
c	Eliminates common expressions (386 only).
d	Default. Disables optimization.
l	Performs various loop optimizations. Note that some programs do not benefit from loop optimizations, but become larger (386 only).
s	Optimizes code for space.
t	Default. Optimizes code for speed. Equivalent to **-O**.
x	Performs maximum optimization. Equivalent to **-Otacl**.

Note

The **-Oc** and **-Ol** options can be used only with the **-M3** option.

Although optimization is very useful for large programs, it should be used with caution. Once source code has been optimized, the control flow can be very difficult to follow, making debugging difficult.

In the following examples, the **-Oca** option instructs the C compiler to eliminate common expressions and relax alias checking in the generated code. The **-Os** option instructs the C compiler to optimize generated code for space. The **-Otacl** option instructs the C compiler to perform maximum optimization on the generated code. The code is optimized for speed, alias checking, common expressions, and loops.

Examples

```
cc -Oca filename.c
cc -Os filename.c
cc -Otacl filename.c
```

In addition, the options **−CSON** and **−CSOFF** perform two similar functions. **−CSON** can be used to eliminate "common subexpressions." **−CSOFF** turns this option off. These have an effect only in conjunction with **−O**. The default is **−CSOFF** for the small model passes and **−CSON** for the large (with **−LARGE**). If one is having trouble compiling a large program because of suspected bugs or lack of heap space, turning optimization off by omitting **−O** may help. An intermediate step would be to use **−CSOFF** to make the optimization simpler.

2.4.7 Data Alignment Options −pack, −Zp1, −Zp2, -Zp4

When storage is allocated for structures, structure members larger than a **char** are ordinarily stored beginning at an **int** boundary. To conserve space you may want to store your structures more compactly. The **−pack** option causes structure data to be "packed" more tightly into memory. This option is also useful when you want to read existing packed structures from a data file. When you use the **−pack** option, each structure member (after the first) is stored beginning at the first available byte, without regard to **int** boundaries. On most processors, using this option results in slower program execution because of the time required to unpack structure members when they are accessed.

```
cc −pack toobig.c
```

Only the **−pack** option is available with **−M0**, **−M1** or **−M2**. With the **−M3** option, the **−Zp1**, **−Zp2** and **−Zp4** options are also available. On 80386 machines, **−pack** is equivalent to **−Zp1**.

Rules for Structure Packing with Zp 1, Zp2, Zp4

Zp1	*Rule:* No special alignment of structure members takes place.
Zp2	*Rule:* All structure members are aligned so that their offset within the structure is a multiple of 2. *Exceptions:* char and unsigned char types and arrays of these types are not aligned.

Zp4 *Rule:* All structure members are aligned so that their offset within the structure is a multiple of 4.

Exceptions: char and unsigned char types and arrays of these types are not aligned.

Short and unsigned short types and arrays of these types are aligned so that their offset within the structure is a multiple of 2.

Structures whose members consist only of char, unsigned char, short and unsigned short types and arrays of those types are aligned so that their offset within the structure is a multiple of 2.

All other structures are aligned so that their offset is a multiple of 4.

2.4.8 DOS Cross Development Options −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 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 Appendix A, "XENIX to DOS: A Cross-Development System," in the XENIX *C Library Guide* and Appendix A, "C Language Portability," of the XENIX *C User's Guide.*

2.4.9 Model and Segment Options −M, −ND, −NT, −NM

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. For a discussion of memory models see section 2.5.

```
cc −Mstring special.c
```

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

s	Create a small model program. This is the default.
m	Create a middle model program.
l	Create a large model program.
h	Create a huge model program.
e	Enable the keywords: **far**, **near**, **huge**, **pascal** and **fortran**. See section 2.6. Also enables certain non-ANSI extensions necessary to ensure compatibility with existing versions of the C compiler (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.
tnum	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.

Note

The **m**, **l**, **h**, **b**, **t**, or **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**.

"Module" is another name for the object file created by the C compiler. Every module has a name, and the **cc** command uses this name in error messages if problems are encountered during linking. The module name is usually the source file's name (without the "*.c*" or "*.s*" extension). This name may be altered with the **−NM** option.

Changing a module's name is useful if the source file being compiled is actually the output of a program preprocessor and generator, such as **lex**(CP) or **yacc**(CP).

A "segment" is a contiguous block of binary code produced by the C compiler. Every module has 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. This name is used by **cc** to define the order in which the segments of the program will appear in memory when loaded for execution. Text segments having the same name are loaded as a contiguous block of code. Data segments of the same name are also loaded as contiguous blocks.

Text and data segment names are normally created by the C compiler. These default names depend on the memory model chosen for the compilation. 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 segments from all modules of a small model program are loaded as a contiguous block. In middle model programs, each text segment has a different name. In large and huge model programs, each text and data segment has a different name. The default text and data segment names for middle and large model programs are given in "Default Names."

You can override the default names used by the C compiler (and override the default loading order) by using the **-ND** and **-NT** options. These options are useful in middle and large model programs where there is no specific loading order. In these programs, contiguous loading for two or more segments is guaranteed by giving the segments the same name.

-ND*name*	Set the data segment name for each compiled or assembled source file to *name*. If not given, the name "_DATA" is used.
-NT *name*	Set the text segment name for each compiled or assembled source file to *name*. If not given, the name "module_TEXT" is used for middle and large model and "_TEXT" for small model.
-NM *name*	Set the module name for each compiled or assembled source file to *name*. If not given, the filename of each source file is used.

The **-NT** option is recommended only when there are so many different text segments that the **-SEG** linker option is insufficient.

The **-ND** option is recommended only for large model programs. If your program requires extra data space, collect all of the global data declarations and put them in a single '*.c*' file that is compiled with **-ND**. The modules compiled with **-ND** should only contain data. If a module compiled with **-ND** contains code, then the value of the DS segment register may be reloaded with the values for the module data.

2.4.10 Compiler Pass Options -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 (e.g. elimination of redundant code, rearrangement, etc.). It creates the object code and outputs listing files (if requested).

Note that when the **-LARGE** option is used, P0 and P1 are combined into a single pass.

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.

2.4.11 Other Options -W, -w, -p, -K, -Fo, -V

The -w and -W options set the level of warning messages produced by the compiler. These options direct the compiler to display messages about statements that may not be compiled as the programmer intends. Warnings indicate potential problems rather than actual errors.

The -W option will allow arguments in the range of 0,1,2,3. The default is 1. The higher option levels (3,2) are especially useful in the earlier stages of program development when messages about potential problems are most helpful. The lower levels (1,0) are best for compiling programs whose questionable statements are intentionally designed.

0	No warning messages are issued.
1	Only warnings about program structure and overt type mismatches are issued.
2	Warnings about strong typing mismatches are issued.
3	Warnings for all automatic conversions are issued.

The **-W** option does not affect the output of *error* messages.

```
        cc -W3 problem.c
```

The **-w** option prevents the compiler from issuing warning messages. Using the **-w** option is the same as using **-W0**.

```
        cc -w ignore.c
```

The **-p** option adds code for program profiling. Profiling code counts the number of calls to each routine in the program and copies this information to the *mon.out* file. This file can be examined using the **prof**(CP) command.

You can reduce the size of a program by removing all the stack probes with the **-K** option. A stack probe is a short routine called by a function to check the program stack for available space. Probes are not needed if the program makes very few function calls or has well known stack usage. Code generated for the 80386 processor does not require stack probes; therefore, this option has no effect if **-M3** is specified.

Although this option, when combined with the **-O** option, makes the smallest possible program, it should be used with discretion. Removing stack probes from a program where the stack use is not well known can cause execution errors.

```
        cc -K well_tested.c
```

The **-Fo** option can be used to change the name of the object module from its default *source.o*. This is useful if the source file is actually the output of a program generator such as **lex**(CP) or **yacc**(CP).

```
        lex tokenize.l
        cc -c -Fotokenize.o lex.yy.c
```

The **-V** option is useful for version control. It simply copies a *string* to the object file. This string may have any value and should be chosen to identify the version of your program.

```
cc -V 2.1.3 help.c
```

2.5 Memory Models

cc can create programs for four different memory models: small, middle, large and huge. In addition, small model programs may be either pure or impure and memory models may be mixed (under limited circumstances).

The following sections describe the characteristics of the various memory models and the options that allow you to manipulate them.

"Text" refers to the code portion of the program, "data" refers to the data portion of the program.

Note

The only memory model supported for 80386 code is pure small model. All models are supported for 86/286 code.

2.5.1 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 processors. There are no impure 80386 programs. The maximum program size is 64K. **cc** creates impure small model programs by default on 8086/80286 systems. They can also be created using the **-Ms** option.

2.5.2 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.

2.5.3 Middle Model

These programs occupy several physical segments, but only one segment contains data. Text is divided among as many segments as required. All calls default to long calls because this model spans more than one segment. Text can be any size. The data must not exceed 64K. Middle model programs are created using the **-Mm** option. These programs are always pure.

2.5.4 Large Model

These programs occupy several physical segments with both text and data in as many segments as required. All calls default to long calls. Special addresses are used to access data in other segments. Text and data may be any size, but no data structure or item may be larger than 64K. Large model programs are created using the **-Ml** option. These programs are always pure.

The special calls and returns used in middle- and large-model programs may affect execution time. In particular, accessing data can cause significant performance degradation in the large and middle models. The degradation of access time is roughly 10 to 50 percent. Middle-model programs require approximately 10 percent more time, and large- and huge-model programs require approximately 50 percent more time.

In middle, large and huge model programs, function pointers are 32 bits long. In large and huge model programs, data pointers are 32 bits long. Programs making use of such pointers must be written carefully to avoid incorrect declaration and use of these variables.

2.5.5 Huge Model

Huge model programs occupy several segments with both text and data in as many segments as necessary. In small, middle, and large models, data constructs cannot span segments. The maximum size is that of the segment, 64K. With the huge model option, **-Mh**, it is possible to circumvent this limitation.

For the purposes of this discussion, an "item" is defined to be those data constructs that can be elements of an array: integers and floating point numbers, structures, and unions.

The huge model implementation is necessarily a compromise - the run-time cost of executing the code necessary to reference items divided across data segments would be prohibitive. Hence, the huge model implementation has the following limitations:

- An item must be contained within a 64K segment (it must not cross a segment boundary). This means that the address of an item may be used to access any part of the item.
- Each group of segments allocated to an array is 64K bytes long. An exception is the segment allocated to the highest part of the array, which may not be fully used.
- The segments must be contiguous. On a processor running in "real" or "unprotected" mode (e.g. when running DOS) this means that the 64K segments must be contiguous in physical memory. On a processor running in protected mode (e.g. XENIX 286 or 386 running 286 code), this means that the segments must have contiguous LDT entries.

There are several consequences of these limitations. No structure or union can be greater than 64K. If an array is greater than 64K but less than 128K it can be offset within a segment to ensure that the elements align to the 64K boundary. If the array requires three or more segments, the size of the elements in the array must be a power of two. This is always true for scalar elements; composite elements that do not comply are flagged by the compiler and the user must pad them appropriately.

2.5.6 Huge Model Address Calculations

There are two ways that you may force huge address calculations: discrete arrays or pointers can be declared as **huge** (keywords or attributes); or all addressing can be declared as **huge** (huge model).

The arithmetic of huge addresses involves some special considerations. There are pointer increment/decrement operations to add/subtract one element size to a pointer. There are static address calculations (static array indexing). Finally, there are based address calculations such as a pointer to a structure element and indexing into an array on a stack frame. Each of these involves arithmetic that requires adjustment of the selector portion of the address.

While huge and large pointers have the same dimension, they are used differently in addressing memory. Only the low order (the first 16) bits of the large pointer are used. They reference the offset within the segment and arithmetic calculations are done only on them. The high order bits reference the segment location. In huge model pointers, arithmetic is done on all 32 bits. This is necessary because huge data constructs are now capable of spanning segments.

The C compiler produces object code that will link on both DOS and on XENIX. The selector portion of an address is different for these two operating systems, but the compiler produces a canonical sequence of instructions that are adjusted during linking to run correctly in each environment.

There are two more issues that the huge model must address: the value produced by:

```
        sizeof(huge_item)
```

and the resultant type of:

```
        huge_ptr1 - huge_ptr2
```

The **sizeof** operator presents a problem because **sizeof** a **huge** item requires a **long int** for full representation, but **sizeof** is normally an **int**.

In order not to maintain the integrity of the language, **sizeof** remains an **int**. However,

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

produces the correct value. The same arguments apply to the difference of two pointers.
The general case that:

```
        sizeof(ptr1-ptr2) == sizeof(int)
```

is still true. However,

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

produces the correct value.

This is a technical departure from "standard" C, but this method produces the fewest errors when existing code is compiled as a huge model program.

2.5.7 Mixed and Hybrid Models

A "mixed" model program is a program in which some modules are compiled as one model and other modules are compiled in a *different* model. A "hybrid" model is a program that is compiled as only *one* model but which has specific elements (functions or data) that are addressed using the **far** keyword.

In most circumstances, it is neither possible nor advisable to mix memory models. In general, if memory models are mixed, library support is not available. Each model has its own library. Normally, **cc** automatically selects the correct small, middle, large and huge versions of the standard libraries based on the configuration option specified.

Mixing models means that any call to a library routine or function will be likely to result in type mismatches.

The more common and useful practice is to create a *hybrid* model program. For example, a large, but infrequently used, array of error messages can be placed in a different data segment freeing a good deal of dynamic heap space. This array would be referenced with a far or **huge** pointer. This hybrid model would run much more efficiently. See section 2.6.1 for a discussion of the **far** keyword.

2.5.8 Table of Pointer and Integer Sizes

The following table defines the sizes (in bits) of text and data pointers and of integers (**int** type) in each memory model.

Model	Data Pointer	Text Pointer	Integer
Small	16	16	16
Middle	16	32	16
Large	32	32	16
Huge	32	32	16

2.5.9 Table of Default Names

The following table lists the default text and data segment names, and the default module name, for each object file.

Model	Text	Data	Module
Small	_TEXT	_DATA	*filename*
Middle	*module*_TEXT	_DATA	*filename*
Large	*module*_TEXT	_DATA	*filename*
Huge	*module*_TEXT	_DATA	*filename*

2.6 Special Keywords

The special keywords are enabled by the e argument to the **−M** option on the **cc** command line. The use of the keywords: **near**, **far** and **huge** occurs in programs that require more than one segment for either text or data. The use of the keywords **pascal** and **fortran** is necessary when including routines compiled with the Pascal or FORTRAN calling protocol or when compiling a routine as if it were Pascal or FORTRAN.

2.6.1 The near, far and huge keywords

The **near**, **far** and **huge** keywords are special type declarators that make it possible to address items in segments other than the one in which the program is resident. The **near** keyword defines an item with a 16-bit address. The **far** and **huge** keywords define an item with a full 32-bit segmented address. Any data item, construct, or function can be addressed.

The keywords override the normal address length generated by the compiler for variables and functions. In small model programs, **far** allows access to data and functions in segments outside of the near segment. In middle and large model programs, **near** allows access to data with just an offset. In all programs, the **huge** keyword allows you to access an array that spans data segments and that is outside the near segment.

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.

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

Declaration	**Size of Address**	**Size of 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 items 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:

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)†
char far * far s;	far (48 bits)	64 bits (far pointer)††
int foo();	near (32 bits)	function returning 32 bits
int far foo();	far (64/48 bits)	function returning 32 bits†††

† This example is shown for syntactic completeness only.

†† Resembles accessing data in a large-model program

††† This example creates problems in most environments. The far call changes the CS register, and makes run-time support unavailable.

2.6.2 The pascal and fortran Keywords

The **pascal** and **fortran** keywords may be considered synonymous. Both invoke the the PLM protocol. Only the **pascal** keyword is discussed here.

Use the **pascal** keyword to either (1) call routines compiled with the PLM protocol, or (2) compile subroutines with the PLM protocol.

There are a number of items of special interest to assembly language programmers:

- Any external Pascal identifiers are mapped to uppercase and not prefaced by underscores (_). This is true for both global variables and function or procedure names.

- In C, the compiler must readjust the stack pointer after making a call to a routine. In Pascal, this is not necessary. All Pascal routines readjust the stack before they return.

- Conventions for returning floating point numbers and structured items differ between C and Pascal. In C, the return value is stored in a register from which the calling routine retrieves the return value. In Pascal, space is reserved on the stack for this return value and a **near** pointer to it is passed as a hidden parameter.

- The protocol for passing parameters differs between C and Pascal. In C the number of parameters is not fixed; the C compiler pushes the parameters from right to left. In Pascal, since the number of parameters is fixed, the PLM protocol dictates that the parameters are pushed from left to right.

Use the −S option to generate an assembly listing if you need to understand exactly what code is being generated.

Chapter 3

ld: the XENIX Link Editor

3.1 Introduction

The XENIX link editor, **ld**(CP), is a companion tool to both the C compiler, **cc**, and the macro assembler, **masm**(CP).

ld creates executable files by combining object modules and resolving external references. The inputs to **ld** are relocatable object files produced by the C compiler or the macro assembler.

For a synopsis of the information presented in this chapter, we refer you to the **ld** page in the *XENIX Reference.*

3.2 Using the Link Editor

The link editor is invoked with the following form:

ld [*options*] *filename1 filename2* ...

where *options* are of the form described in the next section, and *filename* must be either an object file or an archive library containing object files.

Input object files and archive libraries of object files are linked together to form an executable file. If there are no unresolved references encountered, this file will then be made executable.

Object files have the form *name*.o throughout the examples in this chapter. The names of actual input object files need not follow this convention.

If you merely want to link the object files *file1.o* and *file2.o*, then the following command is sufficient:

```
ld file1.o file2.o
```

No directives to **ld** are necessary. If no errors are encountered during the link edit, the output is left on the default file *a.out*.

3.3 Link Editor Options

Input object files are linked in the order in which they are encountered. Options may be interspersed with filenames on the command line. The ordering of options is not significant.

All options for **ld** must be preceded by a dash (–) on the **ld** command line. Options that carry an argument are separated from that argument by white space (blanks or tabs). Following is a summary of all the available options.

−A *num*	Creates a stand-alone program whose expected load address (in hexadecimal) is *num*. This option sets the absolute flag in the header of the *a.out* file. Such program files can only be executed as standalone programs.
−B *num*	Set the text selector bias to the specific hexadecimal number.
−c *num*	Alters the default target CPU in the x.out header. num can be 0, 1, 2, or 3 indicating 8086, 80186, 80286 and 80386 processors respectively. The default on 8086/80286 systems is 0. The default on 80386 systems is 3. Note that this option only alters the default. If object modules containing code for a higher numbered processor are linked, then that will take precedence over the default.
−C	Ignore case when matching symbols. Normally, the link editor is case-sensitive.
−D *num*	Set the data to the specific hexadecimal number.
−F *num*	Sets the size of the program stack to *num* bytes where *num* is hexadecimal. In programs configured with the **-M0**, **-M1**, or **-M2** option, this option changes the default stack size of 1000 bytes (hexadecimal) to *num* bytes (hexadecimal). In programs configured with the **-M3** option, the size of the stack is automatically controlled by the 80386; the **-F** option is not needed in this case. The **−F** option is incompatible with the **−A** option.
−i	Creates separate instruction and data spaces for small model programs. When the output file is executed, the program text and data areas are allocated separate physical segments. The text portion will be read-only and shared by all users executing the file.
−m *mapfile*	Instructs the link editor to produce a mapfile which contains a description of all the segments in the executable file as well as listings of all public symbols and their values (sorted by both name and value).
−Mx	Informs the link editor of the nature of the memory model. The model, *x*, may be s (small), m (middle), l (large), h (huge), or e (mixed). The arguments s, m, and l are mutually exclusive.
−N *pagesize*	This option forces the alignment of each segment to *pagesize* (should be a multiple of 512) boundaries within the linker output file. The default is 1024 for 80386 programs. 8086/80186/80286 programs do not normally have page-aligned x.out files and the default for these is 0.
−n *num*	Instructs the link editor to truncate all symbols to a length equal to the specified *num*.

-o *name*	Produce an output object file named *name*. Overrides the default object file name, *a.out*.
-P	Do not pack segments. Normally, the link editor attempts to pack all logical segments that do not have a group association into the same physical segment. This switch disables packing.
-r	Produces a relocatable object module as output.
-Rx *num*	This option is used in conjunction with the **-M3s** option. **-Rd** is used to relocate a data segment specfied by the *num* argument and is added to the final target value of data fixups. **-Rt** is used to relocate text segments. The default for both data and text segments is 0. (This option applies only to the 80386.)
-srelocating	Instructs the link editor to strip the line number entries and the symbol table information from tbe output object file.
-S *num*	Sets the maximum number of segments allowed to *num*, which must be ≤1024. The default maximum is 128.
-u *symname*	Enter *symname* as an undefined link editor symbol in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.
-v *num*	Takes the specified *number* as a decimal version number identifying the **a.out** that is produced. The version stamp is 2, 3, or 5 for the XENIX version and is stored in the system header.

3.4 The Executable Object File

Object files are produced both by the assembler (typically as a result of invoking the eompiler) and by **ld**. **ld** accepts relocatable object files as input and produces an output object file.

Files produced from the compiler/assembler always contain three segments, called _TEXT, _DATA, and _BSS . The _TEXT segment contains the instruction text (e.g. executable instructions), the _DATA segment contains initialized data variables, and the _BSS (blank static storage) segment contains uninitialized data variables. The following program fragment will serve to illustrate:

```
        int i = 100;              /* initialized variable */
        char abc[200];            /* uninitialized variable */

        main()
        {
                abc[i]=0;         /* assignment */
        }
```

Compiled code from the assignment would be stored in _TEXT . The variable **i** would be located in _DATA , and the uninitialized string of characters, **abc** would be located in the _BSS segment.

There is one exception to this rule: both initialized and uninitialized *statics* are placed into the _DATA segment.

3.5 Communal Variable Allocation

A communal variable is an uninitialized global variable. The link editor follows a number of rules in allocation of communal variables. They are as follows:

> If there are multiple communal variables of the same name defined, the link editor chooses the length of the largest definition and allocates that amount of space in the C_COMMON segment.
>
> If there is a definition of the variable that is initialized (a public definition), it takes precedence over all communal definitions and the link editor allocates the length specified by the PUBDEF in the _DATA segment.
>
> If there is more than one public definition the link editor generates an error message saying that the symbol is multiply defined.

The following example illustrates these rules. Suppose that you link the following three modules, containing these global declarations:

```
            A: char headr[512];
            B: char headr[128];
            C: char headr[256];
```

The link editor recognizes all three object modules (*A,B,C*) as containing declarations for *headr* - an uninitialized array. **ld** chooses the definition in module A as the largest of the three and allocates 512 bytes for *headr* in the C_COMMON segment.

Now suppose that the declarations were as follows:

```
A: char headr[512];
B: char headr[128] = "adc";
C: char headr[256];
```

Module *B*'s array has been initialized and, according to the rules followed by **ld**, it takes precedence over all other declarations. 128 bytes is allocated for *headr* in the segment _DATA.

Note that in this case, any subsequent addressing beyond *headr*[127] will have unpredictable results.

The simplest way to avoid these dangers is to put all global declarations in a single header file that is included in all modules that reference them.

3.6 Pointer and Integer Sizes

The following tables define the bit sizes of text and data pointers in each program memory model enabled by the **-M0**, **-M1**, or **-M2** option.

8086/80286 Memory-Model Text and Data Pointers

Model	**Data Pointer**	**Text Pointer**	**Integer**
Small	16	16	16
Medium	16	32	16
Large	32	32	16
Huge	32	32	16

The following table defines the bit sizes of text and data pointers in each program memory model enabled by the **-M3** option.

80386 Memory-Model Text and Data Pointers

Model	**Data Pointer**	**Text Pointer**	**Integer**
Pure-Text Small	32	32	32

The following table lists the default text- and data-segment names, and the default module name for each object file created by the **-M0**, **-M1**, or **-M2** option.

8086/80286 Memory Model Defaults			
Model	**Text**	**Data**	**Module**
Small	_TEXT	_DATA	*filename*
Medium	*module*_TEXT	_DATA	*filename*
Large	*module*_TEXT	_DATA	*filename*
Huge	*module*_TEXT	_DATA	*filename*

The following table lists the default text- and data- segment names and the default module name for each object file created by the **-M3** option.

80386 Memory-Model Defaults			
Model	**Text**	**Data**	**Module**
Pure-Text Small	_TEXT	_DATA	*filename*
Medium	*module*_TEXT	_DATA	*filename*

3.7 Segment and Register Sizes

The following table summarizes the structure of text and data segments for the four possible program memory models enabled by the **-M0**, **-M1**, or **-M2** option.

8086/80286 Memory-Models Summary			
Model	**Text**	**Data**	**Segment Registers**
Small	1[†]	1[†]	**CS=DS=SS**
Medium	1 per module	1	**DS=SS**
Large	1 per module	1	**DS=SS**
Huge	1 per module	1	**DS=SS**

[†] In impure-text small-model programs, text and data occupy the same segment. In pure-text programs, they occupy different segments and the register **CS != DS**.

The following table summarizes the structure of text and data segments for the two possible program memory models enabled by the -M3 option.

80386 Memory-Model Summary			
Model	**Text**	**Data**	**Segment Registers**
Pure-Text Small	1 per module	1	CS!=DS,DS=ES=SS
Mixed	1 per module	1	DS=SS=ES

Chapter 4

adb: A Program Debugger

4.1 Introduction

adb(CP) is a debugging tool for C and assembly language programs. It carefully controls the execution of a program while letting you examine and modify the program's data and text areas.

This chapter explains how to use **adb**. In particular, it explains how to:

Start the debugger
Display program instructions and data
Run, breakpoint, and single-step a program
Patch program files and memory

It also illustrates techniques for debugging C programs, and explains how to display information in non-ASCII data files.

4.2 Starting and Stopping adb

adb provides a powerful set of commands to let you examine, debug, and repair executable binary files as well as examine non-ASCII data files. To use these commands, you must invoke **adb** from a shell command line and specify the file or files you wish to debug. The following sections explain how to start **adb** and describe the types of files available for debugging.

4.2.1 Starting With a Program File

You can debug any executable C or assembly language program file by entering a command line of the following form:

```
adb [filename]
```

where *filename* is the name of the program file to be debugged. **adb** opens the file and prepares its text (instructions) and data for subsequent debugging. For example, the command:

```
adb sample
```

prepares the program named "sample" for examination and execution.

Once started, **adb** normally prompts with an asterisk (*) and waits for you to enter commands. If you have given the name of a file that does not exist or is in the wrong format, **adb** will display an error message first, then wait for commands. For example, if you invoke **adb** with the command:

```
adb sample
```

and the file "sample" does not exist, **adb** displays the message:

```
        adb: cannot open 'sample'
```

You may also start **adb** without a filename. In this case, **adb** searches for the default file, *a.out*, in your current working directory and prepares it for debugging. Thus, the command:

```
        adb
```

is the same as entering:

```
        adb a.out
```

adb displays an error message and waits for a command if the *a.out* file does not exist.

4.2.2 Starting With a Core Image File

adb also lets you examine the core image files of programs that caused fatal system errors. Core image files contain the contents of the CPU registers, stack, and memory areas of the program at the time the error occurred and provide a way to determine the cause of an error.

To examine a core image file with its corresponding program, you must give the name of both the core and and the program file. The command line has the form:

adb *programfile corefile*

where *programfile* is the filename of the program that caused the error, and *corefile* is the filename of the core image file generated by the system. **adb** then uses information from both files to provide responses to your commands.

If you do not give a core image file, **adb** searches for the default core file, named *core*, in your current working directory. If such a file is found, **adb** uses it regardless of whether or not the file belongs to the given program. You can prevent **adb** from opening this file by using the hyphen (–) in place of the core filename. For example, the command:

```
        adb sample –
```

prevents **adb** from searching your current working directory for a core file.

4.2.3 Starting adb With Data Files

You can use **adb** to examine data files by giving the name of the data file in place of the program or core file. For example, to examine a data file named *outdata*, enter:

```
adb outdata
```

adb opens this file and lets you examine its contents.

This method of examining files is very useful if the file contains non-ASCII data. **adb** provides a way to look at the contents of the file in a variety of formats and structures. Note that **adb** may display a warning when you give the name of a non-ASCII data file in place of a program file. This usually happens when the content of the data file is similar to a program file. Like core files, data files cannot be executed.

4.2.4 Starting With the Write Option

You can make changes and corrections in a program or data file using **adb**, if you open it for writing using the **-w** option. For example, the command:

```
adb -wsample
```

opens the program file *sample* for writing. You may then use **adb** commands to examine and modify this file.

Note that the **-w** option causes **adb** to create a given file if it does not already exist. The option also lets you write directly to memory after executing the given program. See the section "Patching Binary Files" later in this chapter.

4.2.5 Starting With the Prompt Option

You can define the prompt used by **adb** by using the **-p** option. The option has the form:

-p *prompt*

where *prompt* is any combination of characters. If you use spaces, enclose the *prompt* in quotes. For example, the command:

```
adb -p "Mar 10->" sample
```

sets the prompt to "Mar 10->". The new prompt takes the place of the default prompt (*) when **adb** begins to prompt for commands.

Make sure there is at least one space between the **-p** and the new prompt, otherwise **adb** will display an error message. Note that **adb** automatically supplies a space at the end of the new prompt, so you do not have to supply one.

4.2.6 Leaving adb

You can stop **adb**, and return to the system shell, by using the **$q** or **$Q** commands. You can also stop the debugger by entering Ctrl-D.

You cannot stop the **adb** command by pressing the INTERRUPT or QUIT keys. These keys are caught by **adb**, and cause it to wait for a new command.

4.3 Displaying Instructions and Data

adb provides several commands for displaying the instructions and data of a given program, and the data of a given data file. The commands have the form:

address [, *count*] = *format*

address [, *count*] ? *format*

address [, *count*] / *format*

where *address* is a value or expression giving the location of the instruction or data item, *count* is an expression giving the number of items to be displayed, and *format* is an expression defining how to display the items. The equal sign (=) and slash (/) tell **adb** from what source to take the item to be displayed. The question mark (?) displays a given address in a given format. With the question mark (?) the *program file* is examined. The slash (/) tells **adb** to examine the *core file*.

The following sections explain how to form addresses, how to choose formats, and the meaning of each of the display commands.

4.3.1 Forming Addresses

In **adb**, every address has the form

[*segment*:] *offset*

where *segment* is an expression giving the address of a specific segment of 8086/286/386 memory, and *offset* is an expression giving an offset from the beginning of the specified segment to the desired item. Segments and offsets are formed by combining numbers, symbols, variables, and operators. The following are some valid addresses:

```
0:1
0x0bce:772
```

The *segment*: is optional. If not given, the most recently typed segment is used.

4.3.2 Forming Expressions

Expressions may contain decimal, octal, and hexadecimal integers, symbols, **adb** variables, register names, and a variety of arithmetic and logical operators.

Decimal, Octal, and Hexadecimal Integers

Decimal integers must begin with a nonzero decimal digit. Octal numbers must begin with a zero and may have octal digits only. Hexadecimal numbers must begin with the prefix "0x" and may contain decimal digits and the letters "a" through "f" (in both upper and lowercase). The following are valid numbers:

Decimal	Octal	Hexadecimal
34	042	0x22
4090	07772	0xffa

Although decimal numbers are displayed with a trailing decimal point (.), you cannot use the decimal point when entering the number.

Symbols

Symbols are the names of global variables and functions defined within the program being debugged, and are equal to the address of the given variable or function. Symbols are stored in the program's symbol table, and are available if the symbol table has not been stripped from the program file (see **strip** (CP)).

In expressions, you may spell the symbol exactly as it is shown in the source program or the symbol table. Symbols in the symbol table are no more than eight characters long, and those defined in C programs are given a leading underscore (_). The following are examples of symbols:

```
main  _main  hex2bin      __out_of
```

Note that if the spelling of any two symbols is the same (except for a leading underscore), **adb** will ignore one of the symbols and allow references only to the other. For example, if both "main" and "_main" exist in a program, then **adb** accesses only the first to appear in the source and ignores the other.

When you use the **?** command, **adb** uses the symbols found in the symbol table of the program file to create symbolic addresses. Thus, the command sometimes gives a function name when displaying data. This does not happen if the **?** command is used for text (instructions) and the **/** command for data. Local variables cannot be addressed.

adb Variables

adb automatically creates a set of its own variables whenever you start the debugger. These variables are set to the addresses and sizes of various parts of the program file as defined below:

b	base address of data segment
d	size of data
m	execution type
s	size of stack
t	size of text

A user can access storage locations by using the **adb** defined variables. The

```
$v
```

request prints these variables.

adb reads the program file to find the values for these variables. If the file does not seem to be a program file, then **adb** leaves the values undefined.

You can use the current value of an **adb** variable in an expression by preceding the variable name with a less than (<) sign. For example, the current value of the base variable "b" is:

```
<b
```

You can create your own variables or change the value of an existing variable by assigning a value to a variable name with the greater than (>) sign. The assignment has the form:

expression > variable-name

where *expression* is the value to be assigned to the variable, and *variable-name* must be a single letter. For example, the assignment:

```
0x2000>b
```

assigns the hexadecimal value "0x2000" to the variable "h".

You can display the value of all currently defined **adb** variables by using the $v command. The command lists the variable names followed by their values in the current format. The command displays any variable whose value is not zero. If a variable also has a nonzero-segment value, the variable's value is displayed as an address; otherwise it is displayed as a number.

Current Address

adb has two special variables that keep track of the last address to be used in a command and the last address to be typed with a command. The . (dot) variable, also called the current address, contains the last address to be used in a command. The " (double quotation mark) variable contains the last address to be typed with a command. The . and " variables are usually the same except when implied commands, such as the newline and caret (^) characters, are used. (These automatically increment and decrement ., but leave " unchanged.)

Both the . and the " may be used in any expression. The less than (<) sign is not required. For example, the command:

```
.=
```

displays the value of the current address and:

```
"=
```

displays the last address to be entered.

Register Names

adb lets you use the current value of the CPU registers in expressions. You can give the value of the register by preceding its name with the less than (<) sign. For example, the value of the "ax" register can be given as:

```
<ax
```

adb recognizes the following register names for the 286:

ax	register a
bx	register b
cx	register c
dx	register d
di	data index
si	stack index
bp	base pointer
fl	status flag
ip	instruction pointer
cs	code segment
ds	data segment
ss	stack segment
es	extra segment
sp	stack pointer

In addition, **adb** recognizes the following register names for the 386:

eax	register eax
ebx	register ebx
ecx	register ecx
edx	register edx
edi	data index
esi	stack index
ebp	base pointer
efl	status flag
eip	instruction pointer
cs	code segment
ds	data segment
ss	stack segment
es	extra segment
fs	extra segment
gs	extra segment
esp	stack pointer

Note that register names may not be used unless **adb** has been started with a *core* file, or the program is currently being run under **adb** control.

Operators

You may combine integers, symbols, variables, and register names with the following operators:

Unary

~	Not
–	Negative
*	Contents oflocation

Binary

+	Addition
–	Subtraction
*	Multiplication
%	Integer division
&	Bitwise AND
|	Bitwise inclusive OR
	Modulo
#	Round up to thenext multiple

Unaryoperators have higher precedence than binary operators. Allbinary operators have the same precedence. Thus, the expression:

```
2*3+4
```

is equalto 10 and:

```
4+2*3
```

is 18.

You can changethe precedence ofthe operations in an expression by using parentheses. For example, the expression:

```
4+(2*3)
```

is equalto 10.

Note that **adb** uses 32 bit arithmetic. This means that values that exceed 2,147,483,647 (decimal) are displayed as negative values.

Note that the unary * operator treats the given address as a pointer. An expression using this operator resolves to the value pointed to by that pointer. Forexample, the expression:

```
*0x1234
```

is equal to the value at theaddress "0x1234", whereas:

```
0x1234
```

is just equal to "0x1234".

4.3.3 Choosing Data Formats

A format is a letter or character that defines how data is to be displayed. The following are the most commonly used formats:

Letter	Format
o	1 word in octal
d	1 word in decimal
D	2 words in decimal
x	1 word in hexadecimal
X	2 words in hexadecimal
u	1 word as an unsigned integer
f	2 words in floating point
F	4 words in floating point
c	1 byte as a character
s	a null terminated character string
i	machine instruction
b	1 byte in octal
a	the current absolute address
A	the current absolute address
n	a newline
r	a blank space
t	a horizontal tab

A format may be used by itself or combined with other formats to present a combination of data in different forms.

The **d**,**o**,**x**, and **u** formats may be used to display **int** type variables; **D** and **X** to display **long** variables or 32-bit values. The **f** and **F** formats may be used to display single and double precision floating point numbers. The **c** format displays **char** type variables, and the **s** format is for arrays of **char** that end with a null character (null terminated strings).

The **i** format displays machine instructions in 8086/286/386 mnemonics. The **b** format displays individual bytes and is useful for display data associated with instructions, or the high or low bytes of registers.

The **a**,**r**, and **n** formats are usually combined with other formats to make the display more readable. For example, the format:

```
ia
```

causes the current address to be displayed after each instruction.

You may precede each format with a count of the number of times you wish it to be repeated. For example the format:

```
4c
```

displays four ASCII characters.

It is possible to combine format requests to provide elaborate displays. For example, the command:

```
<b,-1/4o4^8Cn
```

displays four octal words followed by their ASCII interpretation from the data space of the core image file. In this example, the display starts at the address "<b", the base address of the program's data. The display continues until the end-of-the-file since the negative count "-1" causes an indefinite execution of the command until an error condition, such as the end of the file occurs. The format, "4o" displays the next four words (16-bit values) as octal numbers. The format "4^" then moves the current address back to the beginning of these four words and the "*C" format redisplays them as eight ASCII characters. Finally, "n" sends a newline character to the terminal. The **C** format causes values to be displayed as ASCII characters if they are in the range 32 to 126. If the value is in the range 0 to 31, it is displayed as an "at" sign (@) followed by a lowercase letter. For example, the value 0 is displayed as "@a". The "at" sign itself is displayed as a double at sign "@@".

4.3.4 Using the = Command

The = command displays a given address in a given format. The command is used primarily to display instruction and data addresses in simpler form, or to display the results of arithmetic expressions. For example, the command:

```
main=a
```

displays the absolute address of the symbol "main" (giving the segment and offset) and the command:

```
<b+0x2000=D
```

displays (in decimal) the sum of the variable "b" and the hexadecimal value "0x2000".

If a count is given, the same value is repeated that number of times. For example, the command:

```
main,2=x
```

displays the value of "main" twice.

If no address is given, the current address is used instead. This is the same as the command:

```
        .=
```

If no format is given, the previous format for this command is used. For example, in the following sequence of commands, both "main" and "start" are displayed in hexadecimal form:

```
        main=x
        start=
```

4.3.5 Using the ? and / Commands

You can display the contents of a text or data segment with the ? and / commands. The commands have the form:

[*address*] [, *count*] ? [*format*]

[*address*] [, *count*] / [*format*]

where *address* is an address with the given segment, *count* is the number of items you wish to display, and *format* is the format of the items you wish to display.

The ? command is typically used to display instructions in the text segment. For example, the command:

```
        main,5?ia
```

displays five instructions starting at the address "main", and the address of each instruction displays immediately before it. The command:

```
        main,5?i
```

displays the instructions, with no addresses other than the starting address.

The / command is typically used to check the values of variables in a program, especially variables for which no name exists in the program's symbol table. For example, the command:

```
        <bp-4?x
```

displays the value (in hexadecimal) of a local variable. Local variables are generally at some offset from the address pointed to by the **bp** register.

4.3.6 An Example: Simple Formatting

The following example illustrates how to combine formats in **?** or **/** commands, to display different types of values that are stored together in the same program. The program to be examined has the following souree statements.

```
char  str1[]  ="This is a characterstring";
int   one     =1;
int   number        =456;
long  lnum    =1234;
float fpt     =1.25;
char  str2[]  ="This is the second character string";

main()
{
      one=2;
}
```

The programis compiled and stored in a file named *sample.*

To start the session, enter:

```
adb sample
```

You can display the value of each individual variable by giving its name and correspondingformat in a **/** command. For example, the command:

```
str1/s
```

displays the contents of "str1" as a string

```
_str1:  This is a character string
```

and the command:

```
number/d
```

displays the contents of "number" as a decimal integer

```
_number:      456.
```

You may choose to view a variable in a variety of formats. For example, you can display the **long** variable "lnum" as a 4-byte decimal, octal, and hexadecimal number by entering the following commands:

```
        Inum/D
        _Inum:       1234
        Inum/O
        _Inum:       02322
        Inum/X
        _Inum:       0x4D2
```

You can also examine all variables as a whole. For example, if you wish to see them all in hexadecimal, enter:

```
        str1,5/8x
```

This command displays eight hexadecimal values on a line, and continues for five lines.

Since the data contains a combination of numeric and string values, it is worthwhile to display each value as both a number and a character to see where the actual strings are located. You can do this with one command by entering:

```
        str1,5/4x4^8Cn
```

In this case, the command displays four values in hexadecimal, then the same values as eight ASCII characters. The caret (^) is used four times, immediately before displaying the characters to set the current address back to the starting address for that line.

To make the display easier to read, you can insert a tab between the values and characters, and give an address for each line by entering:

```
        str1,5/4x4^8t8Cna
```

4.4 Debugging Program Execution

adb provides a variety of commands to control the execution of programs being debugged. The following sections explain how to use these commands as well as how to display the contents of memory and registers.

Note that C does not generate statement labels for programs. This means it is not possible to refer to individual C statements when using the debugger. In order to use execution commands effectively, you must be familiar with the instructions generated by the C compiler and how they relate to individual C statements. One useful technique is to create an assembly language listing of your C program before using **adb**, then refer to the listing as you use the debugger. To create an assembly language listing, use the **-S** option of the **cc** command (see Chapter 2 of the *C User's Guide*, "Cc: a C Compiler").

4.4.1 Executing a Program

You can execute a program by using the **:r** or **:R** command. The command has the form:

```
[address][,count]:r [arguments]

[address][,count]:R[ arguments]
```

where *address* gives the address at which to start execution, *count* is the number of breakpoints you wish to skip before one is taken, and *arguments* are the command line arguments, such as filenames and options, that you wish to pass to the program.

If no *address* is given, then the start of the program is used. Thus, to execute the program from the beginning enter:

```
:r
```

If a *count* is given, **adb** will ignore all breakpoints until the given number have been encountered. For example, the command:

```
,5:r
```

causes **adb** to skip the first 5 breakpoints.

If arguments are given, they must be separated by at least one space each. The arguments are passed to the program in the same way the system shell passes command line arguments to a program. You may use the shell redirection symbols if you wish.

The **:R** command passes the command arguments through the shell before starting program execution. This means you can use shell metacharacters in the arguments to refer to multiple files or other input values. The shell expands arguments containing metacharacters before passing them on to the program.

The command is especially useful if the program expects multiple filenames. For example, the command

```
:R [a-z]*.s
```

passes the argument "[a-z]*.s" to the shell where it is expanded to a list of the corresponding filenames before being passed to the program.

The **:r** and **:R** commands remove the contents of all registers and destroy the current stack before starting the program. This kills any previous copy of the program you may have been running.

4.4.2 Setting Breakpoints

You can set a breakpoint in a program by using the **:br** command. Breakpoints cause execution of the program to stop when it reaches the specified address. Control then returns to **adb**. The command has the form:

address [, *count*] :br*command*

where *address* must be a valid instruction address, *count* is a count of the number of times you wish the breakpoint to be skipped before it causes the program to stop, and *command* is the **adb** command you wish to execute when the breakpoint is taken.

Breakpoints are typically set to stop program execution at a specific place in the program, such as the beginning of a function, so that the contents of registers and memory can be examined. For example, the command:

```
main:br
```

sets a breakpoint at the start of the function named "main". The breakpoint is taken just as control enters the function and before the function's stack frame is created.

A breakpoint with a count is typically used within a function, which is called several times during execution of a program, or within the instructions that correspond to a **for** or **while** statement. Such a breakpoint allows the program to continue to execute until the given function or instructions have been executed for the specified number of times. For example, the command:

```
light,5:br
```

sets a breakpoint at the fifth repetition of the function "light". The breakpoint does not stop the function until it has been called at least five times.

Note that no more than 16 breakpoints at a time are allowed.

4.4.3 Displaying Breakpoints

You can display the location and count of each currently defined breakpoint by using the **$b** command. The command displays a list of the breakpoints given by address. If the breakpoint has a count and/or a command, these are given as well.

The **$b** command is useful if you have created several breakpoints in your program.

4.4.4 Continuing Execution

You can continue program execution after it has been stopped by a breakpoint by using the **:co** command. The command has the form:

```
[address] [,count] :co [signal]
```

where *address* is the address of the instruction at which you wish to continue execution, *count* is the number of breakpoints you wish to ignore, and *signal* is the number of the signal to send to the program (see **signal** (S) in the XENIX *Reference*).

If no *address* is given, the program starts at the next instruction after the breakpoint. If a *count* is given, **adb** ignores the first *count* breakpoint.

4.4.5 Stopping a Program with Interrupt and Quit

You can stop program execution at any time by pressing the INTERRUPT (Ctrl-\) or QUIT (DEL) keys. These keys stop the current program and return control to **adb**. The keys are especially useful for programs that have infinite loops or other program errors.

Note that whenever you press the INTERRUPT or QUIT key to stop a program, **adb** automatically saves the signal and passes it to the program, if it was started by using the **:co** command. This is very useful if you wish to test a program that uses these signals as part of its processing.

If you wish to continue program execution, but you do not wish to send the signals, enter:

```
:co 0
```

The command argument "0" prevents a pending signal from being seut to the program.

4.4.6 Single-Stepping a Program

You can single-step a program, i.e., execute it one instruction at a time, by using the **:s** command. The command executes an instruction and returns control to **adb**. The command has the form:

```
[address ] [, count ] :s
```

where *address* must be the address of the instruction you wish to execute, and *count* is the number of times you wish to repeat the command.

If no *address* is given, **adb** uses the current address. If a *count* is given, **adb** continues to execute each successive instruction until *count* instructions have been executed. For example, the command:

```
main,5:s
```

executes the first 5 instructions in the function *main*.

4.4.7 Killing a Program

You can kill the program you are debugging by using the :k command. The command kills the process created for the program and returns control to **adb**. The command is typically used to clear the current contents of the CPU registers and stack and begin the program again.

4.4.8 Deleting Breakpoints

You can delete a breakpoint from a program by using the **:dl** command. The command has the form:

address :dl

where *address* is the address of the breakpoint you wish to delete.

The **:dl** command is typically used to delete breakpoints you no longer wish to use. The following command deletes the breakpoint set at the start of the function "main".

```
main:dl
```

4.4.9 Displaying the C Stack Backtrace

You can trace the path of all active functions by using the **$c** command. The command lists the names of all functions which have been called but have not yet returned control, as well as the address from which each function was called, and the arguments passed to it.

For example, the command:

```
$c
```

displays a backtrace of the C language functions called.

By default, the **$c** command displays all calls. If you wish to display just a few, you must supply a count of the number of calls you wish to see. For example, the command:

```
,25$c
```

displays upto 25 calls in the current call path.

Note that function calls and arguments are put on the stack after the function has been called. If you put breakpoints at the entry point to a function, the function will not appear in the list generated by the **$c** command. You can remedy this problem by placing breakpoints a few instructions into the function.

4.4.10 Displaying CPU Registers

You can display the contents of all CPU registers by using the **$r** command. The command displays the name and contents of each register in the CPU as well as the current value of the program counter, and the instruction at the current address. For the 286, the display has the form:

```
ax    0x0   fi    0x0
bx    0x0   ip    0x0
cx    0x0   cs    0x0
dx    0x0   ds    0x0
di    0x0   ss    0x0
si    0x0   es    0x0
sp    0x0   sp    0x0
0:0:  addb  al,bl
```

For the 386, the display has the form:

```
eax   0x81000       ef     10x246
ebx   0x0           eip    0x142
ecx   0x0           cs     0x3f
edx   0x8           ds     0x47
edi   0x0           es     0x47
esi   0x0           fs     0x47
ebp   0x0           gs     0x47
esp   0x7           fef8ss 0x47
0x3f:0x142:  push   ebp
```

The value of each register is given in the current default format.

4.4.11 Displaying External Variables

You can display the values of all external variables in a program by using the **$e** command. External variables are variables in your program that have global scope, or have been defined outside of any function. This may include variables that have been defined in library routines used by your program.

The $e command is useful whenever you need a list of the names for all available variables, or to quickly summarize their values. The command displays one name on each line with the variable's value (if any) on the same line.

The display has the form:

```
fac:        0
_errno:     0
_end:       0
__sobuf:    0
_obuf:      0
__lastbu:   0406
__sibuf:    0
__stkmax:   0
Iscadr:     02
__iob:      01664
_edata:     0
```

4.4.12 An Example: Tracing Multiple Functions

The following example illustrates how to execute a program under **adb** control. In particular, it shows how to set breakpoints, start the program, and examine registers and memory. The program to be examined has the following source statements.

```
int     fcnt,gcnt,hcnt;
h(x,y)
{
        int hi; register int hr;
        hi=x+1;
        hr=x-y+1;
        hcnt++ ;
        hj:
        f(hr,hi);
}

g(p,q)
{
        int gi; register int gr;
        gi=q-p;
        gr=q-p+1;
        gcnt++ ;
        gj:
        h(gr,gi);
}

f(a,b)
{
        int fi; register int fr;
        fi=a+2*b;
        fr=a+b;
        fcnt++ ;
        fj:
        g(fr,fi);
}

main()
{
        f(1,1);
}
```

The program is compiled and stored in a file named *sample*. To start the session, enter:

```
adb sample
```

This starts **adb** and opens the corresponding program file. There is no core image file.

The first step is to set breakpoints at the beginning of each function. You can do this with the **:br**command. For example, to set a breakpoint at the start of function "f", enter:

```
f:br
```

You can use similar commands for the "g" and "h" functions. Once you have created the breakpoints, you can display their locations by entering:

```
$b
```

This command lists the address, optional count, and optional command associated with each breakpoint. In this case, the command displays:

```
breakpoints
count bkpt          command
1     _f
1     _g
1     _h
```

The next step is to display the first five instructions in the "f" function. Enter:

```
f,5?ia
```

This command displays five instructions, each preceded by its symbolic address. The instructions in 8086/286/386 mnemonics are

```
_f:          push        bp
_f+1.:       mov         bp,sp
_f+3.:       mov         ax,4
_f+6.:       call        near_chkstk
_f+9.:       push        di
_f+10.:
```

You can display five instructions in the "g" function without their addresses by entering:

```
g,5?i
```

In this case, the display is:

```
_g:    push  bp
       mov   bp,sp
       mov   ax,4
       call  near_chkstk
       push  di
```

To begin program execution, enter:

```
:r
```

adb displays the message:

```
sample: running
```

and begins to execute. As soon as **adb** encounters the first breakpoint (at the beginning of the "f" function), it stops execution and displays the message:

```
breakpoint  _f:    push  bp
```

Since execution to this point caused no errors, you can remove the first breakpoint by entering:

```
f:dl
```

and continue the program by entering:

```
:co
```

adb displays the message:

```
sample: running
```

and begins program execution at the next instruction. Execution continues until the next breakpoint, where **adb** displays the following message:

```
breakpoint  _g:    push  bp
```

You can now trace the path of execution by entering:

```
$c
```

The commands show that only three functions are active: "main", "f" and "start".

```
_f(1., 1.)       from _main+22.
_main (1., -588., -584.)     from _start+50.
_start           from start0+5.
```

Although the breakpoint has been set at the start of function "g", it will not be listed in the backtrace until its first few instructions have been executed. To execute these instructions, enter:

```
,5:s
```

adb single-steps the first five instructions. Now you can list the backtrace again. Enter:

```
$c
```

This time, the list shows four active functions:

```
        _g(2., 3.)             from _f+39.
        _f(1., 1.)             from _main+22.
        _main (1., -588., -584.)      from _start+50.
        _start            from start0+5.
```

You can display the contents of the integer variable "fcnt" by entering:

```
        fcnt/d
```

This command displays the value of "fcnt" found in memory. The number should be "1".

You can continue execution of the program and skip the first 10 breakpoints by entering:

```
        ,10:co
```

adb starts the program and then displays the running message again. It does not stop the program until exactly ten breakpoints have been encountered. The message displayed is shown below:

```
        breakpoint   _h:     push   bp
```

To show that these breakpoints have been skipped, you can display the backtrace again, by entering **$c**.

```
        _g(9., 16.)    from _f+39:
        _f(2., 7.)     from _h+36:
        _h (6., 5.)    from _g+38:
        _g (7., 12.)   from _f+39:
        _f (2., 5.)    from _h+36:
        _h (4., 3.)    from _g+38:
        _g (5., 8.)    from _f+39:
        _f (2., 3.)    from _h+36:
        _h (2., 1.)    from _g+38:
        _g(2., 3.)     from _f+39:
        _f(1., 1.)     from _main+22.
        _main(1., -588., -584.)     from _start+50.
        _start()       from start0+5.
```

4.5 Using the adb Memory Maps

adb prepares a set of maps for the text and data segments in your program, and uses these maps to access items that you request for display. The following sections describe how to view these maps, and how they are used to access the text and data segments.

4.5.1 Displaying the Memory Maps

adb interprets these different file formats and provides access to the different segments through a set of maps. To display the maps, enter: **$m** command. The command has the form:

$m [*segment*]

where *segment* is the number of a segment used in the program.

The command displays the maps for all segments in the program using information taken from either the program and core files or directly from memory. In nonshared files, both text (instructions) and data are intermixed. This makes it impossible for **adb** to differentiate data from instructions, as some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In shared text, the instructions are separated from data. The

?*

command accesses the data part of the *a.out* file. This request tells **adb** to use the second part of the map in the *a.out* file. Accessing data in the *core* file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution. In shared files, the corresponding *core* file does not contain the program text.

If you have started **adb** but have not begun program execution, the **$m** command displays the following:

```
Text Segments
Seg#  File Pos      Vir Size      PhysSize      'sample' - File
63.   160.          3712.         2462.

Data Segments
Seg#  File Pos      Vir Size      Phys Size     'sample' - File
71.   160.          3712.         2462.
```

If you have executed the program, the command display has the form

```
Text Segments
Seg#  File Pos      Vir Size      Phys Size     'sample' - memory
63.   160.          3712.         2462.

Data Segments
Seg#  File Pos      Vir Size      PhysSize      'sample' - memory
71.   160.          3712.         2462.
```

where virtual size is the number of bytes the segment occupies in memory. This size is sometimes different than the size of the segment in the file and will often change as you execute the program. This is due to expansion of the stack or allocation of additional memory during program execution. The filenames to the right always name program file. The file position value is ignored.

If you give a segment number with the command, **adb** displays information only about that segment. For example, the command

```
$m 63
```

displays a map for segment 63 only. The display has the form

```
Segment#= 63.
Type=Text
File position= 160.
Virtual Size= 3712.
Physical Size= 2048.
```

4.5.2 Changing the Memory Map

You can change the values of a memory map by using the **?m** and **/m** commands. These commands assign specified values to the corresponding map entries. The commands have the form

?m *segment-number file-position size*

and

/m *segment-number file-position size*

where *segment-number* gives the number of the segment map you wish to change, *file-position* gives the offset in the file to the beginning of the given address, and *size* gives the segment size in bytes. The **?m** assigns values to a text segment entry; **/m** to a data segment entry.

For example, the following command changes the file position for segment 63 in the text map to 0x2000:

```
?m 63 0x2000
```

The command

```
/m 39 0x0
```

changes the file position for segment 39 in the data map to 0.

4.5.3 Creating New Map Entries

You can create new segment maps and add them to your memory map by using the ?M and /M commands. Unlike ?m and /m, these commands create a new map instead of changing an existing one. These commands have the form

?M *segment-number file-position size*

and

/M *segment-number file-position size*

where *segment-number* gives the number of the segment map you wish to create, *file-position* gives the offset in the file to the beginning of the given address, and *size* gives the segment size in bytes. The ?M command creates a text segment entry; /M creates a data segment entry. The segment number must be unique. You cannot create a new map entry that has the same number as an existing one.

The ?M and /M commands are especially useful if you wish to access segments that are otherwise allocated to your program. For example, the command

```
?M7102504
```

creates a text segment entry for segment "71" whose size is "2504" bytes.

4.5.4 Validating Addresses

Whenever you use an address in a command, **adb** checks the address to make sure it is valid. **adb** uses the segment number, file position, and size values in each map entry to validate the addresses. If an address is correct, **adb** carries out the command; otherwise, it displays an error message.

The first step **adb** takes when validating an address is to check the segment value to make sure it belongs to the appropriate map. Segments used with the **?** command must appear in the text segments map; segments used with the **/** command must appear in the data segments map. If the value does not belong to the map, **adb** displays a bad segment error.

The next step is to check the offset to see if it is in range. The offset must be within the range

```
0 <= offset <= segment-size
```

If it is not in this range, **adb** displays a bad address error.

If **adb** is currently accessing memory, the validating segment and offset are used to access a memory location and no other processing takes place. If **adb** is accessing files, it computes an effective file address

```
effective-file-address = offset + file-position
```

then uses this effective address to read from the corresponding file.

4.6 Miscellaneous Features

The following sections explain how to use a number of useful commands and features of **adb**.

4.6.1 Combining Commands on a Single Line

You can give more than one command on a line by separating the commands with a semicolon (;). The commands are performed one at a time, starting at the left. Changes to the current address and format are carried to the next command. If an error occurs, the remaining commands are ignored.

One typical combination is to place a ? command after a l command. For example, the commands:

```
?l'Th'; ?s
```

search for and display a string that begins with the characters "Th".

4.6.2 Creating adb Scripts

You can direct **adb** to read commands from a text file instead of the keyboard by redirecting **adb**'s standard input file at invocation. To redirect the standard input, use the standard redirection symbol < and supply a filename. For example, to read commands from the file *script*, enter:

```
adb sample <script
```

The file you supply must contain valid **adb** commands. Such files are called script files, and can be used with any invocation of the debugger.

Reading commands from a script file is very convenient when you wish to use the same set of commands on several different object files. Scripts are typically used to display the contents of core files after a program error. For example, a file containing the following commands can be used to display most of the relevant information about a program error:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8xna
```

4.6.3 Setting Output Width

You can set the maximum width (in characters) of each line of output created by **adb** by using the **$w** command. The command has the form:

n$w

where *n* is an integer number giving the width in characters of the display. You may give any width that is convenient for your given terminal or display device. The default width, when **adb** is first invoked, is 80 characters.

The command is typically used when redirecting output to a lineprinter or special terminal. For example, the command:

```
120$w
```

sets the display width to 120 characters, a common maximum width for lineprinters.

4.6.4 Setting the Maximum Offset

adb normally displays memory and file addresses as the sum of a symbol and an offset. This helps associate the instructions and data you are viewing with a given function or variable. When first invoked, **adb** sets the maximum offset to 255. This means instructions or data that are no more than 255 bytes from the start of the function or variable are given symbolic addresses. Instructions or data beyond this point are given numeric addresses.

In many programs, the size of a function or variable is actually larger than 255 bytes. For this reason, **adb** lets you change the maximum offset to accommodate larger programs. You can change the maximum offset by using the **$s** command. The command has the form:

```
n$s
```

where *n* is an integer giving the new offset. For example, the command:

```
4095$s
```

increases the maximum possible offset to 4095. All instructions and data that are no more than 4095 bytes away are given symbolic addresses.

Note that you can disable all symbolic addressing by setting the maximum offset to zero. All addresses will be given numeric values instead.

4.6.5 Setting Default Input Format

You can set the default format for numbers used in commands with the **$d** (decimal), **$o** (octal), and **$x** (hexadecimal) commands. The default format tells **adb** how to interpret numbers that do not begin with "0" or "0x", and how to display numbers when no specific format is given.

The commands are useful if you wish to work with a combination of decimal, octal, and hexadecimal numbers. For example, if you enter:

```
$x
```

you may give addresses in hexadecimal without prepending each address with "0x". Furthermore, **adb** displays all numbers in hexadecimal except those that are specifically requested to be in some other format.

When you first start **adb**, the default format is decimal. You may change this at any time and restore it as necessary using the **$d** command.

4.6.6 Using XENIX Commands

You can execute XENIX commands without leaving **adb** by using the **adb** escape command **!**. The escape command has the form:

```
!command
```

where *command* is the XENIX command you wish to execute. The command must have any required arguments. **adb** passes this command to the system shell which executes it. When finished, the shell returns control to **adb**.

For example, to display the date, enter:

```
!date
```

The system displays the date at your terminal and restores control to **adb**.

4.6.7 Computing Numbers and Displaying Text

You can perform arithmetic calculations while in **adb** by using the = command. The command directs **adb** to display the value of an expression in a given format.

The command is often used to convert numbers in one base to another, to double check the arithmetic performed by a program, and to display complex addresses in easier form. For example, the command:

```
0x2a=d
```

displays the hexadecimal number "0x2a" as the decimal number 42, but:

```
0x2a=c
```

displays it as the ASCII character "*". Expressions in a command may have any combination of symbols and operators. For example, the command:

```
<ax-12*<bx1+<b+5=X
```

computes a value using the contents of the ax and bx registers and the **adb** variable "b". You may also compute the value of external symbols, by entering:

```
main+5=X
```

This is helpful if you wish to check the hexadecimal value of an external symbol address.

Note that the = command can also be used to display literal strings at your terminal. This is especially useful in **adb** scripts where you may wish to display comments about the script as it performs its commands. For example, the command:

```
=3n"C Stack Backtrace"
```

spaces three lines, then prints the message "C Stack Backtrace" on the terminal.

4.6.8 An Example: Directory and Inode Dumps

This example illustrates how to create **adb** scripts to display the contents of a directory file and the inode map of a XENIX file system. The directory file is assumed to be named *dir*, and contains a variety of files. The XENIX file system is assumed to be associated with the device file */dev/src*, and has the necessary permissions to be read by the user.

To display a directory file, you must create an appropriate script, then start **adb** with the name of the directory, redirecting its input to the script.

First, you can create a script file named *script*. A directory file normally contains one or more entries. Each entry consists of an unsigned "inumber" and a 14 character filename. You can display this information by adding the command:

```
0,-1?ut14cn
```

to the script file. This command displays one entry for each line, separating the number and filename with a tab. The display continues to the end of the file. If you place the command:

```
="inumber"8t"Name"
```

at the beginning of the script, **adb** will display the strings as headings for each column of numbers.

Once you have the script file, enter:

```
adb dir- <script
```

(The hyphen (-) is used to prevent **adb** from attempting to open a core file.) **adb** reads the commands from the script and displays the following:

```
inumber    name
652        .
82         ..
5971       cap.c
5323       cap
0          pp
```

To display the inode table of a file system, you must create a new script, then start **adb** with the filename of the device associated with the file system (e.g., the hard disk drive).

The inode table of a file system has a very complex structure. Each entry contains: a word value for the file's status flags; a byte value for the number links; two byte values for the user and group IDs; a byte and word value for the size; eight word values for the location on disk of the file's blocks; and two word values for the creation and modification dates. The inode table starts at address "02000". You can display the first entry by entering:

```
02000,-1?on3bnbrdn8un2Y2na
```

Several newlines are inserted within the display to make it easier to read.

To use the script on the inode table of */dev/src*, enter:

```
adb /dev/src - <script
```

(Again, the hyphen (-) is used to prevent an unwanted core file.) Each entry in the display has the form:

```
02000: 073145
    0163   0164   0141
    0162   10356
    28770  8236   25956  27766  25455  8236   25956  25206
    1976 Feb 5 08:34:56   1975 Dec 28 10:55:15
```

4.7 Patching Binary Files

You can make corrections or changes to any file, including executable binary files, by using the **w** and **W** commands, and invoking **adb** with the **-w** option. The following sections describe how to locate and change values in a file.

4.7.1 Locating Values in a File

You can locate specific values within a file by using the **l** and **L** commands. The commands have the form:

[*address*] ?l *value*

where *address* is the address at which to start the search, and *value* is the value (given as an expression) to be located. The **l** command searches for 2 byte values; **L** for 4 bytes.

The following command:

```
?l
```

starts the search at the current address, and continues until the first match or the end of the file. If the value is found, the current address is set to that value's address. For example, the command:

```
?l'Th'
```

searches for the first occurrence of the string value "Th". If the value is found at "main+210" the current address is set to that address.

4.7.2 Writing to a File

You can write to a file by using the **w** and **W** commands. The commands have the form:

> [*address*] ?w*value*

where *address* is the address of the value you wish to change, and *value* is the new value. The **w** command writes 2 byte values; **W** writes 4 bytes. For example, the following commands change the word "This" to "The ".

```
?l'Th'
?W'The'
```

Note that **W** is used to change all four characters.

4.7.3 Making Changes to Memory

You can also make changes to memory whenever a program has been executed. If you have used an **:r** command with a breakpoint to start program execution, subsequent **w** commands cause **adb** to write to the program in memory rather than the file. This is useful if you wish to make changes to a program's data as it runs, for example, to temporarily change the value of program flags or constants.

Replace this Page
with Tab Marked:

Index

Index

06-21-87

SCO-514-210-014

Index

Chapter 5

C Language Compatibility with Assembly Language

5.1 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.

5.2 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.

5.3 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.

5.4 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.

5.5 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:

```
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.

5.6 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 a 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 arguments 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.

5.7 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 double-word (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.

5.8 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** 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)".

5.9 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
```

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 sequence for passing and returning C language structure returns:

```
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
    repmovsd
    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
    repmovsd
    pop    ebx
    pop    esi
    pop    edi
    leave
    ret
```

5.10 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.

5.11 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:

```
_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  <jvalue>
push  <ivalue>
call  _add
add   esp,8
```

Chapter 6

Error Processing

6.1 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.

6.2 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).

6.3 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 Appendix B 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.

6.4 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*.

6.5 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 Chapter 7 of the XENIX *Programmer's Guide*, "Using Signals," 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.

For details about how to catch, redefine, and restore these signals, see Chapter 7 of the XENIX *Programmer's Guide*.

6.6 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.

Chapter 7

Object and Executable File Formats

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 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 respective Programmer's 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.

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 indicating 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.

Having selected a descriptor table entry, 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. 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

7.4 Definition of Terms

The following terms are fundamental to the 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:

1. 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.

2. 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 (i.e. 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 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:

1. They play a role at LINK time in determining which LSEGs are combined with other LSEGs.

2. They are used in assembly source code to specify groups.

CLASSNAME
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 name. 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.

COMPLETENAME
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 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, etc.) is maintained. The LSEG information is required when multiple LSEGs are combined and when segment addressability (See "Segment Addressing") is established. 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 possible number of LSEGs that 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 allow 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."

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

ld 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. (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 (i.e., "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:

1. The place and type of a LOCATION to be fixed up.

2. One of two possible fixup MODEs.

3. A TARGET, which is a memory address to which LOCATION must refer.

4. 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.) **ld** does not require the presence of the high- or low-order compliment of these items (e.g. in the case of HIBYTE, a high-order word, it doesn't matter if the low-order word is present).

1. A BASE is the high-order word of a pointer.

2. An OFFSET is the low-order word of a pointer.

3. A HIBYTE is the high-order half of an OFFSET.

4. A LOBYTE is the low-order half of an OFFSET.

```
          +----+----+----+----+
Pointer:  |                   |
          +----+----+----+----+

                    +----+----+
Base:               |         |
                    +----+----+

          +----+----+
Offset:   |         |
          +----+----+

               +----+
Hibyte:        |    |
               +----+

          +----+
Lobyte:   |    |
          +----+
```

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 to be 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 (i.e., 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.

(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(<groupname>), <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 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:

(F0) X is a SEGMENT INDEX. The FRAME is the canonic FRAME of the LSEG defined by the INDEX.

(F1) X is a GROUP INDEX. The FRAME is the canonic FRAME defined by the group (i.e., the canonic FRAME defined by the LSEG in the group that is eventually LOCATEd lowest in MAS).

(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.)

(F3) X is a FRAME NUMBER (specifying the obvious FRAME).

(F4) No X. The FRAME is the canonic FRAME of the LSEG containing LOCATION.

(F5) No X. The FRAME is determined by the TARGET. There are four cases:

- (F5a) The TARGET specified a SEGMENT INDEX: in this case, the FRAME is determined as in (F0).
- (F5b) The TARGET specified a GROUP INDEX: in this case, the FRAME is determined as in (F1).
- (F5c) The TARGET specified 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 linkeditordoesnotsupportframemethods F2b, F3, and F5d.

Nomenclature describing FRAMEs is similar to the above nomenclature for TARGETs.

FRAME:	SI(<segmentname>)	[F0]
FRAME:	GI(<groupname>)	[F1]
FRAME:	EI(<symbolname>)	[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 a LOBYTE. (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.

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 (i.e. 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:

- 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.

- 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. (i.e. (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:

- If the LOCATION is a BASE or a POINTER, then FBVAL is stored in the BASE component of the LOCATION.

- 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.

- If the LOCATION is a HIBYTE then (FOVAL/256) is added to the LOCATION ignoring overflow.

7.13 Record Order

A 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.

---

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:

1. A FIXUPP record always refers to the previous DATA record.

2. All LNAMES, SEGDEF, GRPDEF, TYPDEF, and EXTDEF records must precede all records that refer to them.

3. COMENT 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   |   RECORD   |     NAME      |    NUMBER     |  CHK   |
|  TYP   |   LENGTH   |               |               |  SUM   |
|  xxH   |            |               |               |        |
|        |            |               |               |        |
-------------------------------///-----------||||------------
                      |               |
                      +-----rpt-------+
```

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 boxes with three slashes in the top and bottom represent a variable number of bytes, one or more, depending upon content. The wide boxes with four vertical bars in the top and bottom 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.

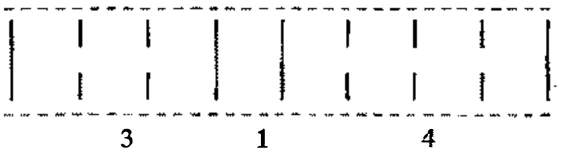
Similarly, some portions of a record format are present only if some given condition is true; these fields are indicated by similar "conditional" or "cond" brackets below the boxes.

7.14.8 Chksum

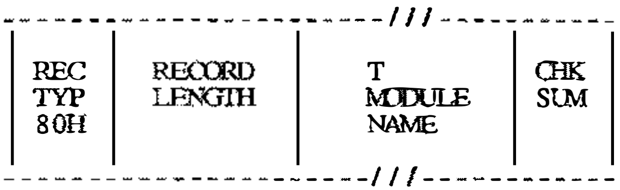
The last field in each record is a check sum, which contains the 2's complement of the sum (modulo 256) of all other bytes in the record. Therefore, the sum (modulo 256) of all bytes in the record equals 0.

7.14.9 Bit Fields

Descriptions of contents of fields will sometimes be at the bit level. Boxes with vertical lines drawn through them represent bytes or words; the 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)

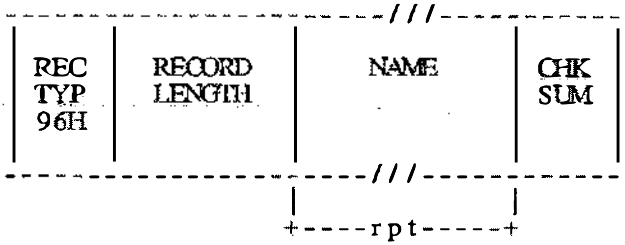


Every module output from a translator must have a T-MODULE HEADER RECORD.

7.14.10 T-Module Name

The T-MODULE NAME provides a name for the T-MODULE.

LIST OF NAMES RECORD
(LNAMES)



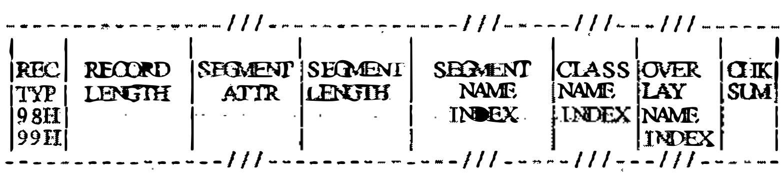
This Record provides a list of names that may be used in following SEGDEF and GRPDEF records as the names of Segments, Classes and/or Groups.

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)



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.

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:

```
----------------------------------------
|  ACB   |  FRAME     |   OFF         |
|   P    |  NUMBER    |   SET         |
|        |            |               |
----------------------------------------
         |                            |
         +----conditional-----------+
```

The ACBP byte contains four numbers which are the A, C, B, and P attribute specifications. This byte has the following format:

```
---------------------------------------------------------
|      |   A   |      |   C   |      |  B  |  P  |
|      |       |      |       |      |     |     |
---------------------------------------------------------
```

"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<=dx<LX) be the offset in X the (combined) LSEG Z; let dx (0<=dx<LX) be the offset in X of a byte, and let dy similarly 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.

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 SegmentLength

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 SegmentNameIndex

The Segment Name is a name the programmer or translator assigns to the segment. Examples: CODE, DATA, TAXDATA, MODULENAME_CODE, STACK. This field provides the Segment Name, by indexing into the list of names provided by the LNAMES Record(s).

7.14.15 Class NameIndex

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 LNAMES 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 LNAMES 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  |  RECORD  |    GROUP    |    GROUP    |  CHK |
| TYP  |  LENGTH  |    NAME     |  COMPONENT  |  SUM |
| 9AH  |          |    INDEX    |  DESCRIPTOR |      |
-------------------------///-----------///------------------
                                |             |
                                +--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 LNAMES Record(s).

7.14.18 Group Component Descriptor

Each GROUP COMPONENT DESCRIPTOR has the following format:

```
------------///-----
|  SI  |  SEGMENT  |
|      |   INDEX   |
|(FFH) |           |
|      |           |
--------------------
```

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 | RECORD |    NAME     |    EIGHT     |  CHK  |
| TYP | LENGTH |  (USUALLY   |     LEAF     |  SUM  |
| 8EH |        |    NULL)    |  DESCRIPTOR  |       |
|     |        |             |              |       |
---------------------------///----------///--------------
                             |              |
                             +--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).

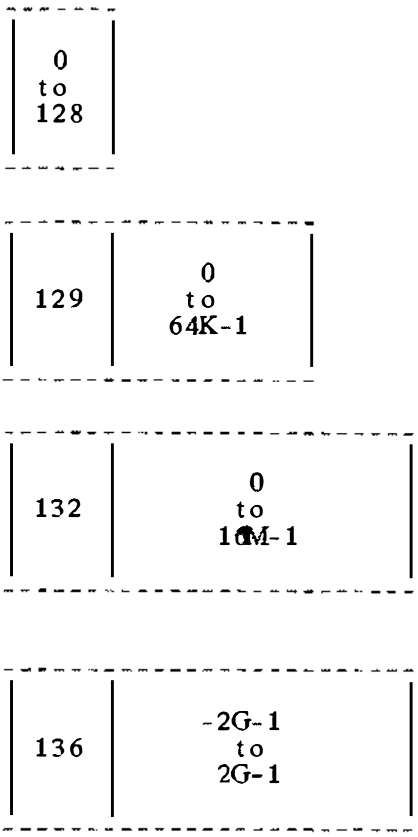
7.14.20 EightLeafDescriptor

This field can describe up to eight Leaves.

```
------------///------
|  E  |    LEAF     |
|  N  | DESCRIPTOR  |
|     |             |
------------///------
      |             |
      +----rpt------+
```

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 formats given on the next page.



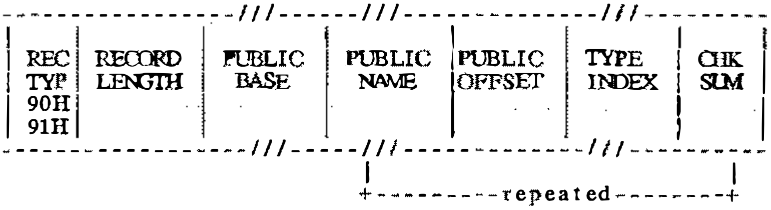
The first format (single byte), containing a value between 0 and 127, represents a Numeric Leaf whose value is the number given.

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.

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.

PUBLICNAMES DEFINITION RECORD
(PUBDEF)

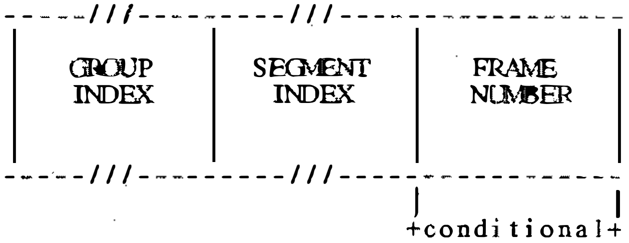


This record provides a list of one or more PUBLIC NAMEs; for each one, three data are provided: (1) a base value for the name, (2) the offset value of the name, and (3) the type of entity represented by the name.

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 Section 7.10, "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 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.

7.14.22 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)

```
-----------------------------///-----------///-----------
| REC  | RECORD     | EXTERNAL      | TYPE          | CHK  |
| TYP  | LENGTH     | NAME          | INDEX         | SUM  |
| 8CH  |            |               |               |      |
|      |            |               |               |      |
-----------------------------///-----------///-----------
                    |                               |
                    +----------repeated-------------+
```

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 kth 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)

```
------------------------///---------------------------------------------
|  REC  |  RECORD  |   LINE   |     LINE     |    LINE    |  CHK  |
|  TYP  |  LENGTH  |  NUMBER  |    NUMBER    |   NUMBER   |  SUM  |
|  94H  |          |   BASE   |              |   OFFSET   |       |
|  95H  |          |          |              |            |       |
------------------------///---------------------------------------------
                              |                           |
                              +---------repeated----------+
```

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:

```
-----///----------///-----
|   GROUP    |  SEGMENT   |
|   INDEX    |   INDEX    |
| (ignored)  |            |
-----///----------///-----
```

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)

```
-------------------------///----------------------------------
| REC  | RECORD | SEGMENT | ENUMERATED |      | CHK |
| TYP  | LENGTH |  INDEX  |    DATA    | DAT  | SUM |
| A0H  |        |         |   OFFSET   |      |     |
| A1H  |        |         |            |      |     |
-------------------------///----------------------------------
                                       |      |
                                       +-rpt-+
```

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.

7.14.32 Dat

This field provides up to 1024 consecutive bytes of relocatable or absolute data.

LOGICAL ITERATED DATA RECORD
(LIDATA)

```
----------------------///--------------------///-----------
|  REC  |  RECORD  |  SEGMENT |  ITERATED  |  ITERATED  | CHK |
|  TYP  |  LENGTH  |  INDEX   |    DATA    |    DATA    | SUM |
|  A2H  |          |          |   OFFSET   |   BLOCK    |     |
|  A3H  |          |          |            |            |     |
----------------------///--------------------///-----------
                                             |            |
                                             +-repeated-+
```

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

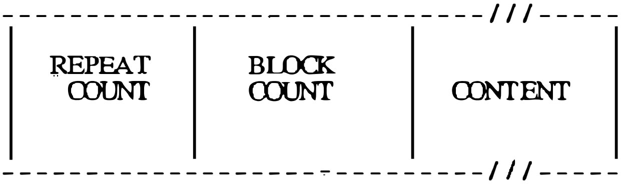
This field must be non-zero and specifies an index relative to the SEGDEF records found previous to the LIDATA RECORD.

7.14.34 Iterated Data Offset

This field specifies either a 16-bit offset (Rectype=A2H) or a 32-bit offset (Rectyp=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:



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.

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)

```
 -------------------------------------///-----------------
|  REC   |  RECORD      |    THREAD      |   CHK  |
|  TYP   |  LENGTH      |      or        |   SUM  |
|  9CH   |              |    FIXUP       |        |
|  9DH   |              |                |        |
 -------------------------------------///-----------------
                        |                |
                        +-----rpt--------+
```

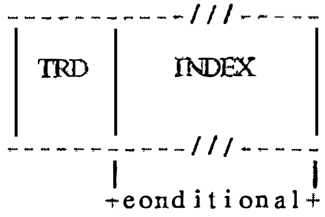
This record specifies 0 or more fixups. Each fixup requests a modification (fixup) to a LOCATION within the previous DATA record. A data record may be followed by more than one fixup record that refers. Each fixup is specified by a FIXUP field that specifies four data: a location, a mode, a target and 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).

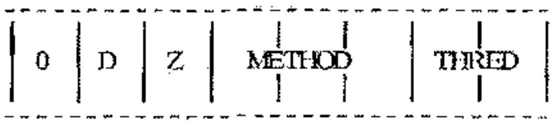
In the RECORD TYPE field, 9CH and 9DH describe 16- and 32-bit FIXUPP records, respectively.

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.

7.14.40 Fixup

FIXUP is a field with the following format:

```
------------------------------///-----------///----------///-----
|                |      |            |            |  TARGET     |
|      LOCAT     | FIX  |   FRAME    |   TARGET   |   DIS-      |
|                | DAT  |   DATUM    |   DATUM    |  PLACEMENT  |
|                |      |            |            |             |
------------------------------///-----------///----------///-----
                        |            |            |             |
                        +conditional+conditional+conditional+
```

LOCAT is a byte pair with the following format:

```
-------------------------------------------------------------------------
| 1 | M |   |  |  |  |  |  |  |  |  |  |  |  |  |
|   |   |   L  O  C  | D A T A  R E C O R D  O F F S E T |
|   |   |   |  |  |  |  |  |  |  |  |  |  |  |  |
-------------------------------------------------------------------------
|                                |                                      |
+-----------lo byte--------------+-----------hi byte-------------------+
```

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	-	8bit	lobyte
1	-	16bit	offset
2	-	16bit	base
3	-	32bit	pointer
4	-	8bit	hibyte
5	-	16bit	offset (linker resolved)
9	-	32bit	offset
11	-	48bit	pointer
13	-	32bit	offset (linker resolved)

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.

FIXDAT is a byte with the following format:

F	FRAME	T	P	TARGT

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.)

"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.

TARGT 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 TARGT 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)

REC TYP 8AH 8BH	RECORD LENGTH	MOD TYP	START ADDRS	CHK SUM

+conditional+

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 areas follows:

MATTR	Z	Z	Z	Z	Z	L

MATTR is a 2-bit subfield that specifies the following module attributes:

MATTR MODULE ATTRIBUTE

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.

"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

```
-------------///-----------///------------------
|  END   |   FRAME     |  TARGET     |  TARGET   |
|  DAT   |   DATUM     |   DATUM     |   DIS-    |
|        |             |             | PLACEMENT |
|        |             |             |           |
-------------///-----------///------------------
         |             |             |           |
         +conditional--+conditional--+conditional+
```

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)

REC TYP 88H	RECORD LENGTH	COMMENT TYPE	COMMENT	CHK SUM

This record allows translators to include comments in object text.

7.14.42 CommentType

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:

N P	N L	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 COMENT 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 Microsoft extensions to OMF, such as the various 32-bit record types.

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 Microsoft 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 | RECORD |   |     EIGHT     | CHK |
| TYP | LENGTH | 0 |     LEAF      | SUM |
| 8EH |        |   |  DESCRIPTOR   |     |
------------------------------///-----------
```

The EIGHT LEAF DESCRIPTOR field has the following format:

```
----------///------
| E |    LEAF    |
| N | DESCRIPTOR |
----------///------
```

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:

```
-------------------------///-------///------
| NEAR  | VAR  |   LENGTH    |    VAR      |
| 62H   | TYP  |     IN      |   SUBTYP    |
|       |      |    BITS     |             |
-------------------------///-------///------
                             |             |
                             +-------------+
                               (optional)
```

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:

```
-------------------------///----------///-----
| FAR  | VAR  |   NUMBER    |  ELEMENT     |
| 61H  | TYP  |     OF      |   TYPE       |
|      | 77H  |  ELEMENTS   |   INDEX      |
-------------------------///----------///-----
```

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 Chapter 4, "ld: the Link Editor") will link the Intel 86 Relocatable Object Format into the executable format described in this section.

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.

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 {                                   /* 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(infile) */
        long          xs_vsize;                 /* virtual size (in core) */
        long          xs_rbase;                 /* relocation base address */
        long          xs_lres;                  /*unused */
        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 he 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.

Under XENIX, areas of memory that are initialized by more than one iterationrecord 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.

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 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.

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 */
};
struct xext {
        /* x.out header extension */
        long             xe_trsize;
                /* size of text relocation */
        long             xe_drsize;
                /* size of data relocation */
                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_mdttype;
                /* 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 (setup 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
 */

#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_SEG          0x0800
        /* segment table present */
#define XE_ABS          0x0400
        /* absolute memory image (standalone) */
#define XE_ITER         0x0200
        /* iterated text/data present */
#define XE_HDATA        0x0100
        /* huge model data */
#define XE_FPH          0x0080
        /* 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_SEP          0x0002
        /* 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_mdttype
 */
#define XE_MDTNONE      0
        /* no machine dependent table */
#define XE_MDT286       1
        /* iAPX286 LDT */

/*
 *      Definitions for xe_ostype
 */
#define XE_OSNONE  0
#define XE_OSXENIX 1
        /* XENIX */
#define XE_OSRMX   2
        /* iRMX */

/*
 *      Definitions for xe_osvers
 */
#define XE_OSXV3    1
        /* XENIX */
```

```
/*
 *      Definitions for xs_type:
 *       Values from 64 to 127 are reserved.
 */
#define XS_TNULL     0        /* unused segment */
#define XS_TTEXT     1        /* text segment */
#define XS_TDATA     2        /* data segment */
#define XS_TSYMS     3        /* symbol table segment */
#define XS_TREL      4               /* relocation segment */

/*
 *      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_AITER     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

Writing Device Drivers

8.1 Introduction

This chapter, along with Chapter 9, "Sample Device Drivers," explains how to write and install device drivers in a XENIX environment. It describes the role of device drivers in a XENIX-based system and discusses other special considerations involved in writing a device driver. It describes the XENIX model of devices in terms of files, tasks to be performed, and interrupts to be processed.

8.1.1 What is a XENIX Device Driver?

For each peripheral device in a XENIX system, there must be a "device driver" to provide the software interface between the device and the system. A XENIX device driver is a set of routines that communicates with a hardware device, and provides a uniform interface to the kernel. This interface allows the kernel to translate user I/O requests into driver tasks to be performed.

8.1.2 Relationship to XENIX Operating System

The XENIX device driver manages the flow of data and control between the user program and the peripheral devices. The path of an I/O request is shown below, starting with a system call from a user program, and ending at the device driver:

```
+--------------+
| User Program |
+--------------+
       |
       |          User Space
---------------|-------------
       |          Kernel Space
       |
+------|----------+-------+
|      |          |       |      +-----------+
|      +--->-------->-------->   | Peripheral |
|  Kernel         |Device |      | Devices    |
|                 |Drivers|      +-----------+
+-------------------------+
```

User Program Requesting I/O

8.1.3 Device Models Supported by XENIX

The XENIX operating system supports two device models: character devices and block devices. This chapter describes how to write device drivers for both device models.

In general, any device that appears to be a randomly addressable set of fixed-size records is a block device; any other type of device is a character device. For example, hard disk drives and floppy disks are block devices, while terminals and line printers are character devices. The XENIX operating system presents a uniform interface to user programs by providing blocking and unblocking in the kernel. Thus, character and block devices look alike to most user programs.

Character device drivers communicate directly with the user program. The process begins when a user program requests a data transfer of some number of bytes between a section of its memory and a specific device. The operating system transfers control to the appropriate device driver. The user program supplies the parameters for the request to the device driver, which, in turn, performs the work. Thus, the operating system has minimal involvement in the request; the data transfer is a private transaction between the user process and the device driver.

Block device drivers require more involvement from the operating system to perform their tasks. Block devices transfer data in fixed-size blocks and are usually capable of random access. (The device does not need to be capable of random access; magnetic tapes are often read or written using block I/O.) The two factors that distinguish block I/O from character I/O are:

- The size of data transfer requests from the kernel to the device is always a multiple of the system block size (called BSIZE) regardless of the size of the user process' original request. A single user process request can generate many system requests to the driver. BSIZE is 1024 bytes in XENIX System V. The device's physical block size may be smaller than BSIZE, in which case the device driver initiates multiple physical transfers to move a single logical block.

- Transfers are never done directly into a user process' memory area. They are always staged through a pool of BSIZE buffers. Program I/O requests are satisfied directly from the buffers. XENIX commands the device driver to read and write from the buffers as necessary. It manages these buffers to perform services such as blocking and unblocking of data and disk caching.

8.1.4 Using Sample Device Drivers

Chapter 9, "Sample Device Drivers," discusses sample device driver source code for a line printer, terminal, hard disk drive and memory-mapped video driver. These source code samples are intended as prototypes from which an experienced programmer can begin writing a device driver for a particular device.

8.1.5 Special Device Files

To a XENIX user, a device usually appears to act like a "file," which is an ordered sequence of bytes. Files that contain data are called "regular files," and files that represent devices are called "special device files." Each file has at least one name. The names of special device files are, by convention, placed in the directory named */dev*.

Each special device file has a "device number" that uniquely identifies the device. The device number consists of two parts, the "major number" and the "minor number." The major number tells the kernel which device driver handles requests for that special file. The minor number can be used by the driver to provide more information about a particular unit of the devices that it controls (such as the unit number).

Before a user process can request I/O, it must first have opened a "special device file." A special device file looks like an ordinary disk file except that it was created by the **mknod**(S) system call, normally invoked using the **mknod**(C) program, both of which are described in the XENIX *Reference*. The special file appears in a directory and has owner and permission fields, as does any disk file, but it contains no data. Instead, it has a pair of eight-bit numbers, called the "major" and "minor" numbers, associated with it. The command **ls -l** displays these numbers:

```
crw--w--w- 1 dave     4,  3 Sep 21 09:49 /dev/tty03
brw------- 1 sysinfo  7,  2 Sep 21 09:49 /dev/mt0
```

Here the file */dev/tty03* has a major device number of four and a minor device number of three. The */dev/mt0* file (magnetic tape) has a major device number of seven and a minor device number of two.

When a user process opens the special device file, XENIX recognizes that it is a special device file and uses the major number to index a table of device driver entry points. If the special device file designates a character device, the table used is *cdevsw*[ ]; if it designates a block device, the table used is *bdevsw*[ ]. These two tables are defined in the */usr/sys/conf/c.c* file generated by the **config**(C) program when the kernel is built. XENIX calls the device driver's open entry-point through this table, supplying as an argument the minor device number. The minor device number usually encodes the unit number, although often a device driver uses some of the bits in the minor number to indicate special options, such as "use double density" in the case of a floppy disk.

The convention is for these special device files to have meaningful names and reside in the */dev* directory. For example:

/dev/tty03

would normally be associated with the major device number of the serial device driver and its minor number would indicate the fourth port. Or:

```
/dev/mt0
```

indicates the block magnetic tape device and:

```
/dev/rmt0
```

indicates the *raw* magnetic tape device.

It is important to note that this is just a convention. The system administrator could just as well assign the same major/minor numbers to either of the two files:

```
/usr/ellen/tty03
```

or:

```
/usr/ellen/magtape
```

with identical results. The name is primarily for user convenience as XENIX kernel uses solely the major and minor device numbers.

8.2 Kernel Environment

This section briefly discusses a few functional aspects of the XENIX operating system:

- modes of operation
- context switching
- system mode stack use
- task time processing
- interrupt time processing

It also describes the services the XENIX kernel provides to device drivers and the rules that device drivers must follow.

8.2.1 Modes of Operation

When a process executes instructions in the user program, it is said to be in "user mode." When it executes instructions in the XENIX kernel, it is said to be in "system mode."

When the kernel receives an interrupt from an external device, it switches to system mode if it was in user mode, and control is passed to the interrupt routine of the appropriate device driver. When the driver is done, it returns, and the processing that was interrupted is resumed.

The processing that was interrupted is referred to as "task time processing," and the processing that took place as a result of the interrupt is called "interrupt time processing."

Although all processes originate as user programs, a given process may run in either user or system mode. In system mode, it executes XENIX kernel code and has privileged access to I/O devices and other services. In user mode, it executes the user's program code and has no special privileges. Where possible, XENIX provides a high level of protection around processes in user mode to prevent a user program from inadvertently damaging the system or other user programs.

A process voluntarily enters system mode when it makes a system call. When an interrupt or trap is received while a process is executing in user mode, the process switches into system mode to handle the interrupt. At this time, it may lose the CPU, and the kernel may decide to switch control or "context" to a different process.

8.2.2 Context Switching

Context switching occurs when the kernel decides to transfer control of the CPU from the currently executing process to a different process.

In user mode, the kernel switches context whenever:

- The process' time slice has expired.
- The process makes a system call that cannot be completed immediately, for example, in the case of a read from a slow input device.
- An interrupt is received that allows a blocked process to proceed. This case occurs when the process has been sleeping at high priority, waiting for the interrupt handler to call *wakeup*() to indicate a completed I/O request. If the priority at which the process is sleeping is higher than that of the currently running process, a context switch occurs.

In system mode, switching contexts is always voluntary. Interrupts can still arrive although they can be locked out for short periods of time, if necessary. When the interrupt service routine returns, control always passes back to the interrupted process. A process voluntarily gives up the processor when it calls the *sleep*() routine.

8.2.3 System Mode Stack

Each process has a special area of memory associated with it called the *u area*. The *u area* is not directly accessible to the user (that is, it is not in the process' normal address space). It contains the information the kernel needs to manage the process and contains space for a system mode stack.

When any process makes a system call, its registers are preserved in its *u area*, and the stack pointer is moved to the beginning of its system mode stack area. When the system call has completed, the registers are restored from the *u area*, the stack pointer is restored to the process' stack, and control is returned to the process. Since each process in the system has its own *u area*, a system running *N* user processes has *N* user stacks and *N* system stacks.

The XENIX operating system and, therefore, the task time portions of the device drivers use a fixed-size system mode stack in the *u area*. In XENIX, the size of this per-process stack is 1024 bytes. It is critical, then, that device driver procedures not create local (frame) buffers of any significant size. The following declaration will cause trouble, since as soon as the routine is called, it requires at least 10 4 bytes of stack space:

```
open()
{
    char buf [512];
    char buf2[512];
          .
          .
          .
}
```

Furthermore, interrupt service routines make use of whatever system stack was set up at the time of the interrupt. If the interrupt occurs while the currently running process is in user mode, the interrupt service routine will have the entire *u* stack area for its use. However, if the interrupt takes place while the process is in system mode, the interrupt routine will be sharing the *u* stack area. For this reason, interrupt service routines must minimize their frame variable declarations, keeping their frame requirements below 100 or so bytes.

8.2.4 Task Time Processing

The operating system manages a number of processes, each corresponding to a user program. Any particular process may be running in system mode or user mode at any given time. When a process makes a system call to request kernel service, the process switches to system mode, and starts running kernel code. When the kernel is executing code at the request of a user program, it is doing "task time processing."

If there are 50 processes running, there may be as many as 50 simultaneous processes in system mode, each with its own local variables. This means that all kernel code must be re-entrant, but it is otherwise fairly simple. Each system process instance has to service only the specific system call that the user program requested. The active process' *u area* is always mapped into the kernel's address space. So, when kernel code is executing, it has information about the request and process it is serving.

Often the kernel cannot service a request immediately. The request may require doing some I/O, or it could even be a request to wait awhile. When a process in system mode blocks while it is waiting for some event, the system scheduler allows some other process to run, either in user or in system mode.

I/O requests originating from the user process are passed via system calls to the device driver. Some parameters of the request, such as byte count and transfer address, are kept in the *u area*. The task time portions of the driver can reference and, perhaps, modify the *u area* cells, since we know that the currently running process's *u area* is always mapped into the kernel address space.

8.2.5 Interrupt Time Processing

When a device interrupt is received, the tasks performed as a result of the interrupt are referred to as "interrupt time processing." When an interrupt arrives, any of the active processes on the system may be executing. Even if this interrupt signals the completion of a user process' request, the interrupt service routine can take no direct action because the process that was interrupted is almost certainly not the process that initiated the request.

Instead, all interrupt time portions of the device driver routines must store information in global data structures for the task time portion of the device driver routines to figure out the result of the interrupt service. Any data or status that the interrupt service routine wants to return to the task time portion of the driver and, perhaps to the requesting user program, must also be passed via global data structures.

The local (frame) variables of the task portion of the device driver are kept in the driver's system mode stack, which is in the *u area*. This *u* *area* is not mapped into the kernel address space at interrupt time since the *u area* there belongs to some other process. The correct *u area* might even be out on the swap disk.

Thus, the interrupt service routine should never attempt to store data in the *u area* or in user memory, and the I/O device itself, via DMA or other means, should attempt to transfer data directly into the user's memory area only when the user memory has been explicitly locked into main memory. In most cases, direct transfers are forbidden.

Usually, this is not a problem. Character devices typically make use of small system-supplied buffers called character lists or *clists*. Block devices use buffers in the system buffer pool. The task time portion of the driver transfers the data from the buffers into the user's memory.

Typically, the task-time portion of the device driver issues a *sleep*() call after it makes the initial I/O request. The interrupt service routine decides what action to take and, if it needs to notify the task time portion, (as opposed to issuing another I/O command), it puts any status information and data into global data structures and issues a *wakeup*( ) call to the task portion. The interrupt service routine then exits to the operating system, and the operating system exits the interrupt. The system scheduler soon reschedules the running process so that the one that has just been awakened is executed. The task time portion of the device driver finds that it has returned from tbe *sleep*( ) call, and that there are status and data bytes waiting in global data structures.

Access to data structures that can be modified at interrupt-time is interlocked with the *spl5*(), *spl6*() and *spl7*( ) routines. These raise the interrupt priority of the CPU so that interrupts that might cause a value change are locked out until the *splx*( ) routine is called. This period must be kept as short as possible. Refer to Section 8.3.2 for a more detailed description of the routines mentioned here.

Device drivers that use the standard interfaces to the kernel are provided with a method for passing information between the interrupt-time portion of a driver and the task-time portion. Standard buffered I/O device drivers note the outcome of the data transfer in the buffer headers associated with the transfer. The header for the list of transfers the driver is working on is defined in */usr/sys/h/iobuf.h* . and the header for the buffer associated with the current transfer is defined in */usr/sys/h/buf.h* . Standard character I/O device drivers use the per device "tty" structure (defined in */usr/sys/h/tty.h*) to pass information about the I/O request.

8.2.6 Interrupt Routine Rules

An interrupt routine operates in a more restricted environment than a task-time routine, since it cannot make any assumptions about the state of the system or about the presence of particular user processes or user data in system memory. This figure illustrates the relationship between the scope of task-time and interrupt time routines:

```
            TASK                  |          INTERRUPT
            TIME                  |             TIME
      +--------------+            |
      | User Program |            |
      +--------------+            |
                                  |
                                  |
+-----------------------------+   |       +-----------+
|          u area             |   |       |           |
| --------------------------- |   |       | Driver    |
|   Kernel          |Drivers  |---------->|  Interrupt|
|                   |         |   |       | Routines  |
+-----------------------------+   |       +-----------+
                                  |
```

Task and Interrupt Time

The key things to remember are that the user process is mapped into memory, and its *u area* is mapped into the kernel's address space only at task-time. Task time processing occurs whenever the user program code itself is executing (user mode) or the operating system is executing and performing services for the program (system mode).

It cannot be assumed that the *u area* is mapped into memory during the execution of an interrupt routine. No interrupt routine, nor any routine that may be called at interrupt time, may make any reference to user memory, the *u area*, or a routine's local variables. This means that the task-time portion of the driver must not try to pass addresses of its local (frame) variables to devices and interrupt-service routines. Those locations are valid only when that individual user process is executing.

8.3 Kernel Support Routines

This section describes the functions (routines) that the kernel provides for device driver use.

Throughout this chapter, functions are described in mini-sections using the following conventions:

- When the function is introduced, the name is in boldface. (In normal text, functions are always italic followed by parentheses.)
- Parameters, if any, are boldfaced and enclosed in parentheses.
- The return value, if any, follows a colon and is in italics.

8.3.1 in(), out(), inb(), and outb()

This section describes the routines used to interface to the registers that access and control a particular device. These registers may reside either in main memory (memory mapped) or in I/O space. There are four routines that provide a portable interface to device registers.

in (port) : *word*

Purpose: This routine returns the value of the word specified by the given port or register address.

Parameters: *port* is an integer value that specifies the I/O address of the desired word.

word is an integer specifying the value of the returned word.

Result: The *(type int)* value of *word* is returned.

Example: To read the status of a word register at I/O address 20 (hex), you may use the following lines of code:

```
int val;
val = in(0x20);
```

inb (port) : *byte*

Purpose: This routine returns the value of the byte specified by the given I/O port or register address.

Parameters: *port* is an integer value that specifies the I/O address of the desired byte.

byte is a byte specifying the value of the returned byte.

Result: The value of *byte* is returned.

out (port, value)

Purpose: This routine sets the word at the specified I/O address to the specified value.

Parameters: *port* is an integer value that specifies the I/O address of the word.

value is the *(type int)* value to which the word will be set.

Result: The word at the specified I/O address is set to the specified value.

outb (port, value)

Purpose: This routine sets tbe byte at the specified I/O address to the specified value.

Parameters: *port* is an integer value that specifies the I/O address of the byte.

value is the byte value that the byte will be set to.

Result: The byte at the specified address is set to the specified value.

8.3.2 spl5(), spl6(), spl7() and splx()

This section describes the routines used to enable and disable interrupts during task-time processing.

spl5 () : *level*

spl6 () : *level*

spl7 () : *level*

> **Purpose:** These routines may be called when interrupts should not be allowed during task-time processing. *spl5*() disables all disk, floppy, printer, and keyboard interrupts. *spl6*() disables everything *spl5*() does, as well as the system clock. *spl7*() disables all interrupts, including serial interrupts.
>
> *spl6*() and *spl7*() are very rarely required, and should be avoided if at all possible. Overuse of *spl6*() causes the system clock to run slow. Overuse of *spl7*() causes serial input lines to lose characters.
>
> These routines return a code corresponding to the preempted interrupt level. This value is used when restoring interrupts with the *splx*() routine.
>
> **Parameters:** *level* is an integer *(type int)* code corresponding to the interrupt level pre-empted by this routine.
>
> **Result:** The value of the pre-empted interrupt level is returned.

splx (oldspl)

Purpose: This routine takes the return value of the *spl5*() *spl6*( ) or *spl7*( ) routines and enables the interrupt levels that were accepted before the *spl* call. *spl* calls (5-7) and their corresponding *splx*( ) calls must form matching pairs. To restore the priority level, *splx*() must be used, not a call to a lower *spl* level.

Parameters: *oldspl* is an integer value specifying the level of interrupts that were disabled by the indicated *spl* call.

Examples: To restrict interrupts during critical device driver processing, you may use the following lines of code:

```
int x;
x = spl5();
/* do critical work */
splx(x);
```

To nest two different *spl* levels, use code along the following lines:

```
int x,y;
x = spl5( );
    /* do work uninterruptable by the disk *
     * driver                              */
y = spl7( );
    /* do work so critical that it can't be  *
     * interrupted by anything               */
splx(y);
    /* back to spl5( ) for more code         *
     * uninterruptable by the disk driver    */
splx(x);
```

Do not do it this way:

```
x = spl5();
    /* do work uninterruptable by the disk *
     * driver                              */
y = spl7( );
    /* do work so critical that it can't be  *
     * interrupted by anything               */
x = spl5( );
    /* do work uninterruptable by the disk *
     * driver                              */
```

At this point, neither a call to *splx*(x) or *splx*(y) will recover the execution priority at which the driver was running before the initial *spl5*(); call.

8.3.3 sleep() and wakeup()

This section describes the routines used to suspend and reawaken requests that cannot be serviced immediately. For example, a device driver may receive a write request when the output buffer is full. In this case, the requesting process can suspend itself by calling *sleep*(). When the condition is resolved, the suspended process is awakened in either of two ways: some other process may awaken the suspended process by calling *wakeup*(), or it can be awakened by a signal.

sleep (chan, pri)

Purpose: This routine suspends a requesting process when one of the conditions required to execute the process cannot be met. This routine should never be called at interrupt time.

Parameters: *chan* is a unique number tbat identifies the sleeping process. The convention for generating this unique number is to use the address of some near data structure within the device driver. Since no two such data structures have the same address, uniqueness is guaranteed.

pri is an integer value that determines the priority of the process when it awakens. If a process goes to sleep at a priority lower than the constant PZERO, the sleep cannot be broken by a signal. This means that such a process can never be killed. Drivers sleeping at a priority below PZERO should have some guarantee of being awakened by other means.

If the condition on which the driver is sleeping is likely a priority above PZERO.

If a signal is received while sleeping, the process normally exits the system call directly, without ever returning from the sleep routine. If the driver has data structures that need to be "cleaned up" before exiting the system call, the *pri* parameter to sleep should have the PCATCH flag or'ed in. This causes sleep to return with a value of 1 if a signal caused the process to be scheduled, and a 0 if a *wakeup()* was issued.

Since this routine can return prematurely when used with a priority above PZERO, callers should ensure that the reason for sleeping is no longer valid.

For example:

```
#include "../h/param.h"
#include "../h/tty.h"

#define    MYTTYPRI   (PZERO+8)

struct  ty mytty;

while ( inb(STATUSREG) & NOTREADY )
    if ( sleep(&mytty,MYTTYPRI |PCATCH)) {
        /* clean up data struc ures */
        ...
        return;
    }
```

```
#include "../h/param.h"
#include "../h/tty.h"

#define    MYTTYPRI   (PZERO+8)

struct tty mytty;

while ( inb(STATUSREG) & NOTREADY )
    if ( sleep(&mytty,MYTTYPRI |PCATCH)) {
        /* clean up data structures */
        ...
        return;
    }
```

wakeup (chan)

Purpose: This routine wakes up any process that has been suspended by the *sleep*() routine. All the processes that have called *sleep*() with the unique number specified are awakened. When a process is awakened, the call to *sleep()* returns, and the process should check that the reason for going to sleep has disappeared.

Parameters: *chan* is a unique number that identifies the sleeping process to be awakened. The conven ion for generating this unique number is to use the address of some near data s ructure the device driver uses. Since no two such data structures have the same address, uniqueness is guaranteed.

8.3.4 timeout() and delay()

This section describes the functions used to suspend driver execution for a period of time, or schedule a function call sometime later.

timeout (function, arg, tim)

Purpose: This routine allows a function to be called at a scheduled time in the future.

Parameters: *function* is the address of the function to be called.

arg is the argument to the function being called.

tim is an integer value specifying the number of clock ticks that sbould elapse before the call. This should be specified using the variable *Hz*, which contains the number of ticks in a second.

Example: This routine can be used, along with *sleep*() and *wakeup*() to provide "busy waiting." The following code sample illustrates this:

```
#define BUSYPRI (PZERO +1)   /* arbitrary */
int stopwait();
int status;

int busywait()  /* wait until status is non-zero */
{
    while (status == 0) {
        timeout(stopwait, 0, Hz/10);  /* 1/10 of a second */
        sleep(&status, BUSYPRI);
    }
}

int stopwait()
{
    wakeup(&status);
}
```

WARNINGS

A driver should never loop without a timeout while waiting for a status change unless the delay involved is shorter than 100 microseconds.

The XENIX timeout table is a finite size. Excessive use of timeouts can cause the table to overflow, which is a *panic*( ). Drivers requiring large numbers of timeouts should consider using the *xxpoll*( ) routine, discussed in section 8.6.1.

---

delay (ticks)

Purpose: This routine causes the calling process to sleep for a specified number of clock ticks, and wakes it up when that many clock ticks have passed.

Parameters: *ticks* is an integer that specifies the number of clock ticks to delay.

Result: After the specified time, the delayed function resumes running.

Warning: This routine should not be called at device initialization (*init*) time. The *delay*( ) routine uses *timeout*(), *sleep*( ), and *wakeup*( ) functions, which depend on the system being fully initialized.

Example: This delays for two seconds:

```
delay (Hz*2);
```

8.3.5 dscralloc(), dscrfree(), dscraddr(), & mmudescr()

This section describes the routines used by drivers running in protected mode to access memory that is not within kernel data. A descriptor from the Global Descriptor Table (GDT) is initialized to map the memory area, and then used to access the memory. These routines do not exist within XENIX kernels running in 'real' or 'unprotected' mode.

Among the uses for these routines are accessing video RAM, talking to device outboard buffers, and generally communicating with any memory outside of the normal kernel address space.

dscralloc () : *sel*

Purpose: This routine allocates a descriptor from the pool of GDT descriptors available for drivers. It returns the selector number of the allocated descriptor.

Return: *sel* is an unsigned short value that specifies the selector number of the allocated descriptor.

Result: This routine returns 0 if no more descriptors are available, and prints the message below on the system console:

```
Out of device descriptors, increase gdt size (NGDT) and relink XENIX
```

Otherwise, it returns the selector number of the allocated descriptor.

Note: It is important that the driver verify that the return value is valid (not 0). Any attempt to use descriptor 0 may cause the kernel to crash.

dscrfree (sel)

Purpose: This routine returns a descriptor that is no longer needed to the pool of available device descriptors. It takes as its only argument the selector number returned from a call to **dscralloc( )**.

A device that uses a descriptor for most or all of its transfers should not release it, but should reuse the same descriptor for each transfer. Only devices that need a descriptor for a short period of time (during initialization, for example) should ever free a descriptor.

Parameters: *sel* is an unsigned short value that specifies the selector number of the descriptor being freed.

dscraddr (sel) : *addr*

Purpose: This routine returns the physical address of the memory addressed by the selector which is provided as the argument.

Parameters: *sel* is an unsigned short value that specifies the selector number provided as the argument.

Result: Returns *addr*, which is the 32-bit physical address of the memory addressed by the selector.

mmudescr (sel, addr, limit, access)

Purpose: This routine initializes a descriptor to map an area of memory.

Parameters: *sel* is an unsigned short value that specifies the selector number of the descriptor allocated by *dscralloc*( ).

addr is the 32-bit physical address of the memory addressed by the selector.

limit is an unsigned short value that specifies the limit of the memory area (its size in bytes − 1).

access is a byte value that specifies an access designation.

Example: The *mmudescr*( ) routine maps a section of memory 1024 bytes long at address 0xB0000 for reading and writing as follows:

```
mmudescr(sel, 0xB0000, 0x3FF, DSA_DATA);
```

access may be RW, RO, or DSA_DATA. RW specifies read/write access to the memory area. RO specifies read/execute access to the memory area. Both RW and RO allow user access to the selectors.

Driver private data selectors should use the DSA_DATA code. DSA_DATA is defined in */usr/sys/h/relsym.h,* which is included by */usr/sys/h/mmu.h.* Both RW and RO are defined in */usr/sys/h/mmu.h.*

To Map Memory Using These Routines

The normal sequence of events for a device driver that needs to use a selector to map memory is:

1. Use *dscralloc*() in the driver initialization routine, or on first open for this device, to reserve a descriptor for this driver's use.
2. For each data transfer, use *mmudescr*( ) to set the descriptor to map the area of memory that the driver needs to access.

Example: This code allocates a descriptor, then maps a 512-byte area beginning at 0xB0000 into the kernel's address space. Remember that the third argument is the limit of the transfer, not its size. After the area is mapped, the *faddr* variable can be used like any other kernel logical address.

```
int seg;
faddr_t faddr;

seg = dscralloc();
mmudescr(seg,0xB0000,511,RW);
faddr = sotofar(seg,0);
```

8.3.6 copyin(), copyout(), and copyio()

copyin (user_addr, sys_addr, nbytes) : *error*

Purpose: *copyin*() copies user data into system data. Although *copyio*() can accomplish the same task using the U_WUD mapping argument, *copyin*() is often used instead for simplicity.

Parameters: *user_addr* is the address in user space from which to copy. It is of type *faddr_t,* as defined (typedef) in */usr/sys/h/types.h.* This is currently a *char* * on unmapped machines, and a *char far* * on mapped machines.

sys_addr is the address in system space to which to copy. It is a character pointer.

nbytes is an integer value that specifies the number of bytes of data to transfer.

Result: Returns -1 on error, such as on an attempt to copy outside of user space.

copyout (sys_addr, user_addr, nbytes) : *error*

Purpose: This routine copies system data into user data. Although *copyio*() can accomplish the same task using the U_RUD mapping argument, *copyout*() is often used instead for simplicity.

Parameters: *sys_addr* is the address in system space from which to copy. It is a character pointer. *user_addr* is the address in user space to which to copy. It is of type *faddr_t*, as defined (typedef) in */usr/sys/h/types.h*. This is currently a *char* * on unmapped machines, and a *char far* * on mapped machines. *nbytes* is an integer value that specifies the number of bytes of data to transfer.

Result: Returns -1 on error, such as on an attempt to copy outside of user space.

copyio (addr, faddr, cnt, mapping) : *error*

Purpose: This routine can be used to copy bytes between a physical address and a logical address. Although *copyio()* can be used to copy almost anything to almost anything, the form in which the arguments must be given make it appropriate for copying between kernel data and user space, and inappropriate for copying between two physical addresses outside of kernel space. For transferring between two physical addresses, direct physical copying is both more efficient and more simple to set up. See the memory-mapped video driver in section 9 for an example of direct physical copying.

On mapped machines, the *copyio()* routine should not be used at interrupt time.

Parameters: *addr* is the physical address to which or from which the data is to be transferred. It is of type paddr_t, which is a 32-bit quantity, and contains the literal physical address.

faddr is the logical address to which or from which the data is to be transferred. It is of type *faddr_t,* which is a *char* far * on mapped machines, and a *char* * on unmapped machines.

On unmapped machines, the logical address is simply an offset from the beginning of the appropriate segment. The *mapping* argument is used to determine which segment is appropriate.

On mapped machines, the logical address has a global descriptor as its 'segment'. Character pointer arguments to system calls are already in this form. Kernel data addresses can be used as 'logical addresses' by casting them to a (faddr_t). Otherwise a logical address can be created using the macro sotofar(seg,off), where off is a 16-bit offset, and seg is the selector used in an earlier *mmudescr*() call.

cnt is an integer value that specifies the number of bytes of data to transfer.

mapping is an integer that designates the direction of the transfer. The possible mapping values are defined in */usr/sys/h/user.h* and are listed below:

U_WUD	transfer from user data
U_RUD	transfer to user data
U_WUI	transfer from user text
U_RUI	transfer to user text
U_WKD	transfer from kernel data
U_RKD	transfer to kernel data

Result: If successful, this routine performs the specified data transfer; otherwise, it returns -1.

Example: *ioctl*( ) could use *copyio*( ) to transfer data between kernel and user space. As a practical fact, it uses *copyin( )* and *copyout( )* for its transfers, so this example is hypothetical.

The *ioctl* interface to a driver has a third argument of indeterminate type. It is assumed here to be an integer pointer, but could be an integer as well.

```
ioctl (fd, cmd, arg)
int fd, cmd;
int *arg;
```

In the kernel, the *ioctl* interface is translated into the device-specific call shown below:

```
xxioctl (dev, cmd, arg)
int dev, cmd;
faddr_t arg;
```

Here "arg" is a pointer to a data structure. The example copies from this data structure into the dst structure.

```
struct foo dst;
        .
        . other ioctl code
        .
/* copy from arg to dst */

if ( copyio ( ktop(&dst), arg, sizeof(foo), U_WUD) == -1 ) {
        u.error = EFAULT;
        return;
}
```

Note: The file named */usr/sys/h/param.h* defines several macros that are useful for converting addresses from one type to another. These macros include:

ftoseg(x)	converts x from an faddr_t to a segment (selector number)
ftooff(x)	converts x from an faddr_t to an offset
sotofar(seg,off)	converts a segment, offset pair into an faddr_t
ptok(x)	converts a physical address within kernel space to a kernel data (char *) logical address
ktop(x)	converts a kernel data logical address to a physical address

U_WUD	transfer from user data
U_RUD	transfer to user data
U_WUI	transfer from user text
U_RUI	transfer to user text
U_WKD	transfer from kernel data
U_RKD	transfer to kernel data

8.3.7 putchar() and printf()

This section describes the routines that display or print messages on the system console.

putchar (c)

Purpose: This routine puts one character on the console, doing a "busy wait" rather than depending on interrupts.

Parameters: *c* is the character to be printed on the console.

printf (format, p1, p2, ...)

Purpose: The kernel *printf*() routine is a simplified version of the standard C library *printf*() routine. It is used to print error messages and debugging information on the system console. The special format characters understood by the kernel *printf* are: %s, %d, %ld, %lx, %u, %D, %X, %x, and %o, plus the NEWLINE and RETURN characters. See **printf(S)** for more information on printf() and its parameters.

Note that this routine is not interrupt driven and will suspend all other system activities while it is executing.

Parameters: *format* is the *printf*() format string.

p1,p2,... are the additional parameters to be printed by the routine.

8.3.8 panic(), signal(), psignal(), and suser()

This section describes routines that perform miscellaneous system functions.

panic (s)

Purpose: This routine is called whenever an unrecoverable kernel error is encountered. It prints the string that is passed as a parameter on the system console and halts the system. This routine should be called only under extreme and unrecoverable circumstances.

Parameters: *s* is a *char* * addressing a message that explains the reason for the system panic.

signal (pgrp, signum)

Purpose: This routine sends the specified signal, *signum*, to all processes in the process group identified by *pgrp*.

Parameters: *pgrp* is an integer that specifies the process group number. A process can determine its own process group number by examining *u.u_ttyp->t_pgrp*. See */usr/sys/h/user.h* and */usr/sys/h/tty.h* for the structures in which these fields are defined.

signum is the signal to be sent.

psignal (proc, signum)

Purpose: This routine sends the specified signal, *signum*, to the process *proc*.

Parameters: *proc* is a pointer to a *struct proc*. A process can determine its own *proc* structure pointer by examining *u.u_procp*. See */usr/sys/h/user.h* for the definition of this field.

signum is the signal to be sent.

suser () : *int*

Purpose: This routine determines whether the user associated with the currently executing process is the super-user. This can be useful, for example, in determining whether special device operations (such as being able to override exclusive use restrictions) are allowed.

Result: The routine returns an integer (*int*): 0 if the current user is not the super-user and 1 if the user is the super-user. As a side effect, *u.u_error* is set to EPERM if the user is not the super-user.

8.3.9 Memory Allocation Routines

The XENIX kernel is limited to 64K of near data, like any other middle or hybrid model program. Most of the memory is already allocated among the various kernel data structures.

If more near memory is required, the quantity of various structures can be reduced until near kernel data fits in 64K. System addressable buffers, *inodes*, and *clists* are the structures most commonly decreased. This can be done by using the **configure**(C) program to modify the constants NINODE, NSABUF, and NCLIST. For information about **configure**, see the XENIX *User's Reference*.

Far data can be allocated by simply declaring it in the driver, using declarations such as:

```
        char far outbuf[1024];
```

This data must then be addressed with far pointers.

Virtual Memory Allocation Routines

These routines are used to allocate kernel-addressable virtual memory on paged machines such as the 80386.

sptalloc (nbytes)

Purpose: This routine allocates *nbytes* of kernel-addressable virtual memory.

Parameters: *nbytes* is the number of bytes of memory to allocate.

Result: The address of the base of the allocated memory is returned. May *panic* if no virtual memory remains; it is unlikely that any legitimate caller of this routine would desire that much memory.

sptfree (virtaddr, nbytes, freeflg)

Purpose: This routine frees the memory allocated by *sptalloc( )*.

Parameters: *virtaddr* is the base of the allocated memory that was returned by *sptalloc()*. *nbytes* is the number of bytes to free. *freeflg* is true if the memory should actually be freed. Some *u*-structure managing routines call *sptfree()* with *freeflg* set to false as a type of memory semaphore.

Result: The virtual memory is freed into its memory map.

Physical Memory Allocation Routines

These routines are used on machines that lack virtual memory, such as the 8086 and the 80286. The means of physical memory allocation differs depending on whether the desired memory is within kernel data space or outside of the kernel.

Kernel Data Allocation. Since the kernel's data segments are already allocated among the various kernel data structures, the only way to allocate kernel data at run-time is to procure a kernel data structure of an appropriate size.

For small memory allocation requirements, the data section of a cblock will provide you with CLSIZE usable bytes, as defined in */usr/sys/h/tty.h*. These data structures can be allocated using the *getcf*() call, and deallocated using the *putcf*() call, which are discussed in section 8.6.1, "Character Device Driver Routines."

For larger requirements, the data section of a disk buffer can provide you with BSIZE usable bytes, currently 1024. These data structures can be allocated with the *getablk*() call, and released using the *brelse*() call, which are discussed in section 8.7.2, "Block Device Driver Routines."

User Memory Allocation. User memory is all memory outside the realm of the kernel. The *mavail()* and *mlrgest()* routines can be used to determine the amount of free memory, and the largest contiguous area of free memory, respectively. User memory is allocated using the *mmuget*() routine, and released using the *mmufree*() routine.

mmuget() allocates memory directly from the system memory map. If memory is allocated during normal driver run time, as opposed to driver initialization, *mmuget*() can have an undesirable side affect.

The memory map can become fragmented, which can prevent large process from swapping in. Deadlock could even occur from the misuse of *mmuget*(). For example, a driver could fragment system memory so that "Process A" had no room to swap in, and yet be waiting for "Process A" to notify it before freeing the allocated memory. Therefore, memory allocated during normal driver run time should be released as soon as possible.

Memory may be permanently allocated via *mmuget()* at driver initialization time. This memory will not cause fragmentation, as it is allocated contiguously with other kernel data.

These routines should absolutely not be used at interrupt time.

mavail (&coremap) : *pages*

> **Purpose:** This routine reports the total number of memory pages currently free. The pages may lie in several different areas, and are not necessarily contiguous.
>
> **Parameters:** *coremap*[] is the global kernel data structure for recording free memory pages.
>
> Result: The total number of free memory pages is returned.

mlrgst (&coremap) : *pages*

> **Purpose:** This routine reports the number of pages in the larges contiguous area of currently free memory.
>
> **Parameters:** *coremap*[] is the global kernel data structure for recording free memory pages.
>
> **Result:** The size in pages of the larges contiguous area of free memory is returned.

mmuget (&base, pages) *: error*

mmuget (pages) : *base*

Purpose: This routine allocates memory from the system memory map. Memory is allocated in units of *pages*, which is a machine-dependent unit. A macro is provided to convert bytes to pages. The *base* address may then be used with *mmudescr()* on memory mapped machines, and may be used directly on unmapped machines (see example).

Parameters: For historical reasons, this routine has two completely different forms: the first on memory mapped machines, the second on unmapped machines. Drivers that intend to run on both architectures require conditionally compiled code.

However, conditional compilation would be necessary even if *mmuget*() did not have two forms, as mapped machines require selector mapping to be performed as well. The selector mapping routines are covered in detail in section 8.3.5. The example should demonstrate the combined use of *mmuget*(), the selector mapping routines, and the various kernel address-modifying macros.

In both cases, base is an unsigned integer, and pages is the number of pages to allocate. The *btoms*( ) macro, defined in */usr/sys/h/param.h*, will perform the indicated bytes-to-pages conversion.

Result: If MMUERR is returned, no memory was allocated. Otherwise, base is assigned to the base of the allocated memory.

Example: This code allocates nbytes bytes of user memory, placing the segment of the allocated area in the variable "seg". Note that all three variables are used again when the memory is freed, below:

```
        int seg, nbytes;
        unsigned short base;

    #ifdef M_I286
        base = mmuget(btoms(nbytes));
        seg  = dscralloc();
        mmudescr(seg,mltoa(base),nbytes,RW);
    #else
        mmuget(&base, btoms(nbytes));
        seg = ptoseg(mltoa(base));
    #endif
```

mmufree (seg, pages)

Purpose: This routine deallocates memory, returning it to the system memory map so it can be reallocated. Memory must be freed in pages, the same unit as in which it was allocated.

Parameters: This routine takes the base value, which is the unsigned integer returned from a mmuget call, and a number of pages, which is the number of pages to free.

On mapped machines, if the mapping selector obtained from *dscralloc*() is not to be reused, it should be released with *dscrfree*().

Example: This code deallocates nbytes bytes of user memory, freeing the memory selector at the same time (*seg*, *nbytes* and *base* are as the *mmuget( )* example.

```
        int seg, nbytes;
        unsigned short base;

        mmufree(base,btoms(nbytes));
    #ifdef M_I286
        dscrfree(seg);
    #endif
```

8.3.10 DMA Allocation Routines

These routines allow DMA usage to be interlocked against DMA requests by other drivers. Not all devices use DMA, but those that do must have exclusive access to their DMA channel for the duration of the transfer.

The number of DMA channels is hardware dependent. Some channels are reserved for such invisible housekeeping functions as screen refresh and cannot be reallocated.

Some machines have DMA chips that malfunction when more than one allocated channel is used simultaneously. To allow installation on these machines, the *dma_single* flag is set by default. On machines that do not suffer from this deficiency, clear the *dma_single* flag to allow simultaneous DMA on multiple channels.

The names of the various channels are defined in the file *dma.h*.

dma_alloc (channel, mode)

Purpose: This routine allocates a DMA channel.

Parameters: *channel* is the channel to be allocated. If *mode* is DMA_NBLOCK, the routine will not sleep until the specified channel is available, instead returning a non-zero value immediately. If *mode* is DMA_BLOCK, the routine will sleep until the channel is available. This routine may only be called at interrupt time if DMA_NBLOCK is specified.

Result: Returns 0 if the channel is allocated, otherwise 1.

dma_relse (channel)

Purpose: This routine releases a DMA channel that was either allocated with *dma_alloc*, or implicitly allocated by *dma_start*. It should be called as soon as the DMA transfer completes.

Parameters: *channel* is the channel name that was presented earlier to *dma_alloc()* or *dma_start*.

Result: No return value.

dma_start (dmareqptr)

Purpose: This routine starts up a DMA request. It is designed to be used at interrupt time. When the channel is available, the *d_proc* routine will be called at *spl6( ), with a pointer to d_param* as an argument. The *d_proc* and *d_param* values are found in the structure pointed to by *dmareqptr*. The routine specified by *d_proc* must follow all the normal rules of interrupt routines. It should be minimal because it may be called during some other device's interrupt routine.

Parameters: The *dmareqptr* structure is defined as follows:

```
struct dmareq {
    struct dmareq    *d_nxt;
    unsigned short    d_chan;
    unsigned short   d_mode;
    paddr_t        d_addr;
    long        d_cnt;
    int     (*d_proc)();
    char        *d_params;
} *dmareqptr;
```

The *d_nxt* field is used to link the structure onto a list of dmareq structures in case it can't be serviced immediately. The *d_mode* field supplies the direction of the transfer: it is either *DMA_Wrmode* (from memory to the device) or *DMA_Rdmode* (from the device to memory). The *d_addr* field contains the physical address from which or to which to transfer. The *d_cnt* field contains the number of bytes or words to transfer. The *d_proc* routine will be called at priority *spl6( )* when the channel is available.

Result: Returns 1 if the request was completed immediately, 0 if it was queued for later execution.

dma_param (channel, mode, addr, cnt)

Purpose: This routine masks the DMA request line on the controller, sets the address and count parameters, and sets the mode (read or write).

Parameters: *channel* is the channel number that was earlier presented to *dma_alloc()*. *mode* is either *DMA_Wrmode* for a write transfer (from memory to the device), or *DMA_Rdmode* for a read transfer (from the device to memory). *addr* is the physical address from which or to which to transfer. *cnt* is the number of bytes to transfer.

Result: The controller is initialized.

dma_enable (channel)

Purpose: This routine clears the mask register on the controller to allow the DMA transfer to begin.

Parameters: *channel* is the channel name that was earlier presented to *dma_alloc()*.

Result: The transfer will take place.

dma_resid (channel) :resid

Purpose: This routine returns the number of bytes that were not transferred by the previous *dma_enable()* request, as a long.

Parameters: *channel* is the channel name that was earlier presented to *dma_alloc()*.

Result: A long integer expressing the number of bytes not transferred will be returned.

8.3.11 Version 7/System V Compatibility Issues

This section describes some of the changes between Version 7 UNIX and System V UNIX that affect the device driver interface.

Device Numbers

In Version 7 UNIX, the *dev* parameter passed to the *open*(), *close*(), *read*(), *write*(), and *ioctl*() driver routines included the major and minor device numbers. In System III and System V, only the minor device number is passed in the *dev* parameter. This means it is no longer necessary for all device drivers to mask out the major device number before checking the minor device number.

iomove ()

Some Version 7 device drivers used a routine called *iomove*() to copy to or from user space. The *iomove*() routine does not exist in System III and System V. However, adding the code shown below provides most of the same capability:

```
#include "../h/param.h"
#include "../h/dir.h"
#include "../h/user.h"
/*
 * iomove - equivalent to the V7 version except we do not provide
 *      any of the standard segflg machinations for writing
 *     to instruction space or kernel data space
 *   NOTE:  u.u_base is an faddr_t
 */

iomove(cp, cnt, flag)
caddr_t cp;
register int cnt;
int flag;
{

   register int ret_val;

   if (cnt == 0)
      return;         /* Nothing to do! */

   if( flag == B_WRITE )
      ret_val = copyio((caddr_t)ktop(cp), u.u_base, cnt, U_WUD);
   else
      ret_val = copyio((caddr_t)ktop(cp), u.u_base, cnt, U_RUD);
   if( ret_val == -1 ) {
      u.uerror = EFAULT;
      return;
   }
   u.u_base += cnt;
   u.u_count -= cnt;
   u.u_offset += cnt;
}
```

8.4 Parameter Passing to Device Drivers

The task-time portion of the device driver has access to the user's *u area* since this is mapped into kernel address space. The kernel routines that process the user process' I/O request, place information describing the request into the process' *u area*. The parameters passed in the *u area* are:

u.u_base	address in user data to read/write data for transfer
u.u_count	the number of bytes to transfer
u.u_offset	the start address within the file for transfer
u.u_segflg	indicates the direction of the transfer
u.u_error	used to return errors to the user.

Refer to the */usr/sys/h/user.h* file for the values to use for u.u_segflg. Refer to **intro**(S) for a list of values for u.u_error. In addition to the parameters passed in the *u area*, the kernel I/O routines pass the minor device number as a parameter to the driver when it is called. Therefore, the driver has all the information it needs to perform the request: the target device, the size of the data transfer, the starting address on the device, and the address in the process' data.

Only device drivers that do not use standard character and block I/O interfaces in the kernel need examine the parameters in the *u area*. Kernel routines that provide these standard interfaces have done the work of converting the values passed in the *u area* into values that the driver expects. In the case of the standard block I/O interface, these parameters are set in the buffer header describing the data transfer. Refer to Section 8.7, "Device Drivers for Block Devices," for more information on using the buffer header information to set up a block data transfer.

Device drivers using the standard character I/O interface use the *clist* buffering scheme and the routines that manipulate the *clist* to effect the data transfers. Refer to Section 8.6, "Device Drivers for Character Devices," for more information on using *clists* and the character I/O interface routines.

8.5 Naming Conventions

There is a naming convention for all driver routines names and all non-static driver global variables. Each driver uses a unique two- to four-character prefix to identify its routines and variables. This helps prevent naming collisions.

For example, the *xxstrategy*( ) function for a hard disk driver might use the prefix "hd". The resulting name would be *hdstrategy*( ); for a floppy driver ("fd"), *fdstrategy*( ). In the following sections, the prefix used is "xx".

8.6 Device Drivers for Character Devices

This section describes XENIX character device drivers. Character devices conform to the XENIX file model; their data consists of a stream of bytes delimited only by the end of file. The XENIX system provides programs with direct access to devices through the special device files described in Section 8.1.5, "Special Device Files."

Many special facilities are provided for the special requirements of serial devices, such as programming functions on input and output (character erase, line kill, tab functions, etc.), and for setting line options such as speed. The drivers for other character-oriented devices such as line printers are basically simplified serial device drivers that do not use special facilities that are available to serial drivers.

Most character device drivers use a data buffering mechanism known as a character list or *clist*. *clists* are used to transfer relatively small amounts of data between the driver and the user program. *clists* are described in more detail in Section 8.6.3, "Character List and Character Block Architecture."

8.6.1 Character Device Driver Routines

The task-time portion of the character device driver is called when a user process requests a data transfer to or from a device under the control of the driver. The system determines the kind of request from the major device number of the device used to do I/O. The driver's job is to take the user process' requests, check the parameters supplied, and set up the necessary information for the device interrupt routine to perform the I/O.

In the case of a write to a slow device, that is, one using *clists*, the driver copies the data from user space into the output *clist* for the device. In the case of direct I/O between the device and user memory, for example, magnetic tapes, the driver simply sets up the I/O request. The routines that provide the interface between the kernel and character device drivers follow.

The next sections discuss the different kinds of character device driver routines:

- Driver Defined Routines
- Serial Driver Support Routines
- Character Passing Routines
- Character List Routines
- Line Discipline Routines

Driver Defined Routines

These routines are driver entry points: the only places where the kernel calls the device driver. "xx" is a mnemonic that refers to the device type. (See 8.5, "Naming Conventions".)

xxinit ()

Purpose: This routine is called to initialize the device when XENIX is first booted. If present, it is called indirectly through its entry in the *dinitsw*[ ] table. */usr/sys/conf/c.c*.

xxopen (dev, flag, id)

Purpose: This routine is called each time the device is opened. It prepares the device for the I/O transfers and performs any error or protection checking.

Parameters: *dev* is an integer that specifies the minor number of the device.

flag is the mode in which the file should be opened. It is the bitwise "or" of the modes defined in */usr/sys/h/file.h*. Note that this *flag* is similar to, though not exactly the same as the *oflag* argument that is passed to the *open*() system call.

id is an integer that specifies whether the device is a character device (0) or a block device (1).

If this routine sets *u.u_error* to a non-zero value, the *open* has failed and the value in *u.u_error* will be returned to the user as *errno*.

xxclose (dev, flag, id)

Purpose: This routine is called on the last close on a device. It is responsible for any cleanup that may be required such as disabling interrupts, clearing device registers, and so on.

Parameters: *dev* is an integer that specifies the minor number of the device.

flag is the mode in which the file should be opened. It is the bitwise "or" of the modes defined in */usr/sys/h/file.h*. Note that this *flag* is similar to, though not exactly the same as the *oflag* argument that is passed to the *open*( ) system call.

id is an integer that specifies whether the device is a character device (0) or a block device (1).

xxintr (vec_num)

Purpose: This routine is called by the kernel when the device issues an interrupt. Since the interrupt typically signals completion of a data transfer, the interrupt routine must determine the appropriate action. This may be taking the received character and placing it in the input buffer, or removing the next character from the output buffer and starting the transmission.

Parameters: *vec_num* is an integer that specifies the interrupt vector number.

Note that *xxintr* will be called at the *spl* level specified in *master* (see section 8.9.2). No interrupts from the same device will be acknowledged unless the interrupt routine explicitly lowers its *spl* level, Interrupt routines do not normally lower their *spl* level.

xxread (dev)

Purpose: This routine is called when a program makes a read system call. Its responsibility is to transfer data to the user's address space. A subroutine is available to transfer one character at a time to the user: *passc*(). This subroutine returns a -1 when there are no more characters to be transferred.

Parameters: *dev* is an integer that specifies the minor number of the device.

xxwrite (dev)

Purpose: This routine is called when a program makes a write system call. Its responsibility is to transfer data from the user's address space. A subroutine is available to transfer one character at a time from the user: *cpass*(). This subroutine returns a -1 when there are no more characters to be transferred.

Parameters: *dev* is an integer that specifies the minor number of the device.

xxproc (tp, cmd)

Purpose: This routine is called to perform output character expansion, output characters, halt or restart character output and, in general, bring about the desired change in the output.

Parameters: *tp* specifies the *tty* value of the device.

cmd specifies the process to be performed. The sample *tty* driver in Chapter 9, "Sample Device Drivers," documents the list of *cmd* argument values for *xxproc*().

xxioctl (dev, cmd, arg, mode)

Purpose: This routine is called by the kernel when a user process makes an *ioctl()* system call for the specified device. It performs hardware dependent functions such as setting the data rate on a character device.

Parameters: *dev* is an integer that specifies the minor device number of the device.

cmd is an integer that specifies the command passed to the system call.

arg specifies the argument passed to the system call.

mode is the bitwise "or" of the file modes defined in */usr/sys/h/file.h*. It is similar, but not exactly the same as, the flags passed to the *open*() system call for the device.

If this routine sets *u.u_error* to a non-zero value, the *ioctl* has failed and the value in *u.u_error* will be returned to the user as *errno*.

xxpoll (ps)

Purpose: This routine, if present, is called by the system clock at *spl6()* during every clock tick. It is useful for repriming devices that constantly lose interrupts.

Parameters: *ps* is an integer that indicates the previous process's priority when it was interrupted by the system clock. The macro USERMODE(*ps*), defined in */usr/sys/h/param.h,* can be used to determine if the interrupted process was executing in user mode.

Serial Driver Support Routines

This section describes the routines that initialize the *tty* structures, start the *tty* output, and empty the *tty* queue. These routines are used almost exclusively by serial drivers.

ttinit (tp)

Purpose: This routine initializes the *tty* structure to specific default values. To set up the default settings for a *tty* device, call this routine immediately after opening the *tty* device. *ttinit* initializes the *t_line, t_iflag, t_oflag* , *t_cflag, t_lflag,* and *t_cc* fields of the *tty* structure.

Parameters: *tp* is a *struct tty** that points to the *tty* data structure associated with the device being used.

ttiocom (tp, cmd, addr, flag)

Purpose: This routine is called for all common *tty ioctl* calls. It is called by the *xxioctl()* routine after a device specific *ioctl* has been performed.

Parameters: *tp* is a *struct tty** that points to the *tty* data structure associated with the device being used.

cmd is an integer specifying an *ioctl* command.

addr specifies the address of the user space where the parameters reside.

flag specifies whether the command is a read or write operation.

ttrstrt (tp)

Purpose: This routine restarts *tty* output after a *timeout*( ) call. It is passed as an argument by the device driver to *timeout*( ) calls.

Parameters: *tp* is a *struct tty** that points to the *tty* data structure associated with the device being used.

ttyflush (tp, cmd)

Purpose: This routine flushes the *tty* queue.

Parameters: *tp* is a *struct tty** that points to the *tty* data structure associated with the device being used.

cmd specifies whether to flush the input (FREAD) queue or the output (FWRITE) queue. (FREAD) and (FWRITE) are defined in */usr/sys/h/file.h*

Character List Routines

The kernel contains a group of small buffers called character lists, or *clists*. A *clist* structure is the head of a linked list queue of characters. The elements in the linked list are called *cblocks* and each *cblock* can hold a small number of characters. These are used for buffering low-speed character devices. Refer to Section 8.6.3, "Character List and Character Block Architecture," for further information on *clists*.

Drivers that do not use the *tty* structure must declare a queue header of type *clist*, or two queue headers if both input and output are to be buffered. The *tty* structure already contains declarations for the needed queue headers. There are eight routines that the driver can use to manipulate *clist* buffers, as described below. All these routines can be used during interrupt-time processing.

getc (cp)

Purpose: This routine moves one character from the *clist* buffer for each call.

Parameters: *cp* specifies the *clist* buffer from which characters are moved.

Result: This routine returns the next character in the buffer or -1 if the buffer is empty.

putc (c, cp)

Purpose: This routine moves one character to the *clist* buffer for each call.

Parameters: *c* is an integer that specifies the character to be moved.

cp specifies the *clist* buffer to which the character is moved.

Result: This routine places the specified character in the buffer or returns -1 if there is no free space. A driver may suspend processing until there is free space by sleeping on the address of *cfreelist*.

Example: This code places the specified character in the buffer, suspending processing if necessary until there is room in the buffer.

```
while (putc(c,cp) == -1)
    sleep(&cfreelist,ARBITRARY_PRI);
```

getcb (cp) : *cbp*

Purpose: This routine moves one *cblock* from the *clist* buffer for each call.

Parameters: *cp* specifies the *clist* buffer from which the *cblocks* are moved.

cbp is a pointer to a *cblock*.

Result: This routine returns the next *cblock* (*cbp*) in the buffer or NULL if the buffer is empty.

putcb (cbp, cp)

Purpose: This routine moves one *cblock* to the *clist* buffer for each call.

Parameters: *cbp* is a pointer that specifies the *cblock* to be moved.

cp is a pointer that specifies the *clist* to which the *cblock* is linked.

Result: This routine places the specified *cblock* on the linked list associated with *cp*.

getcf () : *cbp*

Purpose: This routine takes a *cblock* from the freelist, and returns a pointer to it.

Result: Returns *cbp*, a pointer to a *cblock*, or NULL if there are no free cblocks available. A driver may suspend processing until the free *cblock* situation changes by sleeping on the address of *cfreelist* (see example under *putc*).

putcf (cbp)

Purpose: This routine puts the specified *cblock* onto the freelist.

Parameters: *cbp* is a pointer to a *cblock*.

getcbp (p, cp, n)

Purpose: This routine copies characters from the specified *clist*, *p*, to the buffer addressed by the *cp* argument.

Parameters: *p* is a *struct clist* * .

cp is a *char* * addressing the buffer to which the characters are to be copied.

n is the number of characters to be copied.

Result: This routine returns the number of characters actually copied, which is less than or equal to *n*.

This routine must be called at *spl6( )*.

putcbp (p, cp, n)

Purpose: This routine copies characters from a buffer to the *clist* given as argument.

Parameters: *p* is a *struct clist* * .

cp is a *char* * which addresses the buffer.

n is the number of characters to be copied to the *clist*.

This routine must be called at *spl6( )*.

Character Passing Routines

These routines are the lowest level of character I/O. They are used to pass characters between kernel and user space.

They differ from *copyio*(), *copyin*() and *copyout*() in that the appropriate fields in the user structure (u.u_base, u.u_count, etc.) are updated when these routines are used. Most drivers do not call character passing routines directly, instead relying on the Character List Routines that, in turn, call *cpass*() and *passc*().

cpass () : *c*

> **Purpose:** This routine returns the next character from the user output request,
>
> **Result:** The routine returns a character *c* or the value -1, which indicates that there are no characters left in the output request.

passc (c)

> **Purpose:** This routine passes characters to a user read request.
>
> **Parameters:** *c* is the character to be passed to the read request.
>
> **Result:** The routine returns 0 normally and -1 when the user read request has been satisfied.

Line Discipline Routines

If a serial device is to be used as an interactive terminal, it must support various functions such as character and line erase, echoing, and buffered input. The code needed to perform each of these functions has been abstracted into a set of routines that roughly corresponds to the character device function. Each of these sets is called a "line discipline". One standard line discipline is provided by default. Each of the routines is called through the *linesw*[ ] table initialized in */usr/sys/conf/c.c*. Each entry in this table represents one line discipline, and has entries for eight functions.

The *l_open*() routine should be called on the first open of a device. The *l_close*() routine should be called on the last close of the device. The *l_read*() and *l_write*() routines are called by the drivers read and write routines, to pass characters to and from the calling process. The *l_input*() routine is called to buffer an incoming character, usually upon the receipt of the character from the hardware. *l_ioctl* calls specific routines related to line discipline manipulation. The *l_output* routine gets the next block of characters for output at interrupt time. The *l_mdmint*() routine is not currently used.

8.6.2 Interrupt Routines for Character Device Drivers

The device interrupt routine is entered whenever one of the driver's devices raises an interrupt. Note that, in general, one driver may control several devices. All interrupts, however, are vectored through a single function entry point. The entry point is usually called *xxintr*(), where *xx* is a mnemonic that refers to the device type (see Section 8.5, "Naming Conventions"). It is the driver's responsibility to decide which device caused the interrupt.

When a device raises an interrupt, it generally makes available some status information to indicate the reason for the interrupt. The driver interrupt routine decodes this information. If it indicates that a transfer has just completed, the *wakeup*() routine alerts any process waiting for the transfer to complete. It then checks to see if the device is idle and, if so, looks for more work to start up. Therefore, in the case of output to a terminal, the interrupt routine looks for more work in the *clists* each time a transfer completes.

8.6.3 Character List and Character Block Architecture

The character lists, or *clists*, provide a general character buffering system for use by character device drivers. The mechanism is designed for buffering small amounts of data from relatively slow devices, particularly terminals.

The XENIX kernel has a collection of character blocks called *cblocks*. Each *cblock* contains a link to the next *cblock* and an array of characters. A *clist* is a linked list queue of *cblocks*.

The kernel provides the *getc*() and *putc*() routines, described previously, for putting characters into a *clist* and for removing characters from a *clist*. These routines can be used by all drivers using *clists*. Note that the routines are not the same as the Standard I/O Library routines of the same names.

The static buffer header for each *clist* contains three fields: a count of the number of characters in the list, a pointer to the first character in the list, and a pointer to the last character. The *clist* buffers form a singly linked list as shown below:

```
struct {
int c_cc;
char *c;-------+
char *c_cl;--+ |
} clist;     | |
             | |
             | |   +------+    +------+     +-----+
             | |   | next |--->| next |---->| 0   |
             | |   +------+    +------+     +-----+
             | +-->|      |    |      |     |     |
             |     |      |    |      |     |     |
             |     |chars |    |chars |     |chars|
             |     |      |    |      |     |     |
             |     |      |    |      | +-->|     |
             |     |      |    |      | |   |     |
             |     +------+    +------+ |   +-----+
             |                          |
             +--------------------------+
```

Character List Buffers

All currently unused *cblocks* are kept on a list of free memory blocks. Since there are a limited number of *cblocks* drivers should follow a protocol to prevent a particular process from consuming all available resources.

For output buffering, the driver usually follows a "high- and low-water mark" convention. The driver accepts and queues requests from the user process until the corresponding clist has reached its high-water mark. At that point, the requesting process is suspended via *sleep*( ). When the buffer has drained below the low-water mark, the suspend process is awakened. Two constants for the high- and low-water marks, TTHIWAT and TTLOWAT, are defined in the file */usr/sys/h/tty.h*.

For input buffering, the driver usually buffers the data up to some limit. When this limit is reached, data is discarded to make room for the more recent data.

8.6.4 Terminal Device Drivers

Terminal device drivers normally use *clists* extensively. The terminal device driver should declare one *tty* structure for each terminal line (minor device number). Each *tty* structure contains the static clist headers for three clists. These *clists* are the "raw queue," the "canonical queue," and the "output queue."

When a process writes data to a terminal device, the task-time part of the driver puts the data into the output queue, and the interrupt routine transfers it from the queue to the device.

When a process requests a read of data from the terminal, the situation is slightly more complicated. This is because XENIX provides for processing of characters on input at the option of the requesting process. For example, in normal input the backspace key is interpreted as "delete the last character input," and the line kill character means "delete the whole current line." Certain special characters, such as BACKSPACE must be treated in context, since they depend upon surrounding characters. To handle this, XENIX drivers use two queues for incoming data.

The two queues are the *raw queue* and the *canonical queue*. Data received by the interrupt routine is placed in the *raw queue* with no data processing. At task-time, the driver decides how much processing to do. The user process has the option of requesting *raw* input, where it receives data directly from the *raw queue*. *Cooked*, the opposite of *raw*, input refers to the input after processing for ERASE, LINE KILL, DELETE, and other special treatment. In this case, a task time routine, *canon*(), is used to transfer data from the *raw queue* to the canonical queue. This performs BACKSPACE and LINE KILL functions, according to the options set by the process using the *ioctl*(S) system call.

While the structure of input *clists* may be important as background information, the driver does not need to manipulate the input *clists* directly. In XENIX System V, the direct *clist* processing for *tty* device drivers is normally handled by the specific line discipline. The only processing that the device driver needs to perform is interrupt-level control. The device driver provides interrupt-level control by emptying and filling structures called character control blocks (*ccblock*). Each *tty* structure has a *ccblock* for transmitter (*t_tbuf*) control and a *ccblock* for receiver (*t_rbuf*) control. The *ccblock* structure has the following format:

```
struct ccblock {
    caddr_t  c_ptr;          /*buffer address*/
    ushort_t c_count;        /*free character count */
    ushort_t c_size;         /*buffer size*/
};
```

At receiver interrupt time, the driver fills a receiver ccblock with characters, decrements the character count, and calls the line discipline routine *l_input*( ). At transmitter interrupt time, the driver calls *xxproc*() and the line discipline routine, *l_output*( ), to get a transmitter *ccblock* and then outputs as many characters as possible. Refer to Chapter 9, "Sample Device Drivers," for code examples.

The basic flow of data through the system during terminal I/O is shown in the diagram below:

```
XENIX    | DRIVER      |                        TASK TIME #INTERRUPT
KERNEL   |             |                                  # TIME
         |             |                                  #
  <---   |-<--     <-  |---------< l_read()               #
read()   | xxread()    |              |                   #
system   |             |     +---->----+----<----+        #
call     |             |     |                   |        #
         |             | +-----+            +-----+       #
         |             | |     |            |     |       #
         |             | |canon|            | raw |       #
         |             | |queue|<- canon() <-|queue|<-#receive
         |             | |     |            |     |   #routine
         |             | |     |            |     |   #l_input()
         |             | +-----+            +-----+       #
         |             |                                  #
write()  |             |                                  #
system   |             |                   +------+       #
call     |             |                   |      |       #
  --->   |->--     ->- |---> l_write() --> |output|--->#transmit
         | xxwrite()   |                   |queue |    #routine
         |             |                   |      |    #   xxproc() and
         |             |                   |      |    #   l_output()
         |             |                   +------+    #
```

Data Flow For Terminal Device Drivers

There are two slight complications to the scheme presented in the diagram above. These are output character expansion and input character echo. Output expansion occurs for a few special characters. In the cooked mode, tabs may be expanded into spaces, and the newline character mapped into carriage return plus line feed. There is a facility for producing escape sequences for uppercase terminals and delay periods for certain characters on slow terminals. Note that all these are simple expansions, or mapping single characters, and so do not require a second list as in the case for input. Instead, all the expansion is performed by the *xxproc*( ) routine before placing the characters in the output *clist*.

Character echo is a user process option required by most processes. With this, all input characters are immediately echoed to the output stream without waiting for the user process to be scheduled. Character expansion is performed for echoed characters, as for regular output. Character echo takes place at interrupt time so that a user entering text at a terminal gets fast echo, regardless of whether his program is in memory and running or swapped out on disk.

8.6.5 Other Character Devices

There are three character devices commonly used with XENIX systems: terminals, lineprinters, and magnetic tape drivers. Terminals receive special attention in the XENIX system. Lineprinters and magnetic tape tend to use existing kernel facilities, with little special handling.

Lineprinters

Usually, these are relatively slow, character-oriented devices. The drivers use the *clist* mechanism for buffering data. However, a lineprinter driver is generally simpler than a terminal driver because less processing of output characters is necessary and no processing of input is necessary.

Magnetic Tape Drivers

Magnetic tape is a special case. The data is arranged on the physical medium in blocks, as on a disk. However, it is almost always accessed serially. Furthermore, there is generally only one program accessing a tape drive at a time. Thus, the elaborate kernel buffer management scheme in XENIX (which is designed to optimize disk access when several processes are making simultaneous requests to different parts of the same disk) is not applicable to tapes. The *clist* mechanism is not appropriate either, because of the large amount of data involved.

Usually tape drivers provide two interfaces, block and character. The character interface is used for physical I/O directly between the device and the user process' address space. The block interface makes use of the XENIX kernel buffers and buffer manipulation routines to store data in transit between device and process. Refer to Section 8.7.1, "Character Interface to Block Devices," for information on providing the facility for physical I/O.

8.7 Device Drivers for Block Devices

Block devices are those that may be addressed as randomly accessible fixed size records, rather than individual bytes. Disks fall into this category, as do some magnetic tape systems. XENIX file systems always reside on block devices. However, block devices do not have to be used in this way.

Unlike character device transfers, a block I/O transfer request is not a private transaction between a driver and a user process. The XENIX kernel provides a comprehensive buffer management scheme which is used by block device drivers.

The XENIX kernel maintains a collection of buffers, and keeps track of what data is in them and whether a block is "dirty", that is, has been modified and so needs to be written out to disk. When a user process issues a transfer request to a block device, the kernel buffer routines check the buffers to see if the data is already in memory. If the data is not in memory, a request is passed to the driver to get the data.

Large requests are normally broken down into BSIZE blocks and handled individually, regardless of the size of the process's I/O request, since some may be in memory and some may not.

When a process issues a read request, this generally translates into one or more disk blocks. The kernel checks which of these is already in memory and requests that the driver get the rest. The data from each buffer filled by the driver is copied into the process' memory by the kernel.

In the case of a write request, the kernel copies the data from the user process' memory into the kernel's buffers. If there are insufficient free buffers, the kernel has the driver write some out to disk, using a selection algorithm designed to reduce disk traffic. When all the data is copied out of user space, the kernel can reschedule the process. Note that all the data may not yet be out on disk; some may be in memory buffers and marked as needing to be written out at some later time.

Swap requests are passed directly to the driver without being broken down into BSIZE blocks. This means that if a device is capable of being swapped upon, its driver must be capable of handling transfers of arbitrary multiples of BSIZE.

8.7.1 Character Interface to Block Devices

Sometimes block device drivers provide a character I/O interface as well as one for block I/O. In this case, a separate special device file can be created to access the device through the character interface.

To construct a character I/O interface to a block device, use the utility **mknod(C)**, described in the XENIX *Reference*, to create a character special device file that has the major and minor number as the block special file for this device. Use the **-c** argument of **configure** to enter the routine names within the driver into the configuration files, the same as for a normal character device.

The block device driver must provide the routines *xxread*() and *xxwrite*() described below to implement character I/O.

When a block device is accessed through a character interface, data transfer takes place directly between the device and the process' memory space. There is no intermediate buffering in the kernel buffers or the *clists*. This processes is also referred to as *raw* I/O and in this context, is different from *raw tty* I/O.

The driver receives the request exactly as the process sent it, for whatever size was specified. There is no kernel support to break the job into BSIZE blocks. It has some advantages for certain types of programs.

Programs that need to read or write an entire device can usually do this more efficiently through the character interface, since the device can be accessed sequentially and large transfers can be made. There is also less copying of data between buffers than is used in the block interface. Thus filesystem backup programs, or utilities that copy entire volumes, typically operate through this interface.

The cost of this extra efficiency is that the process has to be locked in memory during the transfer, since the driver has to know where to read and write the data. Tbe routine *physio*(), called by the *xxread*() and *xxwrite*() driver routines, handles locking the process in core for the duration of the data transfer.

8.7.2 Block Device Driver Routines

A block device appears to the kernel as a randomly addressable set of records of size BSIZE, where BSIZE is a manifest constant defined in the */usr/sys/h/param.h* file. The XENIX kernel inserts a layer of buffering software between user requests for block devices and the device driver. This buffering improves system performance by acting as a cache, allowing read ahead and write behind on block devices.

Each buffer in the cache contains an area for BSIZE bytes of data and has associated with it a header of type struct *buf* which contains information about the data in the buffer. When an I/O request is passed to the task time portion of the block device driver, all of the information needed to handle the data transfer request has been stored in the buffer header. This information includes the disk address and whether a read or a write is to be done. The file */usr/sys/h/buf.h* describes the fields in the buffer header. The fields most relevant to the device driver are:

b_dev	the major and minor numbers of the device
b_bcount	the number of bytes to transfer
b_paddr	the physical address of the buffer
b_blkno	the block number on the device
b_error	set if an error occurred during the transfer

The driver validates the transfer parameters in the buffer header and then queues the buffer on a double linked list of pending requests. In each block device driver, this chain of requests is pointed to by a header of type struct *iobuf* named *xxtab*. The file */usr/sys/h/iobuf.h* describes the fields in the request queue header. The requests in the list are kept sorted using the *disksort*() routine. The device interrupt routine takes its work from this list.

When a transfer request is placed in the list, the process making the request sleeps until the transfer is completed. When the process is awakened, the driver checks the status information from the device interrupt routine, and if the transfer is completed successfully, returns a success code to the kernel. The kernel buffer routines are responsible for correlating the completion of an individual buffer transfer with particular user process requests.

The interface between the kernel and the block device driver consists of these kinds of routines:

- Block Interface Routines Defined Within the Driver
- Character Interface Routines Defined Within the Driver
- Character Interface Support Routines
- Block Interface Support Routines

Block Interface Routines Defined Within the Driver

xxinit ()

> **Purpose:** This routine is called to initialize the device when XENIX is first booted. If present, it is called indirectly through the *dinitsw*[ ] table during early initialization.

xxopen (dev, flag, id)

Purpose: This routine is called each time the device is opened. It is the responsibility of this routine to initialize the device and perform any error or protection checking.

Parameters: *dev* is an integer that specifies the device number.

flag is the mode in which the file should be opened. It is the bitwise "or" of the modes defined in */usr/sys/h/file.h*. Note that this *flag* is similar to, though not exactly the same as the *oflag* argument that is passed to the *open*() system call.

id is an integer that specifies whether the device is a character device (0) or a block device (1).

If this routine sets *u.u_error* to a non-zero value, the *open* has failed and the value in u.u_error will be returned to the user as *errno*.

xxclose (dev, flag, id)

Purpose: This routine is called on the last close on a device. It is responsible for any cleanup that may be required, such as disabling interrupts, clearing device registers, ejecting media, and so on.

Parameters: *dev* specifies the device number of the device being closed.

flag is the mode in which the file should be opened. It is the bitwise "or" of the modes defined in */usr/sys/h/file.h*. Again, *flag* is similar to, though not exactly the same as the *oflag* argument that is passed to the *open*( ) system call.

id is an integer that specifies whether the device is a character device (0) or a block device (1).

xxstrategy (bp)

Purpose: This routine is called by the kernel to queue an I/O request. It must make sure the request is for a valid block, and then must insert the request into the queue. Usually the driver calls *disksort*( ) to insert the request into the queue. The *disksort*() routine takes two arguments: a pointer to the head of the queue, and a pointer to the buffer header to be inserted.

Parameters: *bp* is a pointer to a buffer header.

xxstart ()

Purpose: If the task time portion of the driver detects that the device is idle, this routine may start it. It is often called by both task time and interrupt time parts of the driver. It checks whether the device is ready to accept another transfer request and, if so, starts it up, usually by sending it a control word. The rest of the kernel does not know if a start routine is present; there are no indirect start routine entry points.

xxintr (vec_num)

Purpose: This routine is called whenever the device issues an interrupt. Depending on the meaning of the interrupt, it may mark the current request as complete, start the next request, continue the current request, or retry a failed operation. The routine examines the device status information and determines whether the request was successful. The block buffer header is updated to reflect this. The interrupt routine checks to see if the device is idle and, if so, starts it up before exiting.

Note that *xxintr* will be called at the *spl* level specified in *master* (see section 8.9.2). No interrupts from the same device will be acknowledged unless the interrupt routine explicitly lowers its *spl* level. Interrupt routines do not normally lower their *spl* level.

Parameters: *vec_num* is an integer that specifies the interrupt vector number.

xxpoll (ps)

Purpose: This routine, if present, is called by the system clock at *spl6*( ) during every clock tick. It is useful for repriming devices that constantly lose interrupts.

Parameters: *ps* is an integer that indicates the previous process's priority when it was interrupted by the system clock. The macro USERMODE(*ps*), defined in */usr/sys/h/param.h,* can be used to determine if the interrupted process was executing in user mode.

xxhalt ()

Purpose: This routine, if present, is called when the system is shut down.

Character Interface Routines Defined Within the Driver

xxread (dev)

Purpose: The only action taken by this routine is to call *physio*( ) with the appropriate arguments.

Parameters: *dev* specifies the minor device number of the device.

Note: Often a block device driver provides a character device driver interface so that the device can be accessed without going through the structuring and buffering imposed by the kernel's block device interface. For example, a program might wish to read magnetic tape records of arbitrary size or read large portions of a disk directly. When a block device is referenced through the character device interface, it is called raw I/O to emphasize the unstructured nature of the action. Adding the character device interface to a block device requires the *xxread*() and *xxwrite*() routines.

xxwrite (dev)

Purpose: The only action taken by this routine is to call *physio*( ) with appropriate arguments.

Parameters: *dev* specifies the device number of the device.

Note: See Note for *xxread*() routine.

xxioctl (dev, cmd, arg, mode)

Purpose: This routine is called by the kernel when a user process makes an *ioctl*() system call for the specified device. It performs hardware dependent functions such as parking the heads of a hard disk, setting a variable to indicate that the driver is to format the disk, or telling the driver to eject the media when the close routine is called.

If this routine sets *u.u_error* to a non-zero value, the *ioctl* has failed and the value in u.u_error will be returned to the user as *errno*.

Parameters: *dev* specifies the minor number of the device.

cmd specifies the command that was passed to the *ioctl*() system call.

arg specifies the argument that was passed to the *ioctl*() system call.

flag is the mode in which the file should be opened. It is the bitwise "or" of the modes defined in */usr/sys/h/file.h*. Again, *flag* is similar to, though not exactly the same as the *oflag* argument that is passed to the *open*() system call. *mode* specifies the flags that were set on the *open*() system call for the specified device.

Character Interface Support Routines

physio (bs,bp,dev,flag)

Purpose: This routine provides the raw I/O interface for block device drivers. It validates the request, builds a buffer header, locks the process in core, and calls the strategy routine to queue the request.

By default, *physio*( ) assumes that the driver can only transfer multiples of BSIZE bytes, and checks to ensure that the transfer is a BSIZE multiple. *physio*( ) also assumes that the driver is capable of handling a transfer that crosses a 64k physical memory boundary. The fourth argument to *physio*( ) may contain extra flags to override these default assumptions.

Parameters: *bs* is a pointer to the strategy routine for the block device.

bp is a pointer to the buffer header to be used for this request.

dev is the device number of the device.

flag must contain either B_READ, to specify a read operation, or B_WRITE, to specify a write operation. Two additional bits may be or'ed in with B_READ or B_WRITE: B_TAPE, to disable the error checking for a BSIZE-multiple transfer size, and B_NOCROSS, to force *physio*() to divide transfers that cross a 64k physical memory boundary. These constants are defined in */usr/sys/h/buf.h* and */usr/sys/h/iobuf.h.*

Block Interface Support Routines

disksort (disktab,bp)

Purpose: This routine adds a block device I/O request to the queue of such requests for a particular device. It is normally called by the device strategy routine. The queue of requests is sorted in ascending order based on *bp*'s *b_cylin* field (type *ushort*), in an attempt to reduce disk head movement.

Parameters: The *disktab* parameter is the head of the request queue. and is the address of a *struct iobuf* declared within the driver.

bp is a *struct buf* * pointing to the I/O request to be added to the queue.

getablk (flag)

Purpose: This routine acquires a free buffer from the block buffer pool. The pointer returned by this routine addresses a buffer which can be used as required. The buffer can subsequently be returned to the buffer pool by calling *brelse*( ) or *iodone*().

Parameters: *flag* specifies whether the routine should acquire any buffer or a directly addressable buffer (no far pointers are needed). If *flag*==1, the routine will acquire a directly addressable buffer; if *flag*==2, the routine will acquire a far buffer, and if *flag*==0, it will acquire any buffer.

Result: The routine returns a *(struct buf *)* which addresses the allocated buffer. The struct *buf*'s *b_paddr* field contains the physical address of a BSIZE buffer. Depending on the value of *flag*, this may or may not be within the kernel near data segment.

Warnings: There are very few directly addressable buffers in tbe XENIX kernel. Most are already allocated for other functions. If a directly addressable buffer is required, the value of NSABUF may have to be increased using **configure**(C).

This routine should not be used at interrupt time as it may call the *sleep()* routine.

iowait (bp)

Purpose: This routine is called by the higher levels of the kernel I/O system in order to wait for the completion of an I/O operation specified by the buffer addressed by the parameter *bp*. This routine is usually used in conjunction with an interrupt routine that calls the *iodone()* routine. *iowait* should not be called at interrupt time since it may call the *sleep*() routine.

Parameters: *bp* specifies a struct buf * which addresses the buffer involved in the I/O operation.

Result: There is no result returned. The calling process will be allowed to proceed once the I/O operation has been completed.

iodone (bp)

Purpose: This routine signals completion of an I/O operation involving the buffer addressed by *bp*. This routine is called when the driver wishes to signal either successful or erroneous completion of an I/O operation. It differs from the *brelse*() routine in that the higher levels of the kernel I/O system will complete the processing of the buffer before releasing it back to the buffer pool using *brelse*(). This routine is usually used by an interrupt routine to wake up a process that has slept using the *iowait()* routine.

Parameters: *bp* specifies a *struct buf* * which addresses the buffer.

brelse (bp)

Purpose: This routine releases a block buffer to the free pool of buffers. It is called by a block device driver to release a buffer. The contents of the buffer are lost and the driver is not allowed to make any further reference to the buffer.

Parameters: *bp* is a *struct buf* * that addresses the buffer header relating to the buffer to be released.

Result: The buffer addressed by *bp* is returned to the free buffer pool. No errors are possible.

deverr (dp, o1, o2, dn)

Purpose: This routine prints an error message on the system console together with some device specific information passed as parameters to the routine. The exact format of the output is shown in the following *printf* statement:

```
register struct buf *bp;

bp=dp->b_actf;
printf("error on dev %s (%u/%u)",
    dn,
    major(bp->b_dev),
    minor(bp->b_dev));
printf(", block=%D cmd=%x status=%x\n",
    bp->b_blkno,
    o1, o2);
```

Parameters: *dp* is a *struct iobuf* * which is the head of the I/O request queue for the device.

o1 contains driver specific information. It is normally used to provide the controller command which relates to the I/O operation which failed.

o2 contains driver specific information. It is normally used to provide the controller status information at the time of failure.

dn is a char * that points to a short name for the device.

8.8 Compilation, System Configuration, and Kernel Linkage

To make your driver source code part of the XENIX kernel, you first compile your driver in the same way that the rest of the kernel is compiled.

Next, make the various routine names and driver attributes of your driver accessible to the XENIX kernel using the **configure** utility. This program creates several multi-dimensional tables of routine names and driver attributes.

Then, link the kernel by adding the new module to the **ld**(C) command line in the *makefile* provided, and running **make**(CP).

8.8.1 Compiling Device Drivers

Use the XENIX C compiler to compile C source code, or the assembler to create an object module from assembler source. Use the **cc**(CP) or **masm**(CP) commands.

The **cc** command line must contain the following switches:

-K	Disable stack probes.
-DM_KERNEL	Required for conditional code in standard header files

It should also contain ONE of the following:

-M2em	For 80286 processors. Enables 80286 instructions, **near**, and **far** keywords. Compiles middle model, to conform with the kernel program model.
-M0em	For 8086 processors. Uses only 8086 instructions. Enables **near** and **far** keywords. Compiles middle model, to conform with the kernel program model.

For device driver subroutines written in assembly language, the **masm** command line should contain the following switch:

-Mx	Preserves lower case in output. Required for the linker to be able to resolve external declarations to C functions.

An appropriate **cc** or **masm** command line will produce a corresponding ".o" module. For example, *scsi.c* becomes the object module *scsi.o*.

8.8.2 System Configuration

System configuration is the process of placing references to your driver's main functions in various tables. Since the existing parts of the kernel do not know what the functions in your new driver are called, driver functions are referenced by indirect calls into the configuration tables.

The **configure** utility generates and assembles the *c.asm* and *space.inc* source modules that contain these indirect function references. The generated *space.inc* file is one of the header files that is inserted into *space.asm* before that module is assembled into *space.o*. The *c.asm* file is assembled into the object module *c.o*. Both object modules are then linked into the kernel.

Some older "preconfigured" drivers did not require **configure** to be run, as all function references were already in place. For the rest, composing the driver's configuration command is discussed in **configure**(C), *Using the Link Kit*, and briefly, below. Though it may seem easier to edit the C and assembly language configuration files directly, **configure** insulates you from potential changes to the configuration files, and allows you to use the same procedure to configure your driver as the end-user who receives your driver in binary form.

Preconfigured Drivers

Earlier releases made provisions for several common types of drivers by providing null routines that were linked in if the corresponding drivers were not present. Drivers for which preconfiguration was provided will still link in as before. Simply add the name of the driver's ".o " module to the **ld** command line, as before. The earlier scheme of patching the interrupt vector within *vecintsw* at driver initialization time should still be used.

Determining the Vector Number

You must determine your interrupt vector number so you can inform the kernel that your driver should be called when an interrupt is pending on that vector. This information is highly machine and configuration dependent.

In the unmapped kernel, there is only one programmable interrupt controller. Your XENIX vector number is the same as the bus request number on which your peripheral interrupts.

For the mapped kernel, your peripheral device can interrupt on one of the request lines of either a master interrupt controller or single slave interrupt controller, which is connected to master request line 2. Your XENIX vector number does not correspond directly to the bus request numbers. Instead, it is mapped to logical vector numbers which allow for the presence of slave interrupt controllers connected to the main one.

The index of the appropriate vector is determined as follows:

1. If the vector used is on the master controller, just use the vector number directly.
2. If it is on a slave controller (only one is currently supported for the 80286, on master request line 2), take the request line which that controller uses in the master controller, multiply that number by 8 (there are 8 vectors per controller), add 8 (for the master which uses logical vector numbers 0-7), and add the number of the request line used on the slave controller.

For example, take the case of the IBM AT and compatibles: the bus contains request lines IRQ0-7 which interrupt on the master controller (except IRQ2, which is connected to the slave controller), and lines IRQ8-15, which are connected to request lines 0-7 of the slave controller. Actually, only lines IRQ3-7, 9-12, 14, and 15 are on the bus.

Vector Manipulation for Preconfigured Device Drivers

In preconfigured drivers, entry points have been provided for all necessary routines (open, close, etc.) except for the interrupt handler. This must be patched in at system start-up time as follows: an extra entry point has been provided for each of the drivers expected to require an interrupt vector. This entry point's suffix is "-init". This function must replace the appropriate vector in the *vecintsw*[ ] table with a pointer to the interrupt handler function for the particular device driver. This structure is declared in the file */usr/sys/h/conf.h*, and an example of its usage is in *c.c*.

In addition to patching the *vecintsw*[ ] table, the driver *init* routine should patch the *vecintlev*[ ] table. This table specifies the priority or *spl* level of the driver. Most drivers are priority level *spl5*.

An example driver fragment for a hypothetical *spl5* driver "xx" using bus vector 12 is shown below:

```
#include "../h/param.h"
#include "../h/conf.h"
   .
   .
   .
xxintr()
{
   .
   .
   .
}

#define  NUM  (8 + (2*8) + (12-8))

int (*xxoldintr())();

xxinit()
{
  xxoldintr = vecintsw[NUM];
  vecintsw[NUM]  = xxintr;
  vecintlev[NUM] = 5;
   .
   .
   .
  /* perhaps some other one-time driver-local initialization */
}
```

It should be noted that this may not be applicable for all device drivers under all circumstances.

See section 8.10.3 "Vector Collision Considerations" for more information on the selection of interrupt vectors.

Using configure

Before **configure** can be run, you need to know an unused major device number for your device, the vector or vectors on which your device interrupts, and the list of routines in your driver that must be added to the configuration tables.

The **configure** utility enforces the rules that all routines in the driver begin with a common prefix and that the prefix be between 2 and 4 letters long. If your driver prefix is incorrect or inconsistent, change it.

Please read **configure**(C) and *Using the Link Kit,* a chapter in the XENIX User's Guide. *Using the Link Kit* contains a detailed description of how to create a **configure** command line from a driver binary. Authors of drivers have an advantage in that they do not have to discover the names of the routines; the names that must be presented to **configure** are those chosen for the routines that have so far been described, such as the name of a character driver's *write* routine. Maintain a backup copy of the *master* and *xenixconf* files while learning to use **configure**: if you make a mistake you can restore the old files.

Also note that **configure** requires that block drivers have a *tab* structure, and indeed, the vast majority of block drivers do. If you are writing a non-interrupting block driver, simply declare a *struct iobuf xxtab* within your driver.

8.8.3 Linking The Kernel

Your Link Kit contains a *makefile* for linking the kernel. The reference to the new driver should be placed in this file on the **ld** command line prior to any of the object library references (the pairs of options of the form **-l** *lib_xxx*), and following all other object file references. That is, your object file must follow *KMseg.o*, *oemsup.o*, *c.o*, and any other files already on the command line.

For binary distribution, also edit the file *conf/link_xenix* to **link** the new driver in with the kernel. Here too, place the reference to the new driver on the **ld** command line prior to any of the object library references and following all other object file references.

For preconfigured drivers, the **ld** command will find the actual driver first and thus stop looking for it in the libraries, which contain the null routines that are normally linked in.

To link your driver, enter:

```
make
```

link_xenix is what the end user uses to link your driver into the kernel. The end user may not have purchased the XENIX development system and therefore may not have **make**(CP). However, **ld**(C) comes with every system, and no special utility is required to run shell scripts.

Once you have a new XENIX kernel, back up the old one, by typing these or similar commands:

```
# cp ./xenix /xenix.new
```

The new XENIX kernel must be in the / directory. Note that in some versions, a kernel must be one of the first 64 entries in the / directory for **boot** to find it.

8.9 Driver Debugging

The following sections contain information on getting a driver to run, and what to look for if it doesn't.

8.9.1 Booting the New Kernel

Halt your system by entering:

```
# /etc/haltsys
```

You see the "** Normal System Shutdown **" message. Press return to see the boot prompt:

```
Boot
:
```

If you press RETURN, or simply do nothing, the default operating system image is loaded and started. In order for the bootstrap program to locate and load any newly installed device drivers, it must be told to read the */xenix.new* file, which contains the kernel that includes the device driver. To boot the new kernel, enter, at the boot prompt:

```
xenix.new
```

The system will boot up with the "new" kernel.

8.9.2 General Debugging Hints

Debugging a device driver is more an art than a science. This section touches on some of the more useful techniques to try if your driver isn't working.

1. Make sure that you are actually talking to your driver.

   If you get errors such as "no such device" (ENODEV) when you try to access your driver, you might have a problem with either the device node itself, or with your driver configuration, as spelled out in the Link Kit configuration file *c.c*.

   Check your major device number correspondence. Make sure that the major device number and type of the device node you are trying to access corresponds with the appropriate line in the *_bdevsw* or *_cdevsw* array in *c.asm*. For example, you might have written a new printer driver, whose major device number is 6, whose name is "pa", and whose type is "character".

First, check the device node, to make sure that /dev/pa is a character special device, whose **ls -l** listing is along the following lines:

```
crw-rw-rw- 1 root    6,  0 Apr 29 19:56 /dev/pa
```

Then, check *c.asm*, to make sure that there is a set of functions in the *_cdevsw* table that have the "pa" prefix. The line will contain something like:

```
DW $CFG_C6
DD _paopen
DD _paclose
DD _nulldev
DD _pawrite
DD _paioctl
DW 00H
DW 00H
```

The parallel driver has only the routines paopen, paclose, pawrite and paioctl as part of the *_cdevsw* table. Other drivers might have **read**, **_tty**, or in the future, **stream** entries.

Also check the constants *_cdevcnt* and*_cdevmax*, (*_bdevcnt* and*_bdevmax* for a block driver) to make sure they are at least 1 greater than the major number of your driver.

If this correspondence does not hold, revert back to the older **master** and **xenixconf** files and rerun **configure**.

2. Make sure your device registers are where you think they ere.

   The effect of accessing a nonexistent port address varies from machine to machine, but, for example, on the IBM XT or AT you can read values using *inb*() from nonexistent hardware and receive no error code, just a random value.

   Since at least some of the I/O ports on most peripheral controllers are both read and write, you should make sure you can write to one of your device's registers using *outb*(), then read back the value you've written using *inb*(). Even when none of the registers are read/write, as is true on some mouse controllers, you can at least read from one of the status registers using *inb*(), and make sure that the result is reasonable.

3. Work towards getting simple I/O from the driver first, complex I/O later.

   Character devices are usually easier to write to than to read from. For a serial or a printer driver, your first test will probably be to echo "hello, world" to the device, or something equally simple and traditional.

   Block devices are generally easier to read from than to write to, since you have to read back the block you've written to know if you've written it successfully. Many block devices have a "get drive parameters" command, or something similar, which is even more basic than either reading or writing.

4. Use kernel *printf*( ) statements for debugging.

   Although you shouldn't overuse *printf*( ) (in a finished driver, *printf*( ) should only be used for unrecoverable errors), it can be an invaluable debugging tool. Coupled with #ifdef DEBUGs and a global "debug level" flag, you can tailor the verbosity of your debug output to the situation at hand.

   For example, you may have two debug levels:

```
#ifdef DEBUG
 if ( mydebugflg > 0 )
  printf("got to myopen\n");
 if ( mydebugflg > 2 )
   printf("open parameters: dev=%x, flag=%x bc=%x",dev,flag,bc);
#endif
```

   There are occasional situations where a *printf*( ) can change peripheral timing enough to make a difference, but these cases are fairly rare.

5. Use *getchar*( ) to stop kernel output and to set debug levels.

   Kernel *getchar*( ) is similar, though not quite the same, as the standard I/O library routine of the same name. Kernel *getchar*() returns a single character from the keyboard. The character is automatically echoed. The only other processing done on this character is to map RETURN to RETURN/LINE FEED on output. When you have many lines of kernel *printf*() output, inserting *getch r*() statements into your driver is one of the better ways to regulate the *printf*() output flow.

   A second use of *getchar* is to set the level of debugging. For example, in the example above, you could place two lines of code such as:

```
    mydebugflg = getchar();
    mydebugflg -= '0';
```

shortly after the beginning of the open routine, to set the current value of *mydebugflg* to anywhere between 0 and 9.

Note that *getchar*( ) may not work at interrupt time for interrupt routines of certain priorities.

6. Poll before you use interrupts.

   Often the hardest driver routine to get right is the interrupt routine. You can ease this process along a bit by first writing a polled driver: one that busy-waits until the request you made has completed, and then returns status. However, do not leave any busy-wait loops in the finished driver!

   Polled drivers are best first approximations for block devices such as disks. For serial drivers, a polled interface may help you decide how to write to the device. However, be forewarned that performing polled reads will make the system unusably slow.

7. Use *spl{5,7}*() as a debugging aid.

   Sometimes, a driver can be difficult to debug because higher priority interrupts get in the way. A call to *spl7*( ) will shield you from any interruptions by the other devices on the system.

8. Be patient.

   Drivers are complex. So much so, that writing a 300-line device driver takes even an experienced driver-writer several times longer than a utility program of the same length. Don't worry if your driver takes a while to perfect.

8.9.3 Vector Collision Considerations

When designing a device driver to work with XENIX, care should be taken in the selection of the hardware interrupt vector. This is because of the possibility of conflict between device drivers over interrupt vector usage.

8086-based XENIX systems use only one 8259 programmable interrupt controller. Of the 8 vectors available, only vector 2 (bus lead IRQ2) is not currently used. It is appropriate for devices whose drivers are written using *spl5*( ).

80286-based XENIX systems use 2 8259 programmable interrupt controllers. The mapped kernel currently leaves only vectors 9-12 and 15 (bus leads IRQ9-12 and IRQ15) unused. These vectors are also safe to use for devices whose drivers are written using *spl5*( ).

If it is necessary to use one of the other vectors, there are two configuration alternatives:

1. Replace the device driver already using the vector.

2. Provide a special-purpose interrupt handler that "knows" that the vector is shared and takes appropriate precautions.

The first alternative is recommended, but is not always possible. There are problems with the second alternative, because there is no way to prevent the loss of interrupts which can occur when competing with an arbitrary device.

The problem is that the 8259 interrupt controller detects an interrupt request only when the request line changes state from off to on (called *edge-triggered* mode). If all sources for the interrupt request line are not off at the same time after entry to the interrupt service routine, no further *rising edge* on the request signal is detected, and so no more interrupts are seen on that vector until all the sources for the interrupt request line are turned off. The state of the interrupt request line cannot be determined directly from the interrupt controller chip, so the determination must be made by device-specific means for all devices sharing the vector.

However, cohabitation is possible for those devices that interrupt only following a request from the CPU. Disk drivers, tape drivers, and other such devices can "time out", using the *timeout*() function, when waiting for a response to a request, and, upon time out, examine the device to determine if the operation is complete. This approach saves your driver from lost interrupts, but the device with which you share a vector is only immune if it is written using *timeout*( ) as well.

This approach is far from practical for use with devices such as serial communication lines, which can cause interrupts at any time, out of the control of the system using the device. The granularity of control available with *timeout*( ) is far too slow for all but the slowest of communication lines (approximately 110 to 200 baud).

This does not mean, however, that each serial line requires its own interrupt vector. Some serial boards provide enough pollable state information to allow the serial interrupt routine to loop until none of its controlled devices is posting an interrupt. In this example, the key is that a single interrupt routine controls all of the multiple devices on a single vector.

8.9.4 Note on ps

If you change to an alternate name for your kernel, such as *xenix.new*, ps(C) does not work correctly unless you specify the **-n** flag and the pathname of the XENIX kernel you are using.

See ps(C) in the XENIX *Reference* for more information.

8.10 Notes On Preparing a Driver for Binary Distribution

8.10.1 Naming Guidelines

The 2-4 letter name that prefixes all of your driver's routines should describe what kind of a driver it is, as best as is possible in such limited space. For example, the current serial I/O driver uses routines beginning with "sio", and the parallel driver uses routines beginning with "pa".

Preconfigured drivers have had their names reserved in advance. If you are writing a driver for a device that a user might have more than one of, such as an add-on hard disk driver, you might want to be a bit more obscure to prevent later naming conflict. For example, the driver for a TechnoBabble hard disk might begin its routines with the prefix "tbhd".

8.10.2 Style Issues for User Prompting

Most currently configured XENIX devices print out a short message in their initialization routines to notify the user that they are installed. This message must be terse. All the extra drivers that a user could possibly want, combined, should not generate enough messages to scroll the boot-up copyright message off the screen.

For example, this is an appropriate message:

```
2 phasers, 4 photon torpedoes
```

8.10.3 Insulating Drivers Against Configuration Changes

Do not write a driver that relies on particular configuration parameters, for example a certain major device number or interrupt vector. Avoiding such "hardcoded" assumptions helps prevent collisions with other drivers, and insulates the driver from system configuration changes.

Drivers should not, and do not need to be aware of their own major device number. In System V, a driver's major device number is no longer passed to it as a parameter.

Very few drivers have ever needed to know this information, but those that did fell into two categories: drivers performing some form of physical I/O that used the major device number to determine the type of I/O, and block device drivers that needed to know if the device they controlled was the root or the swap device.

Drivers doing physical I/O now differentiate it either by using the block/character parameter of the combined open routine, or by marking the transfer in the b_dev field of the transfer's buffer. Drivers needing to know if they are the root device can find out using the following or something similar:

```
#include "../h/conf.h"

extern struct bdevsw bdevsw[ ];

if ( bdevsw[major(rootdev)].d_open == xxopen ) {
  printf("the xx driver is the root device\n");
      .
      .
      .
}
```

Drivers also should not and do not need to know the vector on which they interrupt. The underlying hardware determines the vectors on which a device is capable of interrupting. When the hardware is only capable of interrupting on one vector, there is little a driver writer can do beyond the timeout schemes discussed in section 8.9.3. If the vector is configurable on the card, some cards allow you to query the vector number directly. An unused vector has a *vecintsw*[] entry of *novec*.

Preconfigured drivers can simply check to see if someone else has already claimed that vector. Other drivers should encourage users to reconfigure when interrupts appear to get lost.

Using configurable port addresses poses similar issues. Like an advisory locking scheme, two drivers should usually be able to mitigate the port addresses and interrupt vectors between them, but a poorly written driver can cause problems for the whole system, sometimes making it look like some other driver is at fault.

8.10.4 Preparing a Driver for Installation Using custom

The best thing you can do for the end user is to supply a driver installation shell script for use with **custom**(C). With such a script, a user has only to type **custom** and select options from the menus.

The **custom** utility extracts the contents of your driver installation floppy, using them to control the custom installation procedure. **custom** requires the presence of the following:

- On each floppy volume, a magic product identification file whose name is derived from the driver package name, the volume, and a machine identification string
- The object module containing your device driver
- A *permlist*, or a file containing the file permissions for the other files and what volumes and packages they belong to.
- The driver installation shell script that forms the table entries binding driver and kernel.

All files on the driver installation floppy should be given by relative pathname, starting at the root. For example, if */bin/ls* were on the floppy, its name on the floppy should be *./bin/ls* .

The magic file has a name of the following form:

./tmp/_lbl/prd=sidd/typ=286AT/rel=1.0.0/vol=01

where *sidd* is the driver's prefix (in this case, it stands for Sample Installable Device Driver), and 286AT is a machine-type specifier. To find the type specifier for your machine check the file */etc/perms/inst* on your system. If you are developing for a different system, check the */etc/perms/inst* file on that system for the type identifier for that machine.

In the above example, 1.0.0 is the software release number of the driver, and 01 is the volume number of the floppy containing the driver. Note that there is no volume 0: volume numbers must start at 01 and be consecutive.

This file must exist on each volume of your driver installation set (incrementing the volume number). It can be an empty file; its contents are ignored.

The *permlist* is a file containing a list of the files on the floppy, their permissions, and their packages. It will be used by **custom** both as an argument to *fixperm*(C) and to determine which driver files belong to which package. This makes it easy for the user to install one driver in a driver suite containing many. The *permlist* must live in *./tmp/perms.* Below is a sample *permlist*:

```
    #
    #  Copyright (C) The Santa Cruz Operation, 1985.
    #  This Module contains Proprietary Information of
    #  The Santa Cruz Operation, Microsoft Corporation
    #  and AT&T, and should be treated as Confidential.
    #
    #prd=sidd
    #typ=286AT
    #rel=1.0.0
    #set="Sample Installable Device Driver"
    #
    # User id's:
    #
    uidroot      0
    #
    # Group id's:
    #
    gid root     0
    #
    #
    #!SIDD    11    Sample Installable Device Driver
    #
    # Fields are: package [d,f,x]mode, user/group, links, path, volume

  SIDD     F644     root/root     1     ./tmp/perms/sidd          01
  SIDD     F755     root/root     1     ./tmp/init.sidd           01
  SIDD     f644     root/root     1     ./usr/sys/conf/sidd.o     01
```

Some of the fields are self-explanatory and can be copied verbatim. The *prd, typ, rel,* and *set* fields are comments to **fixperm** but are meaningful to **custom**. They must agree with the *prd, typ,* and *rel* entries in the magic filename, above. The set field is used by custom when it prompts for the users choice of packages to install.

Fields starting with '#!' are package specifiers. At least one must be present so that **custom** has something to prompt for. The '11' in the #!SIDD field above is the size, in 512 byte blocks, (as reported by **du**(C)) of all the files in the package. The comment following the size is also used in driver prompting.

The final section contains the package specifier, file type and permission, ownership, link count, file name and volume for each file on the distribution. The file type is d for directory, x for executable file, and f for normal file. If the file type is capitalized, the file is optional, and custom will not complain if it is missing. The files section is explained in more detail in **fixperm(C)**.

The driver installation shell script has the following duties:

- Check to see if the link kit is present, and install it if it isn't.
- Add the new driver entry points to the kernel using **configure**
- Edit the name of the new driver into the shell script *link_xenix*
- Run *link_xenix* to link the kernel
- Make the device nodes in */dev*.
- Run the shell script *hdinstall*, which backs up the old *xenix*, and puts your new *xenix* in its place.

Here is a sample installation shell script for the aforementioned Sample Installable Device Driver. It must be extracted into */tmp*, and have a name that starts with "init."

```
:
#
#  Copyright (C) The Santa Cruz Operation, 1985, 1986.
#  This Module contains Proprietary Information of
#  The Santa Cruz Operation, Microsoft Corporation
#
#  Driver initialization script
#
PATH=/bin:/usr/bin:/etc

cd /

# Get the permlist for the set containing the link kit package.
# Link Kit Release 2.0 is found in the "base" set; link kit release
# 2.1 and 2.2 is found in the "inst" set.

if [ -f /etc/base.perms ]; then
    PERM=/etc/base.perms
elif [ -f /etc/inst.perms ]; then
    PERM=/etc/inst.perms
else
    echo "Cannot locate /etc/base.perms or /etc/inst.perms" >&2
    exit 1
fi

# test to see if link kit is installed
until   fixperm -i -d LINK $PERM
do case $? in
    4) echo "The Link Kit is not installed." >&2 ;;
    5) echo "The Link Kit is only partially installed." >&2;;
    *) echo "Error testing for Link Kit. Exiting."; exit 1;;
    esac
    # Not fully installed. Do so here
    while  echo "Do you wish to install it now? (y/n) \c"
    do read ANSWER
        case $ANSWER in
        Y|y)  custom -o -i LINK
            break
            ;;
        N|n)  echo "Drivers cannot be installed without the Link Kit."
            exit 1
            ;;
        *)  echo "Please answer 'y' or 'n'. \c"
            ;;
        esac
    done
done

echo "adding device entry points"

cd /usr/sys/conf

# if the 'sidd' driver is present in the "master" file, "configure -j sidd"
# prints its major device number and returns 0. if the driver
# is not present in "master", "configure -j sidd" prints an error and
# returns 1
#
# configure -j NEXTMAJOR returns the smallest available major
# device number.
#
# configure -m $major ...  adds the given device entry points to
# xenix.
#
if major=`configure -j sidd`
then
    echo "Device entry points already configured"
else
    major=`configure -j NEXTMAJOR`
    configure -m $major -b -a siddopen siddclose siddstrategy siddtab || {
        echo "Cannot add device entry points to XENIX"
        exit 1
    }
fi

echo "adding sample driver to link line"

grep -s sidd.o link_xenix >/dev/null || {
    # add sidd.o to link line
    cp link_xenix link_xenix.00 || {
        echo "Cannot copy link_xenix" >&2
        exit 1
    }
```

```
        trap "mv link_xenix.00 link_xenix; exit 1" 1 2 3 15
        sed "s!c.o!& sidd.o!" link_xenix.00 > link_xenix || {
            echo "Cannot edit link_xenix" >&2
            mv link_xenix.00 link_xenix
            exit 1
        }
        trap 1 2 3 15
        chmod 700 link_xenix
    }

    echo "\nRe-linking the kernel ... \c"
    if link_xenix; then
        hdinstall
        echo "\nInstallable Device Driver installation complete.\n"
    else
        echo "\nLink failed, you will have to re-link the kernel\n"
    fi

    # make sample device nodes
    /etc/mknod /dev/sidd00 b $major 0
    /etc/mknod /dev/sidd01 b $major 1

    exit 0
```

To summarize, the custom-installable driver installation floppy must contain a *permlist*, a magic product identification file, the object module (extract into /usr/sys/conf), and the driver initialization script. Like all custom-installable floppies, a driver installation floppy is otherwise a normal tar volume.

```
-rw-r--r-- 1  root     0 Dec  9 08:46 ./tmp/_lbl/prd=sidd/typ=286AT/rel=1.0.0/vol=01
-rw-r--r-- 1  root   669 Dec 10 20:15 ./tmp/perms/sidd
-rw-r--r-- 1  root  6157 Dec 10 18:59 ./usr/sys/confsidd.o
-rwxrwxr-x 1  root  2097 Dec 10 20:14 ./tmp/init.sidd
```

8.11 Warnings

The following warnings can help you avoid problems when writing a device driver:

- Do not defer interrupts with *spl5*() or other *spl* calls any longer than necessary.
- Do not change the per process data in the *u* structure at interrupt time.
- Do not call *seterror*( ) or *sleep*( ) at interrupt time.
- Do not set your priority level at interrupt time to a lower priority than the one at which your interrupt routine was called.
- Make interrupt time processing as short as possible.
- Protect buffer and *clist* processing with the appropriate *spl* calls.
- Avoid "busy waiting" whenever possible.
- Never use floating point arithmetic operations in device driver code.
- If any assembly language device driver sets the direction flag (using **std**), it must clear it (using **cld**) before returning.
- Keep the local (stack) data requirements for your driver very small.

Chapter 9

Sample Device Drivers

9.1 Introduction

This chapter provides sample device driver code for a lineprinter, terminal, hard disk drive, and a memory-mapped video driver. Each segment of code (usually about 50 lines) is followed by some general comments, which describe the routines used and explain key lines in the program. These key lines are identified by line numbers.

9.2 Sample Device Driver for Line Printer

```
/*
** lp- prototype line printer driver
*/
#include "../h/types.h"
#include "../h/param.h"
#include "../h/sysmacros.h"
#include "../h/file.h"
#include "../h/tty.h"
#include "../h/conf.h"

#define LPPRI   PZERO+5
#define LOWAT   50
#define HIWAT   150

/* register definitions */

#define RBASE    0x00           /*base address of registers */
#define RDATA    (RBASE + 0)   /*place character here */
#define RSTATUS  (RBASE +1)   /*nonzero means busy*/
#define RCNTRL   (RBASE + 2)   /* write control here */

/* control definitions */
#define CINIT    0x01          /*initialize the interface */
#define CIENABL  0x02          /* Interrupt enable */
#define CRESET   0x04          /*interface reset */

/* flags definitions */
#define FIRST    0x01
#define ASLEEP   0x02
#define ACTIVE   0x04

struct clist lp_queue;
int lp_flags = 0;

int lpopen(), lpclose( ), lpwrite( ), lpintr( );

lpopen(dev)
int dev;
{
    if ((lp_flags & FIRST) == 0) {
        lp_flags |= FIRST;
        outb(RCNTRL, CRESET);
    }
        outb(RCNTRL, CIENABL);
}

lpclose(dev)
```

```
48 int dev;
49 {
50 }
```

Description of Device Driver for Line Printer

The device driver presented here is for a single parallel interface to a printer. It transfers characters one at a time, buffering the output from the user process through the use of character blocks (*cblocks*).

12:	LPPRI is the priority at which a process sleeps when it needs to stop. Since the priority is greater than PZERO, a signal sent to the suspended process will awaken it.
13:	LOWAT is the minimum number of characters in the buffer. If there are fewer than LOWAT characters in the buffer, a process that was suspended (because the buffer was full) can be restarted.
14:	HIWAT is the maximum number of characters in the queue. If a process fills the buffer up to this point, it will be suspended via *sleep*() until the buffer has drained below LOWAT.
18-21:	The device registers in this interface occupy a contiguous block of address, starting at RBASE, and running through RBASE+2. The data to be printed is placed in RDATA, one character at a time. Printer status can be read from RSTATUS, and the interface can be configured by writing into RCNTRL.
29-31:	The flags defined in these lines are kept in the *lp_flags* variable. FIRST is set if the interface has been initialized. ASLEEP is set if a process is asleep waiting for the buffer to drain below LOWAT. ACTIVE is set if the printer is active.
33:	*lp_queue* is the head of the linked list of *cblocks* that forms the output buffer.
34:	*lp_flags* is the variable in which the aforementioned flags are kept.

lpopen() - lines 38 to 47

The *lpopen*() routine is called when some process makes an **open(S)** system call on the special file that represents this driver. Its single argument, *dev*, represents the minor number of the device. Since this driver supports only one device, the minor number is ignored.

41-43:	If this is the first time (since XENIX was booted) that the device has been touched, the interface is initialized by setting the CRESET bit in the control register.
45:	Interrupts from this device are enabled by setting the CIENABL bit in the control register.

lpclose() - lines 48 to 51

The *lpclose*( ) routine is called on the last close of the device; that is, when the current close(S) system call results in zero processes referencing the device. No action is taken.

```
52 lpwrite(dev)
53 int dev;
54 {
55     register int c;
56     int x;
57
58     while((c=cpass())>=0){
59        x = spl5();
60        while(lp_queue.c_cc > HIWAT){
61           lpstart();
62           lp_flags |= ASLEEP;
63           sleep(&lp_queue, LPPRI);
64        }
65        splx(x);
66        putc(c, &lp_queue);
67     }
68     x=spl5();
69     lpstart();
70     splx(x);
71 }
72
73 lpstart()
74 {
75     if(lp_flags&ACTIVE)
76        return; /* interrupt chain is keeping printer going */
77     lp_flags |= ACTIVE;
78     lpintr(0);
79 }
80
81
82 lpintr(vec)
83 int vec;
84 {
```

```
85  int tmp;
86
87  if((lp_flags & ACTIVE) == 0)
88      return;   /* ignore spurious interrupt */
89
90  /* pass chars until busy */
91  while (inb(RSTATUS) == 0 && (tmp = getc(&lp_queue)) >= 0)
92      outb(RDATA, tmp);
93
94  /* wakeup the writer if necessary */
95  if(lp_queue.c_cc < LOWAT && lp_flags & ASLEEP) {
96      lp_flags &= ~ASLEEP;
97      wakeup(&lp_queue);
98  }
99
100  /* wakeup writer if waiting for drain */
101  if(lp_queue.c_cc <= 0)
102      lp_flags &= ~ACTIVE;
103 }
104
```

lpwrite() - lines 52 to 72

The *lpwrite*( ) routine is called to move the data from the user process to the output buffer. Code is defined as follows:

58:	While there are still characters to be transferred, do what follows.
59-66:	Raise the processor priority so the interrupt routine can't change the buffer. If the buffer is full, make sure the printer is running, note that the process is waiting, and put it to sleep. When the process wakes up, check to make sure the buffer has enough space, then go back to the old priority and put the character in the buffer.
68-70:	Make sure the printer is running, by locking out interrupts and calling *lpstart*( ).

lpstart() - lines 73 to 81

The *lpstart*() routine ensures that the printer is running. It is called twice from *lpwrite*(), and serves simply to avoid duplicate code. Code is defined as follows:

78-79:	If the printer is running, just return; otherwise, mark it ACTIVE, and call *lpintr*( ) to start the transfer of characters.

lpintr() - lines 82 to 104

The *lpintr*( ) routine is called from two places: *lpstart*( ), and from the kernel interrupt handling sequence when a device interrupt occurs. Code is defined as follows:

87-88:	If *lpintr*( ) is called unexpectedly, or the driver doesn't have anything to do, it just returns.
91-92:	While the printer indicates it can take more characters and the driver has characters to give it, the characters come from the buffer through *getc*( ), and pass to the interface by writing to the data register.
95-97:	If the buffer has fewer than LOWAT characters in it, and some process is asleep waiting for room, wake it up.
101-102:	If the queue is empty, turn off the ACTIVE flag. Note that the interrupt that completes the transfer and empties the buffer is in some sense "spurious," since it will occur with the ACTIVE flag reset.

9.3 Sample Device Driver for Terminal

```
 1 /*
 2 ** td - terminal device driver
 3 */
 4 #include "../h/types.h"
 5 #include "../h/param.h"
 6 #include "../h/sysmacros.h"
 7 #include "../h/dir.h"
 8 #include "../h/user.h"
 9 #include "../h/file.h"
10 #include "../h/tty.h"
11 #include "../h/conf.h"
12
13 /* registers */
14 #define RRDATA    0x01  /* received data */
15 #define RTDATA    0x02  /* transmitted data */
16 #define RSTATUS   0x03  /* status */
17 #define RCNTRL    0x04  /* control */
18 #define RIENABL   0x05  /* interrupt enable */
19 #define RSPEED    0x06  /* data rate */
20 #define RIIR      0x07      /* interrupt identification */
```

```
#define RENABL   0x08  /* interrupt control */
#define RLSR     0x09  /* line status register */
#define RTHR     0x0a  /* transmit holding register */

/* status register bits */
#define SRRDY    0x01  /* received data ready */
#define STRDY    0x02  /* transmitter ready */
#define SOERR    0x04  /* received data overrun */
#define SPERR    0x08  /* received data parity error */
#define SFERR    0x10  /* received data framing error */
#define SDSR     0x20  /* status of dsr (cd) */
#define SCTS     0x40  /* status of clear to send */

/* control register */
#define CBITS5   0x00  /* five bit chars */
#define CBITS6   0x01  /* six bit chars */
#define CBITS7   0x02  /* seven bit chars */
#define CBITS8   0x03  /* eight bit chars */
#define CDTR     0x04  /* data terminal ready */
#define CRTS     0x08  /* request to send */
#define CSTOP2   0x10  /* two stop bits */
#define CPARITY  0x20  /* parity on */
#define CEVEN    0x40  /* even parity otherwise odd */
#define CBREAK   0x80  /* set xmitter to space */

/* interrupt enable */
#define EXMIT    0x01  /* transmitter ready */
#define ERECV    0x02  /* receiver ready */
#define EMS      0x04  /* modem status change */

/* interrupt identification */
#define IRECV    0x01
#define IXMIT    0x02
#define IMS      0x04

#define NTDEVS   2
#define VECT0    34
#define VECT1    35

#define TURNOFF  0
#define TURNON   1

```

Description of Device Driver for Terminal

This driver supports one serial terminal on a hypothetical UART type interface.

12-18: The interface for each line consists of ten registers. The values that would be defined here represent offsets from the base address, which is defined in line 84. The base address differs for each line. The data to be transmitted is placed one character at a time into the RTDATA register. Likewise, the received data is read one character at a time from the RRDATA register.

You can determine the status of the UART by examining the contents of the RSTATUS register. Then you can adjust the UART configuration by changing the contents of the RCNTRL register. Interrupts are enabled or disabled by setting the bits in the RIENABL register. The data rate is set by changing the contents of the RSPEED register. Interrupts are identified by setting the bits in the RIIR register.

33-42: The two low order bits of the "control register" determine the length of the character sent. The next two bits control the data-terminal-ready and request-to-send lines of the interface. The next bit controls the number of stop bits, the next controls whether parity is generated, and the next controls whether generated parity is even or odd. Finally, the most significant bit, if it is set, forces the transmitter to continuous spacing.

45-47: The three low order bits of the "interrupt enable" register control whether the device generates interrupts under certain conditions. If bit 0 is set, an interrupt is generated every time the transmitter becomes ready for another character. If bit 1 is set, an interrupt is generated every time a character is received. If bit 2 is set, an interrupt is generated every time the data-set-ready line changes state.

50-52: After an interrupt, the value in the interrupt identification register will contain one of three values, indicating the reason for the interrupt.

```
63 /* datarates*/
64 int td_speeds[] = {
65     /*B0   */  0,
66     /* B50  */  2304,
67     /* B75  */  1536,
68     /* B110 */  1047,
69     /*B134 */  857,
70     /*B150 */  768,
71     /*B200 */  0,
72     /* B300 */  384,
73     /* B600 */    192,
74     /*B1200*/   96,
75     /*B1800*/   64,
76     /* B2400*/   48,
77     /*B4800*/   24,
78     /*B9600*/   12,
79     /* EXTA */    6,  /*19.2kbps*/
80     /*EXTB */   58  /*2000 bps*/
81 };
82
83 struct tty td_tty[NTDEVS];
84 int td_addr[NTDEVS] = { 0x00, 0x10 };
85
86
87 tdopen(dev,flag)
88 int dev, flag;
89 {
90     register struct tty *tp;
91     int addr;
92     int tdproc();
93     int x;
94
95     if (UNMODEM(dev) >=NTDEVS) {
96        seterror(ENXIO);
97        return;
98     }
99     tp = &td_tty[UNMODEM(dev)];
100     addr=td_addr[UNMODEM(dev)];
101     if( (tp->t_lflag & XCLUDE) && !suser()) {
102        seterror(EBUSY);
103        return;
104     }
105     if((tp->t_state&(ISOPEN|WOPEN)) ==0){
106        ttinit(tp);
107        tp->t_proc=tdproc;
108        tp->t_oflag=OPOST|ONLCR;
109        tp->t_iflag=ISTRIP|IXON;
110        tp->t_lflag=ECHO|ICANON|ISIG|ECHOE|ECHOK;
111        tdparam(dev);
112     }
113     x = spl5();
114     if (ISMODEM(dev) ||tp->t_cflag&CLOCAL || tdmodem(dev, TURNON))
115        tp->t_state |=CARR_ON;
116     else
117        tp->t_state&= ~CARR_ON;
```

64-80:	These lines define the values to be loaded into the RSPEED register in order to get various data rates.

83:	Each line must have a *tty* structure allocated for it.
84:	Here, the base addresses of the registers are defined for each line.

tdopen() - lines 87 to 126

The *tdopen*() routine is called whenever a process makes an **open**(S) system call on the special file corresponding to this driver. Code is defined as follows:

95-98:	If the minor number indicates a device that doesn't exist, indicate the error, and return.
101-104:	If the line is already open for exclusive use, and the current user is not the super-user, indicate the error and return.
105-112:	If the line is not already open, initialize the *tty* structure via a call to *ttinit*( ), set the value of the *proc* field in the *tty* structure, initialize the input and output mode flags, and configure the line by calling *tdparam*(). Note that the flags are initialized so that the terminal will behave in a reasonable manner if used as the console in single-user mode.
113:	Defer interrupts so the interrupt routines cannot change the state while it is being examined.
114-117:	If the line is not using modem control, or if it is not turning on the data-terminal-ready and request-to-send signals (which results in carrier-detect being asserted by the remote device), indicate that the carrier signal is present on this line. Otherwise, indicate that there is no carrier signal.

```
118  if(!(flag&FNDELAY))
119     while((tp->t_state&CARR_ON)==0) {
120        tp->t_state |= WOPEN;
121        sleep((caddr_t)&tp->t_canq, TTIPRI);
122     }
123  (*linesw[tp->t_line].l_open)(tp);
124  splx(x);
125 }
126
127 tdclose(dev)
128 {
129     register struct tty *tp;
130
131     tp= &td_tty[UNMODEM(dev)];
132     (*linesw[tp->t_line].l_close)(tp);
133     if(tp->t_cflag & HUPCL)
```

```
134         tdmodem(dev, TURNOFF);
135     tp->t_lflag &= ~XCLUDE; /* turn off exclusive use bit */
136     /* turn off interrupts */
137     outb(td_addr[UNMODEM(dev)] + RIENABL, 0);
138 }
139
140 tdread(dev)
141 {
142     register struct tty *tp;
143
144     tp = &td_tty[UNMODEM(dev)];
145     (*linesw[tp->t_line].l_read)(tp);
146 }
147
148 tdwrite(dev)
149 {
150     register struct tty *tp;
151
152     tp = &td_tty[UNMODEM(dev)];
153     (*linesw[tp->t_line].l_write)(tp);
154 }
155
156 tdparam(dev)
157 {
158     register int cflag;
159     register int addr;
160     register int temp, speed, x;
161
162     addr = td_addr[UNMODEM(dev)];
163     cflag = td_tty[UNMODEM(dev)].t_cflag;
164
165     /* if speed is B0, turn line off */
166     if( (cflag & CBAUD) == B0){
167         outb(addr + RCNTRL, inb(addr+RCNTRL) &~CDTR & ~CRTS);
168         return;
169     }
170
```

118-122:	If **open(S)** is supposed to wait for the carrier, wait until the carrier is present.
123:	Call the *l_open*() routine indirectly through the *linesw* table. This completes the work required for the current line discipline to open a line.
124:	Allow further interrupts.

tdclose() - lines 127 to 138

The *tdclose*( ) routine is called on the last close on a line.

132:	Call the close(S) routine through the *linesw* table to do the work required by the current line discipline.
133-134:	If the "hang up on last close" bit is set, drop the data-terminal-ready and request-to-send signals.
135:	Reset the exclusive use bit.
137:	To prevent spurious interrupts, disable all interrupts for this line.

tdread() and tdwrite() - lines 140 to 155

Both of these routines simply call the relevant routine via the *linesw* table; the called routine performs the appropriate action for the current line discipline.

tdparam() - lines 156 to 170

The *tdparam*() routine configures the line to the mode specified in the appropriate *tty* structure.

162-163:	Get the base address and flags for the referenced line.
166-168:	The speed B0 means "hangup the line."

```
171   /*setupspeed */
172   outb(addr + RSPEED, td_speeds[cflag&CBAUD]);
173
174   /* setupline control */
175   temp = (cflag &CSIZE)>>4;/* length */
176   if ( cflag & CSTOPB)
177       temp |=CSTOP2;
178   if ( cflag& PARENB ) {
179       temp |=CPARITY;
180       if ((cflag&PARODD) ==0)
181           temp |=CEVEN;
182   }
183   temp |=CDTR |CRTS;
184   out( addr+ RCNTRL, temp);
185
186   /* setup interrupts*/
187   temp = EXMIT;
188   if ( cflag & CREAD )
189       temp |= ERECV;
```

```
190    outb(addr+RENABL, inb(RENABL) | temp);
191 }
192
193 tdmodem(dev, cmd)
194 int dev, cmd;
195 {
196    register int addr;
197
198    addr=td_addr[UNMODEM(dev)];
199    switch(cmd) {
200       case TURNON: /* enable modem interrupts, set DTR & RTS true */
201          outb(addr+RENABL, inb(RENABL) |EMS);
202          outb(addr+RCNTRL, inb(RENABL) |CDTR |CRTS );
203          break;
204       case TURNOFF: /* disable modem interrupts, reset DTR, RTS */
205          outb(addr+RENABL, inb(RENABL) & ~EMS);
206          outb(addr+RCNTRL, inb(RENABL) & ~(CDTR |CRTS) );
207          break;
208    }
209    return (inb(addr+RSTATUS) &SDSR);
210 }
211
212 tdintr(vec)
213 int vec;
214 {
215    register int iir, dev, inter;
216
217    switch( vec ) {
218       case VECT0:
219          dev=0;
220          break;
221       case VECT1:
222          dev=1;
223          break;
224       default:
225          printf("tdint: wrong channel interrupt (%x)\n",vec);
226          return;
227    }
```

173: The remainder of the *tdparam*( ) routine simply loads the device registers with the correct values.

tdmodem( ) - lines 1913 to 211

The *tdmodem*( ) routine controls the data-terminal-ready and request-to-send line signals. Its return value indicates whether data-set-ready signal (carrier detect) is present for the line.

200-203:	If *cmd* was TURNON, turn on modem interrupts, and assert data-terminal-ready and request-to-send.
204-207:	If *cmd* was TURNOFF, disable modem interrupts, and drop data-terminal-ready andrequest-to-send.
209:	Return a zero value if there is no data-set-ready on this line, otherwise return a non-zero value.

tdintr() - lines 212 to 236

The *tdintr*( ) routine determines which line caused the interrupt and the reason for the interrupt, and calls the appropriate routine to handle the interrupt.

217-227:	Different lines will result in different interrupt vectors being passed as the *tdintr*() routine's argument. Here, the minor number is determined from the interrupt vector that was passed to *tdintr*( ).

```
228     while( (iir= inb(td_addr[dev]+RIIR)) != 0) {
229        if((iir & IXMIT) != 0)
230            tdxint(dev);
231        if( (iir & IRECV) != 0 )
232            tdrint(dev);
233        if( (iir & IMS) != 0 )
234            tdmint(dev);
235     }
236 }
237
238 tdxint(dev)
239 {
240     register struct tty *tp;
241     register int addr;
242
243     tp = &td_tty[UNMODEM(dev)];
244     addr = td_addr[UNMODEM(dev)];
245     if( inb(addr +RSTATUS) & STRDY)
246     {
247        tp->t_state &= ~BUSY;
248        if (tp->t_state & TTXON) {
249              outb(addr+ RTDATA, CSTART);
250              tp->t_state &= ~TTXON;
251        } else if(tp->t_state & TTXOFF) {
252              outb(addr+ RTDATA,CSTOP);
253              tp->t_state &= ~TTXOFF;
254        } else
255              tdproc(tp, T_OUTPUT);
256     }
```

```
257 }
258
259 tdrint(dev)
260 {
261     register int c, status;
262     register int addr;
263     register struct tty *tp;
264     int flg;
265
266     tp=&td_tty[UNMODEM(dev)];
267     addr=td_addr[UNMODEM(dev)];
268
269     /*get char and status */
270     c=inb(addr+RRDATA);
271     status=inb(addr+RLSR);
272
```

228–236:	While the interrupt identification register indicates that there are more interrupts, call the appropriate routine. When the condition that caused the interrupt is resolved, the UART will reset the bit in the register by itself.

tdxint() - lines 238 to 258

The *tdxint*( ) routine is called when a transmitter ready interrupt is received. It may issue a CSTOP character to indicate that the device on the other end must stop sending characters; it may issue a CSTART character to indicate that the device on the other end may resume sending characters, or it may call *tdproc*( ) to send the next character in the queue.

245–247:	If the transmitter is ready, reset the busy indicator.
248–250:	If the line is to be restarted, send a CSTART, and reset the indicator.
251–253:	If the line is to be stopped, send a CSTOP, and reset the character.
254–255:	Otherwise, call *tdproc*( ) and ask it to send the next character in the queue.

tdrint() - lines 259 to 327

The *tdrint*( ) routine is called when a receiver interrupt is received. All it has to do is pass the character, along with any errors, to the appropriate routine via the *linesw* table.

270-271: Get the character and status.

```
273    /*
274     * Were there any errors on input?
275     */
276    if( status & SOERR)    /* overrun error*/
277       c |= OVERRUN;
278    if( status & SPERR)    /*parityerror*/
279       c |= PERROR;
280    if(status & SFERR)    /*framing error */
281       c |= FRERROR;
282
283    if(tp->t_rbuf.c_ptr == NULL)
284       return;
285    flg=tp->t_iflag;
286    if (flg&IXON){
287       register int ctmp;
288       ctmp = c & 0177;
289       if(tp->t_state &TTSTOP){
290          if(ctmp== CSTART || flg&IXANY)
291          (*tp->t_proc)(tp, T_RESUME);
292       }else{
293          if(ctmp == CSTOP)
294             (*tp->t_proc)(tp, T_SUSPEND);
295       }
296       if (ctmp== CSTART || ctmp == CSTOP)
297       return;
298    }
299    if(c&PERROR && !(flg&INPCK))
300       c &= ~PERROR;
301    if(c&(FRERROR|PERROR|OVERRUN)){
302       if((c&0377) ==0){
303          if (flg&IGNBRK)
304          return;
305          if (flg&BRKINT) {
306                (*linesw[tp->t_line].l_input)
307                (tp,L_BREAK);
308                return;
309          }
310       } else {
311          if (flg&IGNPAR)
312          return;
313       }
314    } else{
315       if (flg&ISTRIP)
316          c &=0177;
317       else{
318          c&=0377;
319       }
320    }
```

```
321     *tp->t_rbuf.c_ptr= c;
322     tp->t_rbuf.c_count--;
323     (*linesw[tp->t_line].l_input)(tp, L_BUF);
324 }
325
326 tdmint(dev)
327 {
328     register struct tty *tp;
329     register int addr, c;
330
331     tp = &td_tty[UNMODEM(dev)];
332     if(tp->t_cflag &CLOCAL) {
333         return;
334     }
335     addr= td_addr[UNMODEM(dev)];
336
337     if(inb(addr +RSTATUS) & SDSR) {
338         if((tp->t_state & CARR_ON)==0) {
339             tp->t_state|=CARR_ON;
340             wakeup(&tp->t_canq);
341         }
342     } else{
343         if(tp->t_state& CARR_ON) {
344             if(tp->t_state & ISOPEN) {
345                 signal(tp->t_pgrp, SIGHUP);
346                 tdmodem(dev, TURNOFF);
347                 ttyflush(tp, (FREAD|FWRITE));
348             }
349             tp->t_state&=~CARR_ON;
350         }
351     }
352 }
353
354 tdioctl(dev, cmd, arg, mode)
355 int dev;
356 int cmd;
357 faddr_t arg;
358 int mode;
359 {
360   if(ttiocom(&td_tty[UNMODEM(dev)], cmd, arg, mode))
361     tdparam(dev);
362 }
363
```

277-282:	If any errors were detected, set the appropriate bit in *c*.
286-322:	This code determines whether the character is XON and if output is stopped, it restarts it. If the character is XOFF, output is suspended.

	Further error checking is then carried out and characters in error are discarded. The character is then placed in the queue.
323:	And finally, pass the character and errors to the *l_input*( ) routine for the current line discipline.

tdmint() - lines 326-353

The *tdmint*( ) routine is called whenever a modem interrupt is caught.

332-333:	If there is no modem support for this line, just return.
337-347:	If a data-set-ready is present for this line, and it wasn't before, mark the line as having carrier, and wake up any processes that are waiting for the carrier before their *tdopen*( ) call can be completed.
342-349:	If no data-set-ready is present for this line, and one existed before, send a hangup signal to all of the processes associated with this line, call *tdmodem*() to hang up the line, flush the output queue for this line by calling *ttyflush*( ), and mark the line as having no carrier.

tdioctl( ) - lines 354-363

The *tdioctl*( ) routine is called when some process makes an **ioctl(S)** system call on a device associated with the driver. It just calls *ttiocom*( ) which returns a non-zero value if the hardware must be reconfigured.

```
364 tdproc(tp, cmd)
365 register struct tty *tp;
366 {
367   register int c;
368   register int addr;
369
370   extern ttrstrt();
371
372   addr = td_addr[tp - td_tty];
373   switch (cmd) {
374
375   case T_TIME:
376     tp->t_state &= ~TIMEOUT;
377     outb(addr+RCNTRL, inb(addr+RCNTRL) & ~CBREAK);
378     goto start;
379
380   case T_WFLUSH:
381     tp->t_tbuf.c_size -= tp->t_tbuf.c_count;
```

```
382   tp->t_tbuf.c_count=0;
383  case T_RESUME:
384   tp->t_state &= ~TTSTOP;
385   goto start;
386
387  case T_OUTPUT:
388  start:
389   if(tp->t_state&(TIMEOUT|TTSTOP|BUSY))
390     break;
391   {
392     register struct ccblock *tbuf;
393
394     tbuf = &tp->t_tbuf;
395     if( tbuf->c_ptr==NULL ||
396       tbuf->c_count == 0) {
397       if( tbuf->c_ptr)
398         tbuf->c_ptr -= tbuf->c_size
399             - tbuf->c_count;
400       if(! (CPRES &
401         (*linesw[tp->t_line].l_output)(tp)))
402           break;
403     }
404     tp->t_state |= BUSY;
405     outb(addr+ RTHR, *tbuf->c_ptr++);
406     tbuf->c_count--;
407   }
408   break;
409
```

tdproc() - lines 364 to 441

The *tdproc*( ) routine is called to effect some change on the output, such as emitting the next character in the queue, or halting or restarting the output.

373:	The *cmd* argument determines the action taken.
375-378:	The time delay for outputting a break has finished. Reset the flag that indicates there is a delay in progress, and stop sending a continuous space. Then restart output by jumping to **start**. A WFLUSH command resets the character buffer pointers and the count.
383-385:	Either a line on which output was stopped is restarting, or someone is waiting for the output queue to drain. Reset the flag indicating that output on this line is stopped, and start the output again by jumping to **start** (line 388).

391-407: Try to put out another character. If some delay is in progress (TIMEOUT) or the line output has stopped (TTSTOP) or a character is in the process of being output (BUSY), just return.

405-420: This code manipulates the character queue in order to output either a block of characters (by calling the *l_output*() routine) or perform a single character output operation (in this example via the *outb*( ) routine).

Note that if the device is capable of outputting more than one character in a single operation then this should be done, and the buffer pointer (*c_ptr*) and the count (*c_count*) should be adjusted appropriately.

```
410  case T_SUSPEND:
411    tp->t_state|=TTSTOP;
412    break;
413
414  case T_BLOCK:
415    tp->t_state&=~TTXON;
416    tp->t_state|=TBLOCK;
417    if (tp->t_state&BUSY)
418      tp->t_state|=TTXOFF;
419    else
420      outb(addr+RTDATA, CSTOP);
421    break;
422
423  case T_RFLUSH:
424    if(!(tp->t_state&TBLOCK))
425      break;
426  case T_UNBLOCK:
427    tp->t_state &=~(TTXOFF|TBLOCK);
428    if (tp->t_state&BUSY)
429      tp->t_state|=TTXON;
430    else
431      outb(addr+RTDATA,CSTART);
432    break;
433
434  case T_BREAK:
435    outb(addr+RCNTRL, inb( addr+RCNTRL) |CBREAK);
436    tp->t_state|=TIMEOUT;
437    timeout(ttrstrt, tp, HZ/4);
438    break;
439  }
440 }
441
```

410-412:	To stop the output on this line, since there is no way to stop the character we have already passed to the controller, just flag the line stopped, and drop through.
414-421:	To tell the device on the other end to stop sending characters, reset the flag asking to stop the line, and mark the line stopped. If the line is already busy, set the flag; otherwise, output a CSTOP character.
423-425:	A process is waiting to flush the input queue. If the device hasn't been blocked, just return. Otherwise, drop through and unblock the device.
426-432:	To tell the device on the other end to resume sending characters, adjust the flags. If the controller is sending a character, set the flag to send a CSTART later; otherwise, send the CSTART now.
434-438:	To send a break, set the transmitter to continuous space, mark the line as waiting for a delay, and schedule output to be restarted later.

9.4 Sample Device Driver for Disk Drive

```
 1 /*
 2 ** hd- prototype hard disk driver
 3 */
 4
 5 #include "../h/types.h"
 6 #include "../h/param.h"
 7 #include "../h/sysmacros.h"
 8 #include "../h/buf.h"
 9 #include "../h/iobuf.h"
10 #include "../h/dir.h"
11 #include "../h/conf.h"
12
13
14 /*disk parameters*/
15 #define NHD    4              /*number of drives*/
16 #define NPARTS  8               /*#partitions/disk*/
17 #define NCPD   600            /*#cylinders/disk*/
18 #define NTPC   4             /*#tracks/cylinder*/
19 #define NSPT   10             /*#sectors/track*/
20 #define NBPS   512             /*#bytes/sector*/
21 #define NSPC   (NSPT*NTPC)          /*sectors/cylinder*/
22 #define NSPB   (BSIZE/NBPS)         /*sectors/block*/
23 #define NBPC   (NTPC*NSPT/NSPB)     /*blocks/cylinder*/
24
```

```
25 /* addresses of controller registers */
26 #define RBASE   0x00          /* base of all registers */
27 #define RCMD    (RBASE+0)     /* command register */
28 #define RSTAT   (RBASE+2)     /* status - nonzero means error */
29 #define RCYL    (RBASE+4)     /* target cylinder */
30 #define RTRK    (RBASE+6)     /* target track */
31 #define RSEC    (RBASE+8)     /* target sector */
32 #define RADDRL  (RBASE+10)    /* target memory address lo 16 bits */
33 #define RADDRH  (RBASE+12)    /* target memory address hi 8 bits */
34 #define RCNT    (RBASE+14)    /* number of sectors to xfer */
35 #define RDRV    (RBASE+16)    /* drive select register */
36
37 /* bits in RCMD register */
38 #define CREAD   0x01     /* start a read */
39 #define CWRITE  0x02     /* start a write */
40 #define CRESET  0x03     /* reset the controller */
41
42 /*
43 ** minor number layout is 0000dppp in binary
44 ** where d is the drive number and ppp is the partition
45 */
46 #define drive(d)  (minor(d) >> 3)
47 #define part(d)   (minor(d) & 0x07)
48
49 /* partition table */
50 struct partab {
51   daddr_t len;   /* # of blocks in partition */
52   int  cyloff;  /* starting cylinder of partition */
53 };
54
```

Description of Device Driver for Disk Drive

The device driver presented here is for an intelligent controller that is attached to one or more disk drives. The controller can handle multiple sector transfers that cross track and cylinder boundaries.

15:	NHD defines the number of drives the controller can be attached to.
16:	NPARTS defines the number of partitions which can be configured on a single drive.
17-23:	Each disk drive attached to the controller has NCPD cylinders; each cylinder has NTPC tracks (or NSPC sectors), and each track has NSPT sectors. The sectors are NBPS bytes long and each cylinder has NBPC blocks. Each block has NSPB sectors.

26-35:	The controller registers occupy a region of contiguous address space starting at RBASE and running through RBASE+16.
38-40:	To make the controller perform some action, the registers that describe the transfer (RCYL, RTRK, RSEC, RADDRL, RADDRH, RCNT) are set to the appropriate values, and then the bit representing the desired action is written into the RCMD register.
46-47:	The *drive*( ) and *part*( ) macros split out the two parts of the minor number. Bits 0 through 2 represent the partition on the disk, and the remaining bits specify the drive number. Thus, the minor number for drive 1, partition 2 would be 10 decimal.
50-53:	Large disks are typically broken into several partitions of a more manageable size. The structure that specifies the size of the partitions specifies the length of the partition in blocks, and the starting cylinder of the partition.

```
53 int hdread(), hdwrite(), hdintr(), hdstrategy();
54
55 #define  MNT1SZ  300    /* size of /mnt1 file system, in cylinders */
56 #define  MNT2SZ   50    /* size of /mnt2 file system, in cylinders */
57 #define  TMP1SZ  100    /* size of /tmp file system, in cylinders */
58 #define  TMPOFS  (MNT1SZ+MNT2SZ)   /* offset of user file system */
59
60 struct partab hd_sizes[8] = {
61   NCPD*NBPC,     0,          /* whole disk */
62   MNT1SZ*NBPC,   0,          /* mnt1 area */
63   MNT2SZ*NBPC,   MNT1SZ,     /* mnt2 area */
64   TMP1SZ*NBPC,   TMPOFS,     /* tmp1 area */
65   0,        0,               /* spare */
66   0,        0,               /* spare */
67   0,        0,               /* spare */
68   0,        0,               /* spare */
69 };
70
71 #define ERRLIM  10    /* maximum retries */
72
73 struct iobuf  hdtab;    /* start of request queue */
74 struct buf   rhdbuf;    /* header for raw i/o */
75 /*
76 **  Strategy Routine:
77 **  Arguments:
78 **   Pointer to buffer structure
79 **  Function:
80 **   Check validity of request
81 **   Queue the request
82 **   Start up the device if idle
```

```
83 */
84 int hdstrategy(bp)
85 register struct buf *bp;
86 {
87   register int dr, pa;  /* drive and partition numbers */
88   daddr_t sz, bn;
89   int x;
90
91   dr = drive(bp->b_dev);
92   pa = part(bp->b_dev);
93   bn = bp->b_blkno * NSPB;
94   sz = (bp -> b_bcount + BMASK) >> BSHIFT;
95   if(dr<NHD && pa<NPARTS && bn>=0 && bn<hd_sizes[pa].len &&
96     ((bn+ sz < hd_sizes[pa].len) || (bp->b_flags & B_READ)))
97   {
98     if (bn + sz > hd_sizes[pa].len) {
99       sz = (hd_sizes[pa].len - bn) * NBPS;
100       bp->b_resid = bp->b_bcount - (unsigned) sz;
101       bp->b_bcount = (unsigned) sz;
102     }
103   } else {
104     bp->b_flags |= B_ERROR;
105     iodone(bp);
106     return;
107   }
108   bp->b_cylin = (bp->b_blkno / NBPC) + hd_sizes[pa].cyloff;
109   x = spl5();
110   disksort(&hdtab, bp);
111   if(hdtab.b_active == NULL)
112     hdstart();
113   splx(x);
114 }
115
```

60-69:	This driver splits a disk into up to eight pieces, but at present, only four are used. The first partition covers the whole disk. The remaining three split the disk three ways, one partition for each of the *mnt1*, *mnt2*, and *tmp* filesystems.
73:	The buffer headers representing requests for this driver are linked into a queue, with *hdtab* forming the head of the queue. In addition, information regarding the state of the driver is kept in *hdtab*.
74:	Each block driver that wants to allow raw I/O allocates one buffer header for this purpose.

hdstrategy() - lines 84 to 114

The *hdstrategy*() routine is called by the kernel to queue a request for I/O. The single argument is a pointer to the buffer header which contains all of the data relevant to the request. The strategy routine is responsible for validating the request, and linking it into the queue of outstanding requests.

91-94:	First, compute various useful numbers that will be used repeatedly during the validation process.
95-107:	If the request is for a non-existent drive or a non-existent partition, if it lies completely outside the specified partition, or is a write, and ends outside the partition, the B_ERROR bit in the *b_flags* field of the header is set to indicate that the request has failed. The request is then marked as complete by calling *iodone*() with the pointer to the header as an argument. If the request is a read, and ends outside the partition, it is truncated to lie completely within the partition.
108:	Compute the target cylinder of the request for the benefit of the *disksort*() routine.
109:	Block interrupts, to prevent the interrupt routine from changing the queue of outstanding requests.
110:	Sort the request into the queue by passing it and the head of the queue to *disksort*().
111-112:	If the controller is not already active, start it up.
113:	Re-enable interrupts and return to the user process.

```
116 /*
117 *   Startup Routine:
118 *   Arguments:
119 *    None
120 *   Function:
121 *    Compute device-dependent parameters
122 *    Startup device
123 *    Indicate request to I/O monitor routines
124 */
125 hdstart()
126 {
127   register struct buf *bp;   /* buffer pointer */
128   register unsigned sec;
129
130   if((bp = hdtab.b_actf) == NULL) {
131     hdtab.b_active = 0;
132     return;
133   }
134   hdtab.b_active = 1;
```

```
135
136  sec= ((unsigned)bp->b_blkno * NSPB);
137  out(RCYL, sec/NSPC);           /*cylinder */
138  sec %=NSPC;
139  out(RTRK, sec / NSPT);            /* track */
140  out(RSEC, sec % NSPT);            /*sector*/
141  out(RCNT, bp->b_bcount/NBPS);      /*count */
142  out(RDRV, drive(bp->b_dev));       /*drive*/
143  out(RADDRL, bp->b_paddr & 0xffff);   /* memory address lo */
144  out(RADDRH, bp->b_paddr >> 16);      /* memory address hi */
145  if(bp->b_flags & B_READ)
146    out(RCMD, CREAD);
147  else
148    out(RCMD, CWRITE);
149 }
150
151 /*
152  *   Interrupt routine:
153  *    Check completion status
154  *    Indicate completion to i/o monitor routines
155  *    Log errors
156  *    Restart (on error) or start next request
157  */
158 hdintr()
159 {
160  register struct buf *bp;
161
162  if(hdtab.b_active == 0)
163    return;
164
```

hdstart() - lines to 125 to 149

The *hdstart*() routine calculates the physical address on the disk, and starts the transfer.

130-133:	If there are no active requests, mark the state of the driver as idle, and return.
134:	Mark the state of the driver as active.
136-138:	Calculate the starting cylinder, track, and sector of the request, and load the controller registers with these values.
139-144:	Load the controller with the drive number, and the memory address of the data to be transferred.

145–149: If the request is a read request, issue a read command; otherwise, issue a write command.

hdintr() - lines 158 to 184

The *hdintr*( ) routine is called by the kernel through the *vecintsw* table whenever the controller issues an interrupt.

162–163: If an unexpected call occurs, just return.

```
165  bp=hdtab.b_actf;
166
167  if (in(RSTAT) != 0) {
168    out(RCMD, CRESET);
169    if (++hdtab.b_errcnt <= ERRLIM) {
170      hdstart( );
171      return;
172    }
173    bp->b_flags |=B_ERROR;
174    deverr(&hdtab,bp,in(RSTAT),0);
175  }
176  /*
177   *  Flag current request complete, start next one
178   */
179  hdtab.b_errcnt=0;
180  hdtab.b_actf=bp->av_forw;
181  bp->b_resid=0;
182  iodone(bp);
183  hdstart( );
184 }
185
186 /*
187  * raw read routine:
188  *  This routine calls "physio" which computes and validates
189  *  a physical address from the current logical address.
190  *
191  *   Arguments
192  *    Full device number
193  *   Functions:
194  *    Call physio which does the actual raw (physical) I/O
195  *    The arguments to physio are:
196  *     pointer to the strategy routine
197  *     buffer for raw I/O
198  *     device
199  *     read/write flag
200  */
201 hdread(dev)
202 {
203
```

```
204 physio(hdstrategy, &rhdbuf, dev, B_READ|B_NOCROSS);
205 }
206
207 /*
208 *   Rawwrite routine:
209 *   Arguments(to hdwrite):
210 *    Full device number
211 *   Functions:
212 *    Call physio which does actual raw (physical) I/O
213 */
```

165:	Get a pointer to the first buffer header in the chain; this is the request that is currently being serviced.
167-175:	If the controller indicates an error, and the operation hasn't been retried ERRLIM times, try it again. If it has been retried ERRLIM times, assume it is a hard error, mark the request as failed, and call *deverror*() to print a console message about the failure.
179-183:	Mark this request complete, take it out of the request queue, and call *hdstart*() to start on the next request.

hdread() - lines 201 to 213

The *hdread*() routine is called by the kernel when a process requests raw read on the device. All it has to do is call *physio*( ), passing the name of the strategy routine, a pointer to the raw buffer header, the device number, and a flag indicating a read request for a device that cannot directly address across 64k boundaries. The *physio*() routine does all the preliminary work, and queues the request by calling the device strategy routine.

Last Five Lines of Sample Driver

```
214  hdwrite(dev)
215  {
216
217    physio(hdstrategy, &rhdbuf, dev, B_WRITE|B_NOCROSS);
218  }
```

hdwrite() - lines 214 to 218

The *hdwrite*() routine is called by the kernel when a process requests a raw write on the device. Its responsibilities and actions are the same as *hdread*(), except that it passes a flag indicating a write request for a device that cannot directly address across 64k boundaries.

9.5 Memory-Mapped Video Driver

```
#include "../h/types.h"
#include "../h/param.h"
#include "../h/sysmacros.h"
#include "../h/dir.h"
#include "../h/signal.h"
#include "../h/user.h"
#include "../h/mmu.h"
#include "../h/file.h"
#include "../h/errno.h"
#define INDEX   0x3d4    /*index and data register location   */
#define DATA    0x3d5    /* for the CRT Controller 6845       */
#define MODE    0x3d8    /* address for mode register         */
#define COLOR   0x3d9    /* address for color register        */
#define MBASE   0xB8000L /* starting address even lines       */
#define SCN_GRF 0x0a     /* graphics mode to put in MODE reg  */
#define SCN_TXT 0x0d     /* text mode to put in MODE reg      */
#define SIZE    0x4000   /* size of the video RAM             */
#define HI_IN   0x10     /* hi-intensity                      */

#define ACOL    0x20     /* main color or alternate color     */

char grchar_mode[ ]= { 0x71, 0x50, 0x5a, 0x0a, 0x1f, 0x06, 0x19,
          0x1c, 0x02, 0x07, 0x06, 0x07, 0x00, 0x00 };

char grgraf_mode[ ]= { 0x38, 0x28, 0x2d, 0x0a, 0x7f, 0x06, 0x64,
          0x70, 0x02, 0x01, 0x06, 0x07, 0x00, 0x00 };

struct gr_mode{
    char   scr_mode;
    char   scr_color;
    int    scr_off;
}gr_mode;

faddr_t gr_basaddr;

grinit()
{
    int sel;

    sel = dscralloc();
    mmudescr(sel, MBASE, SIZE-1, DSA_DATA);
    gr_basaddr = sotofar(sel, 0);
}

gropen()
```

```
46 {
47     gr_mode.scr_color = 0x00;
48     gr_mode.scr_mode = 0x00;
49     gr_mode.scr_off  = 0x00;
50 }
51
52 grclose()
53 {
54     gr_mode.scr_mode = 0;
55     gr_mode.scr_color = 0x00;
56     gr_setcolor();
57     gr_setscrn(0);
58 }
59
```

This device driver is for a 286 processor running in protected mode. It allocates a global descriptor for the video RAM, then uses *mmudescr()* to map that RAM into the kernel's address space. Once mapped, far pointers are used to address the video memory.

A driver for an unmapped machine would be similar, except that no selector mapping would be performed.

8-11:	These are the registers used to manipulate the 6845 CRT controller. INDEX and DATA are used for mode initialization, MODE to determine the type of graphics mode to be used, and COLOR to set the background color.
12:	MBASE is the starting address of the video RAM.
14-15:	SCN_GRF and SCN_TXT are the codes written into the MODE register to determine whether text or graphics mode should be used.
16:	SIZE is the size of the video RAM.
18-20:	HI_IN is or'ed into the color register to produce high-intensity background colors. GRAF is a binary flag that contains the current state of the screen, either text or graphics. ACOL contains the code for the currently selected color palette.
22-26:	*grchar_mode* and *grgraf_mode* are arrays containing the initialization sequences for the respective modes.
28-32:	*gr_mode* is a structure containing the current screen attributes: scr_mode, scr_col, and scr_off contain the screen mode, the current color attributes, and the offset from the beginning of the RAM from which to perform I/O, respectively.

34:	*gr_basaddr* points to the beginning of the video RAM.

grinit() - lines 36-43

This routine is called once, at system boot time. Its responsibility is to procure the global selector that places the video RAM within the kernel's address space.

40:	*sel* is assigned the value of an available descriptor.
41:	The descriptor is mapped to an area of memory starting at MBASE, progressing on for an additional SIZE-1 bytes, in such a way that the mapped memory can be both read from and written to (DSA_DATA).
42:	*gr_basaddr* is a **far** pointer created by using the *sotofar*() macro to combine the previously mapped segment and a 0 offset. Note that *gr_basaddr* is not a real **far** pointer, but because it has been mapped, it is a virtual **far** pointer pointing to MBASE.

gropen() - lines 45-50

Called on every open of the video RAM deviee.

47:	*gr_mode.scr_color* is set to its initial color, black.
48:	*gr_mode.scr_mode* is set to its initial mode, text.
49:	*gr_mode.scr_off* is set to point to the beginning of the video RAM.

grclose() - lines 52-58

On the final close of the video device, set the text versus graphics register back to text, and the color register back to its default black.

54-55:	Set variables appropriate for the new state.
56-57:	Call appropriate routines so that the current state is reflected in the hardware.

```
60 grread()
61 {
62     gr_rw(FREAD);
63 }
64
```

```
grwrite()
{
   gr_rw(FWRITE);
}

gr_rw(mode)
{
   register int extent;
   intcount;

   extent = SIZE - gr_mode.scr_off;
   if( extent < u.u_count)
      count = extent;
   else
      count = u.u_count;

   if(mode==FREAD)
      grcopy(gr_basaddr + gr_mode.scr_off, u.u_base, count);
   else
      grcopy(u.u_base, gr_basaddr + gr_mode.scr_off, count);

   u.u_count -=count;
   gr_mode.scr_off +=count;
}

grioctl(dev, cmd, arg, mode)
intdev, cmd, mode;
faddr_t arg;
{
   switch (cmd) {
      case ('a'):
          if((int)arg) {
             gr_mode.scr_mode |=ACOL;
             gr_mode.scr_color |= ACOL;
          } else{
              gr_mode.scr_mode &=~ACOL;
              gr_mode.scr_color &=~ACOL;
          }
          gr_setcolor( );
          break;
       case ('b'):
          gr_mode.scr_color &=0xf0;
          gr_mode.scr_color |= ((char)(arg) & 0x0f);
          gr_setcolor( );
          break;
       case ('c'):
          grclear(gr_basaddr, SIZE);
          break;
       case ('g'):
          gr_setscrn(1);
115             break;
116         case ('i'):
117             if((int)arg) {
118                 gr_mode.scr_mode |=HI_IN;
119                 gr_mode.scr_color |= HI_IN;
120             } else {
121                 gr_mode.scr_mode &=~HI_IN;
122                 gr_mode.scr_color &=~HI_IN;
123             }
124             gr_setcolor();
125             break;
126         case ('s'):
127             if ((unsigned int)arg >= SIZE) {
128                 u.u_error =EINVAL;
129                 return;
130             }
131             gr_mode.scr_off = (unsigned int)arg;
132             break;
133         case ('t'):
134             gr_setscrn(0);
135             break;
136     }
137 }
138
```

grread(), grwrite() - lines 60- 68

These simply call the combined read/write routine, *gr_rw*(), with an appropriate argument.

gr_rw() - lines 70- 88

The combined read and write routine.

75-79:	Determine how many more bytes there are between the current video position and the end of the video. If the request would extend beyond the end of the video RAM, truncate it.
81-84:	call the *grcopy*() routine to perform the actual transfer between the video RAM and the user's memory.
86-87:	Update *u.u_count* so it contains the number of bytes not read or written. Update *scr_off* so it contains the current offset into the video RAM.

grioctl() - lines 90 to 141:

This routine performs other manipulation on the screen device besides reading and writing. The clear (**c**) command also writes to screen memory, but saves the user from having to zero a 16k buffer by hand in order to clear the screen.

96-104:	The **a** command selects either the primary or the alternate color palette, depending on whether its third argument is a 1 or a 0.
105-109:	The **b** command sets the background color to the low-order 4 bits of the integer given as argument.
110-112:	The **c** command clears the screen, calling *grclear*() to perform the actual pointer manipulation.
113-115:	The **g** command sets the screen into graphics mode.
116-125:	The **i** command selects the background intensity
126-132:	The **s** command "seeks" the current read/write position to the desired byte of the RAM. Since characters are written to the RAM as whole bytes, not bits, the finest granularity available for seeks is also the byte level.
133-135:	The **t** command sets the screen back into text mode.

```
139 gr_setcolor()
140 {
141     outb(COLOR, gr_mode.scr_color);
142 }
143
144 /*
145  * setup character or graphics mode.
146  */
147 gr_setscrn(mode)
148 int mode;
149 {
150     register int i, s;
151
152     s = spl5();
153     for (i=0;i <=0x0D;i++){
154         outb(INDEX, i);
155         outb(DATA, !mode ? grchar_mode[i] : grgraf_mode[i]);
156     }
157     outb(MODE,(mode) ? SCN_GRF : SCN_TXT);
158     splx(s);
159 }
160
161 grcopy(from,to,count)
```

```
162 register faddr_t from;
163 register faddr_t to;
164 int count;
165 {
166     do{
167         *to++ = *from++;
168     } while (--count);
169 }
170
171 grclear(to,count)
172 register faddr_t to;
173 int count;
174 {
175     do{
176         *to++='\0';
177     }while(--count);
178 }
```

gr_setcolor() - lines 139 to 141

Sets the hardware COLOR register to the currently selected color set.

gr_setscrn() - lines 147 to 159

Sets the screen into graphics or text mode. Must be performed at a high priority to prevent a kernel *printf*() coming from an interrupt process from becoming mangled.

153–156:	Send the mode initialization strings to the appropriate registers.
157:	Inform the controller if this is graphics or text mode

grcopy() - lines 161 to 169

Copies *count* bytes from virtual **far** pointer *from* to virtual **far** pointer *to.*

grclear() - lines 171-178

Clears *count* bytes starting at virtual **far** pointer *to*.

Appendix A
C Language Portability

A.1 Introduction

The standard definition of the C programming language leaves many details to be decided by individual 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 in either target machine hardware or compilers. C was designed to compile to efficient code for the target machine (initially a PDP-11), so many of the language features that are 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, which is determined by the system calls and library routines it uses during execution, file pathnames 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 8-bit microprocessors to large mainframes. This appendix is concerned with the portability of C code in the XENIX programming environment. This is a more restricted problem to consider since all XENIX systems to date run on hardware with the following basic characteristics:

ASCII character set
8-bit bytes
2-byte or 4-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 is performed. This is 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. These are described briefly in a later section.

This appendix is not intended as a C language primer. It is assumed that the reader is familiar with C, and with the basic architecture of common microprocessors.

A.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. The first is to avoid using inherently non-portable language features. The second is to isolate any non-portable interactions with the environment, such as I/O to nonstandard devices. For example, programs should avoid hard-coding pathnames unless a pathname is common to all systems (e.g., */etc/passwd*).

Files required at compile-time (i.e., include files) may also introduce non-portability if the pathnames are not the same on all machines. In some cases include files containing machine parameters can be used to make the source code itself portable.

A.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 on XENIX systems.

A.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 exactly an 8-bit byte.

A.3.2 Word Length

In C, the size of the basic data types for a given implementation are not formally defined. Thus, 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 should be made. For example, on some machines **short** is the same length as **int**, whereas on others **long** is the same length as **int**.

Programs that need to know 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:

```
        #define MAXPOS  ((int)(((unsigned)-1)>> 1))
```

This is preferable to something like:

```
#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 non-portable constant.

A.3.3 Storage Alignment

The C language defines no particular layout for storage of data items relative to each other, or for storage of elements of structures or unions within the structure or union.

Some CPU's, such as the 80286, 80386, PDP-11, and M68000 require that data types longer than one byte be aligned on even byte address boundaries. Others, such as the 8086 and VAX-11 have no such hardware restriction. However, even with these machines, most compilers generate code that aligns words, structures, arrays, and long words on even addresses, or even long word addresses. Thus, on the VAX-11, the following code sequence gives *8*, even though the VAX hardware can access an int (a 4-byte word) on any physical starting address:

```
struct s_tag {
    char c;
    int i;
};
printf("%d\n", sizeof(s_tag));
```

The principal implications of this variation in data storage are that data accessed as non-primitive data types is not portable, and code that takes advantage of the architecture of a particular machine is not portable.

Thus, unions containing structures are non-portable if the union is used to access the same data in different ways. Unions are only likely to be portable if they are used to place different data in the same space at different times. For example, if the following union were used to obtain 4 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:

```
        struct s_tag st;

          ...

        write(fd, &st, sizeof(st));
```

This ensures portability of the source code. It does not produce a portable data file. Portability of data is discussed in a later section.

Note that the **sizeof** operator returns the number of bytes an object would occupy in an array. Thus on machines where structures are always aligned to begin on a word boundary in memory, the **sizeof** operator will include any necessary padding for this in the return value, even if the padding occurs after all useful data in the structure. This occurs whether or not the argument is actually an array element.

A.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. Any program that makes use of knowledge of the internal byte order in a word is not portable. For example, on some 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 could be used to access the high and low order bytes of an integer separately, and in a manner which would inhibit portability. The correct way to do this is to use mask and shift operations to extract the required byte:

```
        #define LOBYTE(i)  (i&0xff)
        #define HIBYTE(i)  ((i>>8)&0xff)
```

Note that even this operation is only applicable to machines with two bytes in 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* will be the byte value read. On other machines, the byte is read into some byte other than the low order one, and the value of *c* is different.

A.3.5 Bitfields

Bitfields are not implemented in all C compilers. When they are, no field may be larger than an **int**, and no field can overlap an **int** boundary. If necessary the compiler will leave gaps and move to the next **int** boundary.

The C language makes no guarantees about whether fields are assigned left to right, or right to left in an **int**. Thus, while bitfields may be useful for storing flags and other small data items, their use in unions to dissect bits from other data is definitely non-portable.

To ensure portability no individual field should exceed 16 bits.

A.3.6 Pointers

The C language is fairly generous in allowing manipulation of pointers, to the extent that most compilers will not object to non-portable pointer operations. The **lint** program is particularly useful for detecting questionable pointer assignments and comparisons.

The common non-portable use of pointers is the use of casts to assign one pointer to another pointer of a different data type. This almost always makes some assumption about the internal byte ordering and layout of the data type, and is therefore non-portable. In the following code, the byte order in the given array is not portable:

```
char c[4];
long *lp;

lp = (long *)&c[0];
*lp = 0x12345678L;
```

The **lint** program will issue warning messages about such uses of pointers. Code like this is very rarely necessary or valid. It is acceptable, however, when using the **malloc** function 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 **lint** will issue a warning concerning illegal type conversion. In addition, the **malloc** function is written to always return a starting address suitable for storing all types of data. **Lint** does not know this, so it gives a warning about possible data alignment problems too. 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 attract a warning message from **lint**.

The XENIX *C Language Reference* states that a pointer may 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 a **long** on some machines and **short** on others. In general, do not assume that:

```
        sizeof(char *) == sizeof(int)
```

In most implementations, the null pointer value, **NULL** is defined to be the integer value 0. This can lead to problems for functions that expect pointer arguments larger than integers. For portable code, always use:

```
        func((char *)NULL);
```

to pass a **NULL** value of the correct size.

A.3.7 Address Space

The address space available to a program running under XENIX varies considerably from system to system. On a small PDP-11 there may be only 64K bytes available for program and data combined. Larger PDP-11's and some 16 bit microprocessors allow 64K bytes of data and 64K bytes of program text. Other 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.

A.3.8 CharacterSet

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** datatype, and all characters must have positive values.

In the ASCII character set, all characters have values between zero and 127. Thus, they can all be represented in 7 bits, and on an 8-bits-per-byte machine they are all positive, whether **char** is treated as signed or unsigned.

There is a set of macros defined under XENIX in the header file */usr/include/ctype.h* that should be used for most tests on character quantities. They provide insulation from the internal structure of the character set and in most cases their names are more meaningful than the equivalent line of code. Compare:

```
        if(isupper(c))
```

to:

```
    if((c >= 'A') && (c <= 'Z'))
```

With some of the other macros, such as *isdigit* to test for a hex 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.

A.4 Compiler Differences

There are a number of C compilers running under XENIX. On PDP-11 systems there is the "Ritchie" compiler. Also on the 11, and on most other systems, there is the Portable C Compiler.

A.4.1 Signed/ Unsignedchar, 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.

A.4.2 Shift Operations

The left shift operator, "<<", shifts its operand a number of bits left, filling vacated bits with zero. This is a so-called logical shift. The right shift operator, ">>", performs a logical shift operation when applied to an unsigned quantity. When applied to a signed quantity, the vacated bits may be filled with zero (logical shift) or with sign bits (arithmetic shift). The decision is implementation-dependent, and code that uses knowledge of a particular implementation is non-portable.

The PDP-11 compilers use an arithmetic right shift. To avoid sign extension it is necessary to shift and mask out the appropriate number of high order bits:

```
        charc;

        c = (c >> 3) & 0x1f;
```

You can also avoid sign extension by usingusingthe divide operator:

```
        charc;

        c = c / 8;
```

A.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

The link editor used may also place a restriction on the number of unique characters in C external symbols.

Symbols unique in the first six characters are unique to most C language processors.

On some non-XENIX C implementations, uppercase and lowercase letters are not distinct in identifiers.

A.4.4 Register Variables

The number and type of register variables in a function depends 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 a PDP-11, up to three register declarations are significant, and they must be of type **int**, **char**, or **pointer**. While other machines and compilers may support declarations such as:

```
register unsigned short
```

this should not be relied upon.

Since the compiler ignores excess variables of register type, the most important register type variables should be declared first. Thus, if any are ignored, they will be the least important ones.

A.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**.

For example:

```
char c;

if(c == 0x80)
    ...
```

will never evaluate true on a machine which sign extends since *c* is sign extended before the comparison with *0x80*, an **int**.

The only safe comparison between **char** type and an **int** is the following:

```
char c;

if(c == 'x')
    ...
```

This is reliable because C guarantees all characters to be positive. The use of hard-coded octal constants is subject to sign extension. For example, the following program prints "ff80" on a PDP-11:

```
main()
{
    printf("%x\n",'\200');
}
```

Type conversion also takes place when arguments are passed to functions. Types **char** and **short** become **int**. Machines that sign extend **char** can give surprises. For example, the following program gives −128 on some machines:

```
        char c = 128;
        printf("%d\n", c);
```

This is because *c* is converted to **int** before passing to the 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 is to code defensively and allow for the possibility of sign extension:

```
        char c = 128;
        printf("%d\n", c & 0xff);
```

A.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 variable too. In such cases the code is dependent upon the size of various data types.

In XENIX there is an include file, */usr/include/varargs.h*, that contains macros for use in variable argument functions to access the arguments in a portable way:

```
    typedef char *va_list;
    #define va_dcl intva_alist;
    #define va_start(list) list=(char*)&va_alist
    #define va_end(list)
    #define va_arg(list,mode)
            ((mode *)(list += sizeof(mode)))[-1]
```

The *va_end()* macro is not currently required. Use of the other macros will be demonstrated by an example of the **fprintf** library routine. This has a first argument of type **FILE***, and a second argument of type **char***. Subsequent arguments are of unknown type and number at compilation time. They are determined at run time by the contents of the control string, argument2.

The first few lines of **fprintf** to declare the arguments and find the output file and control string address could be:

```
    #include <varargs.h>
    #include <stdio.h>

    int
    fprintf(va_alist)
    va_dcl
    {
        va_list ap;/* pointer to arg list    */
        char *format;
        FILE *fp;

        va_start(ap); /* initialize argpointer */
        fp = va_arg(ap, FILE *);
        format = va_arg(ap, char *);

            ...
    }
```

Note that there is just one argument declared to **fprintf**. This argument is declared by the *va_dcl* macro to be type **int**, although its actual type is unknown at compile time. The argument pointer *ap* is initialized by *va_start* to the address of the first argument. Successive arguments can be picked from the stack so long as their type is known using the *va_arg* macro. This has a type as its second argument, and this controls what data is removed from the stack, and how far the argument pointer *ap* is incremented. In **fprintf**, once the control string is found, the type of subsequent arguments is known and they can be accessed sequentially by repeated calls to *va_arg()*. For example, arguments of type **double**, **int ***, and **short** could be retrieved as follows:

```
        double dint;
        int *ip;
        short s;

        dint = va_arg(ap, double);
        ip = va_arg(ap, int *);
        s = va_arg(ap, short);
```

The use of these macros makes the code more portable, although it does assume a certain standard method of passing arguments on the stack. In particular, no holes must be left by the compiler, and types smaller than **int** (e.g., **char** and **short** on long word machines) must be declared as **int**.

A.4.7 Side Effects, Evaluation Order

The C language makes few guarantees about the order of evaluation of operands in an expression, or arguments to a function call. Thus:

```
        func(i++, i++);
```

must be considered non-portable, and even:

```
        func(i++);
```

is unwise if **func** is ever likely to be replaced by a macro, since the macro may use *i* more than once. There are certain XENIX macros commonly used in user programs; these are all guaranteed to use their argument once, and so can safely be called with a side-effect argument. The most common examples are *getc*(), *putc*(), *getchar*(), and *putchar*.

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. Thus the declaration:

```
        registerint a, b, c, d;
```

on a PDP-11 where only three register variables may be declared could make any three of the four variables register type, depending on the compiler. The correct declaration is to decide the order of importance of the variables being register type, and then use separate declaration statements, since the order of processing of individual declaration statements is guaranteed to be sequential:

```
        register int a;
        register int b;
        register int c;
        registerintd;
```

A.5 Program 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.

We are concerned here primarily with portability under the XENIX operating system. Many of the XENIX system calls are specific to that particular operating system environment and are not present on all other operating system implementations of C. Examples of this are *getpwent*() for accessing entries in the XENIX password file, and *getenv* () which is specific to the XENIX concept of a process environment.

Any program containing hard-coded pathnames to files or directories, or containing user IDs, login names, terminal lines or other system dependent parameters is non-portable. These types of constants should be in header files, passed as command line arguments, obtained from the environment, or obtained by using the XENIX default parameter library routines **dfopen**, and **dfread**.

Within XENIX, most system calls and library routines are portable across different implementations and XENIX releases. However, a few routines have changed in their user interface. The XENIX library routines are usually portable among XENIX systems.

Note that the members of the **printf** family, **printf**, **fprintf**, **sprintf**, **sscanf**, and **scanf** have changed in several ways during the evolution of XENIX, and some features are not completely portable. The return values of these routines cannot be relied upon to have the same meaning on all systems. 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 reference manual page for **printf** contains the correct specification for these routines.

A.6 Portability of Data

Data files are almost always non-portable across different machine 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 achieve data file portability is to write and read data files as one dimensional character arrays. This avoids alignment and padding problems if the data is written and read as characters, and interpreted that way. Thus, ASCII text files can usually be moved between different machine types without too many problems.

A.7 Lint

lint is a C program checker which attempts to detect features of a collection of C source files that are non-portable or incorrect C. One particular advantage of **lint** over any compiler checking is that **lint** checks function declaration and usage across source files. Neither compiler nor link editor do this.

lint will generate warning messages about non-portable pointer arithmetic, assignments, and type conversions. Passage unscathed through **lint** is not a guarantee that a program is completely portable.

A.8 Byte Ordering Summary

The following conventions are used in the tables below:

a0	The lowest physically addressed byte of the data item. *a0 + 1*, and so on.
b0	The least significant byte of the data item, *b1* being the next least significant, and so on.

Note that any program that actually makes use of the following information is guaranteed to be non-portable!

Byte Ordering for Short Types

CPU	Byte Order	
	a0	a1
PDP-11	b0	b1
VAX-11	b0	b1
8086	b0	b1
286	b0	b1
386	b0	b1
M68000	b1	b0
Z8000	b1	b0

Byte Ordering for Long Types

CPU	Byte Order			
	a0	a1	a2	a3
PDP-11	b2	b3	b0	b1
VAX-11	b0	b1	b2	b3
8086 *	b0	b1	b2	b3
8086 **	b2	b3	b0	b1
286	b0	b1	b2	b3
386	b0	b1	b2	b3
M68000	b3	b2	b1	b0
Z8000	b3	b2	b1	b0

Note that byte ordering for long types is compiler dependent (not CPU dependent) on PDP-11 and 8086 based machines. This table is based on a PDP-11 using the Ritchie compiler. 8086 * shows byte ordering for compilers using XENIX System V (little-endian) word order. 8086 ** shows byte ordering for XENIX 3.0 (big-endian) compilers. 8086 users can use **dtype**(C)to determine whether a filesystem is word-swapped.

Appendix B

C Compiler and Link Editor Error Messages

Error messages in the warning, fatal, and compilation error message categories have the same basic form:

filename (*linenumber*) : *msg-type error-number*: *message*

where *filename* is the name of the source file being compiled, *linenumber* identifies the line of the file containing the error, *msg-type* is "warning," "fatal," "error," or "command line," *error-number* is the number associated with the error, and *message* is a self-explanatory description of the error or warning. Command line error messages simply give a message about the command line, so they do not contain references to line numbers and filenames.

The messages for each category are listed below in numerical order, along with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number.

Section B.2.6, "Compiler Limits," summarizes limits imposed by the XENIX C Compiler (for example, the maximum size of a macro definition).

Note

Occasionally, the text of an error message will make reference to either an MS-DOS switch or filename. These error messages are provided for cross-development purposes and should only occur in the DOS environment.

B.2.1 Compiler Internal Error Messages

The following messages are generated in the event of an internal compiler error:

error 0: UNKNOWN ERROR
: An unforeseen error condition has been detected by the compiler.

error 124: code generation error
: The compiler could not generate code for an expression. Usually this occurs with a complex expression; try rearranging the expression.

fatal error 1: assertion count exceeds 5; stopping compilation
: The compiler performs internal consistency checks during the course of compilation. This message indicates that the consistency check failed, and the compiler cannot continue operation.

B.2.2 Command Line Messages

The following messages indicate errors on the command line used to invoke the compiler. If possible, the compiler continues operation, displaying a warning message. In some cases, command line errors are fatal and the compiler terminates processing.

Command line error 0: unknown command line error

An unforeseen error condition has been detected by the compiler. Please report this condition to the support center listed on the support information card received with your software.

Command line error 2: listing has precedence over assembly output

Two different listing options were chosen; the assembly listing is not created.

Command line error 3: a previously defined model specification has been overridden

Two different memory models are specified; the model specified later is used.

Command line error 4: unknown -A subswitch '*%c*'

A letter given with the -A option is not recognized.

Command line error 5: only one memory model allowed

You must choose one memory model; you cannot specify more than one.

Command line error 6: missing source file name

You must give the name of the source file to be compiled.

Command line error 7: too many commas

Too many commas appear on the command line.

Command line error 8: comma needed before *filename*

The fields in the command line must be set off by commas.

Command line error 9: a filename (not a path name) is required

The name of a directory is given where the name of a file is required.

Command line warning 10: ignoring unknown flag *string*

One of the options given on the command line is not recognized and is ignored.

Command line error 12: too many *option* flags, *string*

Too many letters are given with a specific option (for example, with the **-O** option).

Command line error 13: unknown option (*c*) in *option*

One of the letters in the given option is not recognized.

Command line error 14: argument list for *name* too big

The combined length of all arguments on the command line (including the program name) may not exceed 128 bytes.

Command line error 15: 80186/286/386 selected over 8086

Both the **-M0** option and either the **-M1** or **-M2** option are given; **-M1** or **-M2** takes precedence.

Command line error 16: optimizing for space over time

This message confirms that the **-Os** option is used for optimizing.

Command line error 16: optimizing for space over time
: This message confirms that the **/Os** option is used for optimizing.

Command line error 17: unknown floating point option
: The specified floating-point option (an **/FP** option) is not one of the five valid options.

Command line error 18: only one floating point model allowed
: You can only give one of the five floating-point (**/FP**) options on the command line.

Command line error 19: could not execute *filename*
: The specified compiler file could not be found.

Command line error 20: could not execute *name*. Please insert diskette and press any key.
: One of the compiler passes cannot be found on the current disk. Insert the disk containing the named file and press any key.

B.2.3 Warning Error Messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking. The number in square brackets ([ ]) at the end of each message gives the minimum warning level that must be set for the message to appear.

warning 0: unknown error
: Internal error. Contact technical support.

warning 1: macro *identifier* requires parameters [1]
: The given *identifier* was defined as a macro taking one or more arguments, but the *identifier* is used in the program without arguments.

warning 2: too many actual parameters for macro *identifier* [1]
: The number of actual arguments specified with an identifier is greater than the number of formal parameters given in the macro definition of the identifier.

warning 3: not enough actual parameters for macro *identifier* [1]
: The number of actual arguments specified with an identifier is less than the number of formal parameters given in the macro definition of the identifier.

warning 4: missing close parenthesis after 'defined' [1]
: The closing parenthesis is missing from an **#if defined** phrase.

warning 5: *identifier* : redefinition [1]
: The given *identifier* is redefined.

warning 6: #undef expected an identifier
: The name of the identifier whose definition is to be removed must be given with the **#undef** directive.

warning 7: unmatched close comment '*/' [1]
: A comment is started (with '/*') but is not closed (with '*/').

warning 8: newline in string constant [1]
: A newline character is not preceded by an escape character (\) in a string constant.

warning 9: string too big, leading chars truncated [1]
A string exceeds the compiler limit on string size. To correct this problem, you must break the string down into two or more strings.

warning 10: illegal null char [1]
The single quotes delimiting a character constant must contain one character. For example, the declaration "char a = " is illegal. To represent a null character constant, use an escape sequence (for example, \0).

warning 11: identifier truncated to "*identifier*" [1]
Only the first 31 characters of an identifier are significant.

warning 13: constant too big [1]
Information is lost because a constant value is too large to be represented in the type to which it is assigned.

warning 14: *identifier* : bitfield type must be unsigned [1]
Bitfields must be declared as **unsigned** integral types. A conversion has been supplied.

warning 15: *identifier* : bitfield type must be integral [1]
Bitfields must be declared as **unsigned** integral types. A conversion has been supplied.

warning 16: *identifier* : no function return type [2]
The return type is missing from a function declaration; the default return type will be **int**.

warning 17: cast of int expression to far pointer [1]
A **far** pointer represents a full segmented address. On an 8086/80286 processor, casting an **int** value to a **far** pointer produces an address with a meaningless segment value.

warning 18: *identifier* : uses undefined struct/union *identifier* [2]
The name of a structure or **union** type is used before the type is defined.

warning 19: '*identifier*' : unknown size [1]
The size of the named variable is not specified.

warning 20: too many actual parameters [1]
The number of arguments specified in a function call is greater than the number of parameters specified in the argument type list or in the function definition.

warning 21: too few actual parameters [1]
The number of arguments specified in a function call is less than the number of parameters specified in the argument type list or in the function definition.

warning 22: pointer mismatch: parameter *n* [1]
The given parameter has a different pointer type than is specified in the argument type list or the function definition.

warning 23: string ignored (must also specify string)
The declaration of the given parameter specifies a **union** type, but the parameter's type does not correspond to the type of any of the union members.

warning 24: different types : parameter *n*
The type of the given parameter in a function call does not agree with the argument type list or the function definition.

warning25: function declaration specified variable args [1]
: The argument type list in a function declaration ends with a comma, indicating that the function can take a variable number of arguments, but no formal parameters for the function are declared.

warning26: function was declared with formal arguments [1]
: The function was declared to take arguments, but the function definition does not declare formal parameters.

warning27: function was declared without formal argument list [1]
: The function was declared to take no argument (the argument type list consists of the word **void**) but formal parameters are declared in the function definition or arguments are given in a call to the function.

warning28: parameter *n* declaration different
: The type of the given parameter does not agree with the corresponding type in the argument type list or with the corresponding formal parameter.

warning29: declared parameter list differs from definition [1]
: The argument type list given in a function declaration does not agree with the types of the formal parameters given in the function definition.

warning30: first parameter list is longer than the second [1]
: A function is declared more than once and the argument type lists in the declarations differ.

warning31: second parameter list is longer than the first [1]
: A function is declared more than once, and the argument type lists in the declarations differ.

warning32: unnamed struct/union as parameter [1]
: The structure or **union** type being passed as an argument is not named, so the declaration of the formal parameter cannot use the name and must declare the type.

warning33: function must return a value [2]
: A function is expected to return a value unless it is declared as **void**.

warning34: sizeof returns 0 [2]
: The **sizeof** operator is applied to an operand that yields a size of zero.

warning35: no return value [2]
: A function declared to return a value does not do so.

warning36: unexpected formal parameter list [1]
: A formal parameter list is given in a function declaration and is ignored.

warning37: '*identifier*' : formal parameters ignored [1]
: Formal parameters appeared in a function declaration (for example, "extern int *f(a,b ,c);"). The formal parameters are ignored.

warning38: *identifier* : formal parameter has bad storage class [1]
: Formal parameters must have **auto** or **register** storage class.

warning 39: '*identifier*' : function used as an argument [1]
A formal parameter to a function is declared to be a function, which is illegal. The formal parameter is converted to a function pointer.

warning 40: near/far/huge on *identifier* ignored [1]
The **near**, **far**, and **huge** keywords have no effect in the declaration of the given *identifier* and are ignored.

warning 41: formal parameter *identifier* is redefined [1]
The given formal parameter is redefined in the function body, making the corresponding actual argument unavailable in the function.

warning 42: '*identifier*' : has bad storage class [1]
The specified storage class cannot be used in this context (for example, function parameters cannot be given class). The default storage class for that context is used in place of the illegal class.

warning 43: '*identifier*' : void type changed to int [1]
Only functions may be declared to have **void** type.

warning 44: huge on '*identifier*' ignored, must be an array [1]
The **huge** keyword can only be used in array declarations.

warning 45: '*identifier*' : array bounds overflow [1]
Too many initializers are present for the given array. The excess initializers are ignored.

warning 46: '&' on function/array, ignored [1]
You cannot apply the address-of operator to a function or array identifier.

warning 47: '*operator*': different levels of indirection [1]
An expression involving the specified operator has inconsistent levels of indirection. For example,

```
char**p;
char*q;

.
.
.
p=q;
/* different levels of indirection */
```

warning 48: array's declared subscripts different [1]
An array is declared twice with differing sizes. The larger size is used.

warning 49: *operator* : indirection to different types [1]
The indirection operator (*) is used in an expression to access values of different types.

warning 50: strong type mis-match [2]
Two different but compatible types are used: for example, a **typedef** type with a non-**typedef** type, or two different but equivalent **struct** or **union** types.

warning 51: data conversion [3]
    Two data items in an expression had different types, causing the type of one item to be converted.

warning 52: different enum types [1]
    Two different **enum** types are used in an expression.

warning 53: at least one void operand [1]
    An expression with type **void** is used as an operand.

warning 54: '*operator*' : illegal with enums [1]
    You may not use the given operator with an **enum** value. The **enum** value is converted to **int** type.

warning 55: type following '*keyword*' is illegal, ignored [1]
    An illegal combination occurs (for example, **unsigned float**).

warning 56: overflow in constant arithmetic [1]
    The result of an operation exceeds 0x7FFFFFFF.

warning 57: overflow in constant multiplication [1]
    The result of an operation exceeds 0x7FFFFFFF.

warning 59: conversion lost segment [1]
    The conversion of a **far** pointer (a full segmented address) to a **near** pointer (a segment offset) results in the loss of the segment address.

warning 60: conversion of a long address to a short address [1]
    The conversion of a long address (a 32-bit pointer) to a short address (a 16-bit pointer) results in the loss of the segment address.

warning 61: long/short mismatch in arguments : conversion supplied [1]
    An integral type is assigned to an integer of a different size, causing a conversion to take place. For example, a **long** is given where a **short** was declared, etc.

warning 62: near/far mismatch in arguments: conversion supplied [1]
    A pointer is assigned to a pointer with a different size, resulting in the loss of a segment address from a **far** pointer or the addition of a segment address to a **near** pointer.

warning 63: function *identifier* too large for post-optimizer [0]
    The named function was not optimized because insufficient space was available. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning 64: procedure too large, skipping [loop inversion *or* branch sequence *or* cross jump] optimization and continuing[0]
    Some optimizations for a function are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning 65: recoverable heap overflow in post optimizer - some optimizations may be missed [0]
    Some optimizations are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning 66: local symbol table overflow [0]
The compiler has run out of memory. Remove some declarations and try recompiling.
warning 67: unexpected characters following '% s' directive [1]
The ANSI standard requires that only white space follow a line with a **#endif** or **#else** directive. Use comments.
warning 68: unknown pragma [1]
A pragma has been given that the compiler does not know how to interpret.
warning 69: conversion of near pointer to long integer [1]
A near pointer is being converted to a long integer, which involves first extending the high-order word with the current data-segment value.
warning 72: missing semi-colon [1]
The compiler sees that a semicolon is missing and inserts one.
warning 73: scoping too deep, deepest scoping merged when debugging [1]
The **-Zi** option has been used and static nesting level is greater than 13. Variables declared at that level or higher all appear to have been declared at the same level when debugging.
warning 74: non standard extension used - *'description'* [3]
A language feature has been used that is not contained in the ANSI standard for C.
warning 75: size of switch expression or CASE constant too large - converted to int
Long value of expression truncated to **int**.

B.2.4 Fatal Error Messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it terminates after displaying the error message.
fatal error 0: unknown fatal error
fatal error 1: assertion count exceeds 5; stopping compilation
More than five assertion errors have accumulated, and the compiler cannot continue processing.
fatal error 2: out of heap space
The compiler has run out of dynamic memory space. This usually means that your program has many symbols and complex expressions. To correct the problem, break down the file into several smaller source files.
fatal error 3: error count exceeds *n*; stopping compilation
Errors in the program are too numerous or too severe to allow recovery, and the compiler must terminate.
fatal error 4: 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.

fatal error 6: write error on compiler intermediate file
: The compiler is unable to create the intermediate files used in the compilation process. The exact reason is unknown.

fatal error 7: unrecognized flag *string* in *pass*
: One of the options given on the command line is not recognized and the file cannot be processed.

fatal error 8: no input file specified
: You must give at least one source file as input to the compiler.

fatal error 9: compiler limit : possibly a recursively defined macro
: The expansion of a macro exceeds the available space. Check to see whether the macro is recursively defined, or if the expanded text is too large.

fatal error 10: compiler limit : macro expansion too big
: The expansion of a macro exceeds the available space.

fatal error 11: recursively defined macro *identifier*
: The given identifier is defined recursively.

fatal error 12: bad parenthesis nesting
: The parentheses in a preprocessor directive are not matched.

fatal error 13: cannot open *filename*
: The given file cannot be opened.

fatal error 14: too many include files
: Nesting of **#include** directives exceeds the limit of ten levels.

fatal error 15: cannot find *filename*
: The given file does not exist or cannot be found. Check to make sure your environment settings are valid and that you have given the correct pathname for the file.

fatal error 16: #if[n]def expected an identifier
: You must specify an identifier with the **#ifdef** and **#ifndef** directives.

fatal error 17: invalid integer constant expression
: The expression in an **#if** directive must evaluate to a constant.

fatal error 18: unexpected '#elif'
: The **#elif** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

fatal error 19: unexpected '#else'
: The **#else** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

fatal error 20: unexpected '#endif'
: An **#endif** directive appears without a matching **#if**, **#ifdef**, or **#ifndef** directive.

fatal error 21: bad preprocessor command '*string*'.
: The characters following the number sign (#) do not form a preprocessor directive.

fatal error 22: expected '#endif'
: An **#if**, **#ifdef**, or **#ifndef** directive was not terminated with an **#endif** directive.

fatal error 26: 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.

fatal error 27: DGROUP data allocation exceeds 64K
: Large or huge model allocation of variables to the default segment exceeds 64K on an 86/286 processor; use the **/GT** option to move items into separate segments.

fatal error 32: cannot open listing file *filename*
: The filename or pathname given for the listing file is not valid.

fatal error 33: cannot open assembly language output file *filename*
: The filename or pathname given for the assembly language output file is not valid.

fatal error 34: cannot open source file *filename*
: The filename or pathname given for the source file is not valid.

fatal error 41: cannot open compiler intermediate file
: The compiler is unable to create intermediate files used in the compilation process because no more file handles are available. This can usually be corrected by changing the "files =" line in CONFIG.SYS to allow a larger number of open files (60 is the recommended setting).

fatal error 42: cannot open compiler intermediate file – no such file or directory
: The compiler is unable to create intermediate files used in the compilation process because the TMP environment variable is set to an invalid directory or path.

fatal error 43: cannot open compiler intermediate file
: The compiler is unable to create intermediate files used in the compilation process. The exact reason is unknown.

fatal error 44: out of disk space for compiler intermediate file
: The compiler is unable to create intermediate files used in the compilation process because no more space is available. To correct the problem, make more space available on the disk and recompile.

fatal error 50: code segment too large
: Segment larger than 4 gigabytes on 80386, 64K on 8086 and 80286.

B.2.5 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 displays additional error messages. However, no object file is produced.

error 0: UNKNOWN ERROR
: An unforeseen error condition has been detected by the compiler. Please report this condition to the support center listed on the support information card received with your software.

error 1: newline in constant
  A newline character in a character or string constant must be preceded by the backslash escape character (\).

error 2: out of macro actual parameter space
  Arguments to preprocessor macros may not exceed 256 bytes.

error 3: missing open paren after keyword 'defined'
  Parentheses must surround the identifier to be checked in an **#if** directive.

error 4: expected 'defined(id)'
  An **#if** directive has a syntax error.

error 5: #line expected a line number
  A **#line** directive lacks the mandatory line number specification.

error 6: #include expected a file name
  An **#include** directive lacks the mandatory filename specification.

error 7: #define syntax
  A **#define** directive has a syntax error.

error 8: '*c*' : unexpected in macro definition
  The character *c* is misused in a macro definition.

error 9: reuse of macro formal *identifier*
  The parameter list in a macro definition contains two occurrences of the same identifier.

error 10: '*c*' : unexpected in formal list
  The character *c* is misused in a macro definition's list of formal parameters.

error 11: '*identifier*' : definition too big
  Macro definitions may not exceed 256 bytes.

error 12: missing name following '<'
  An **#include** directive lacks the mandatory filename specification.

error 13: missing '>'
  The closing angle bracket ('>') is missing from an **#include** directive.

error 14: preprocessor command must start as first non-whitespace
  Non-whitespace characters appear before the number sign (#) of a preprocessor directive on the same line.

error 15: too many chars in constant
  A character constant is limited to a single character or escape sequence. (Multicharacter character constants are not supported.)

error 16: no closing single quote
  A newline character in a character constant must be preceded by the backslash escape character (\).

error 17: illegal escape sequence
  The character(s) after the escape character (\) do not form a valid escape sequence.

error 18: unknown character '0x*n*'
  The given hexadecimal number does not correspond to a character.

error 19: expected preprocessor command, found '*c*'
The character following a number sign (#) is not the first letter of a preprocessor directive.

error 20: bad octal number '*n*'
The character *n* is not a valid octal digit.

error 21: expected exponent value, not '*n*'
The exponent of a floating point constant is not a valid number.

error 22: '*n*' : too big for char
The number *n* is too large to be represented as a character.

error 23: divide by 0
The second operand in a division operation (/) evaluates to zero, giving undefined results.

error 24: mod by 0
The second operand in a remainder operation (%) evaluates to zero, giving undefined results.

error 25: '*identifier*' : enum/struct/union type redefinition
The given *identifier* has already been used for an enumeration, structure, or union tag.

error 26: '*identifier*' : member of enum redefinition
The given *identifier* has already been used for an enumeration constant, either within the same enumeration type or within another enumeration type with the same visibility.

error 27: compiler limit : struct/union nesting
Nesting of structure and union definitions may not exceed five levels.

error 28: struct/union member needs to be inside a struct/union
Structure and union members must be declared within the structure or union.

error 29: '*identifier*' : fields only in structs
Only structure types may contain bitfields.

error 30: struct/union member redefinition
The same identifier was used for more than one structure or union member.

error 31: '*identifier*' : function cannot be struct/union member
A function cannot be a member of a structure; use a pointer to a function instead.

error 32: '*identifier*' : base type with near/far/huge not allowed
Declarations of structure and union members may not use the **near**, **far**, and **huge** keywords.

error 33: '*identifier*' : field has indirection
The bitfield is declared as a pointer (*), which is not allowed.

error 34: '*identifier*' : field type too small for number of bits
The number of bits specified in the bitfield declaration exceeds the number of bits in the given **unsigned** type.

error 35: '*identifier*' : unknown size
A member of a structure or union has an undefined size.

error 36: left of '*->identifier*' or '*.identifier*' must have struct/union type
The expression before the member selection operator '->' is not a pointer to a structure or union type, or the expression

before the member selection operator '.' does not evaluate to a structure or union.

error 37: left of '->' or '.' specifies undefined struct/union '*identifier*'
: The expression before the member selection operator '->' or '.' identifies a structure or union type that is not defined.

error 38: '*identifier*' : not struct/union member
: The given *identifier* is used in a context that requires a structure or union member.

error 39: '->' requires struct/union pointer
: The expression before the member selection operator '->' is not a pointer to a structure or union.

error 40: '.' requires struct/union name
: The expression before the member selection operator '.' is not the name of a structure or union.

error 41: keyword 'enum' illegal
: The **enum** keyword appears in a structure or union declaration, or an **enum** type definition is not formed correctly.

error 42: keyword 'enum' required
: The **enum** keyword is required in declarations of enumeration types.

error 43: illegal break
: A **break** statement is legal only when it appears within a **do**, **for**, **while**, or **switch** statement.

error 44: illegal continue
: A **continue** statement is legal only when it appears within a **do**, **for**, or **while** statement.

error 45: *identifier* : label redefined
: The given *identifier* appears before more than one statement in the same function.

error 46: illegal case
: The **case** keyword may only appear within a **switch** statement.

error 47: illegal default
: The **default** keyword may only appear within a **switch** statement.

error 48: more than one default
: A **switch** statement contains too many **default** labels (only one is allowed).

error 49: cast has illegal formal parameter list
: A formal parameter list is given in a type cast expression.

error 50: non-integral switch expression
: Switch expressions must be integral.

error 51: case expression not constant
: Case expressions must be integral constants.

error 52: case expression not integral
: Case expressions must be integral constants.

error 53: case value '*n*' already used
: The case value *n* has already been used in this **switch** statement.

error 54: expected (to follow '*identifier*'
The context requires parentheses after the function *identifier*.
error 55: expected formal parameter list, not a type list
An argument type list appears in a function definition instead of a formal parameter list.
error 56: illegal expression
An expression is illegal because of a previous error. (The previous error may not have produced an error message.)
error 57: expected constant expression
The context requires a constant expression.
error 58: constant expression is not integral
The context requires an integral constant expression.
error 59: syntax error : '*token*'
The given *token* caused a syntax error.
error 60: syntax error : EOF
The end of the file was encountered unexpectedly, causing a syntax error.
error 61: syntax error : identifier *identifier*
The given *identifier* caused a syntax error.
error 62: type '*identifier*' unexpected
The given type is misused.
error 63: '*identifier*' : not a function
The given *identifier* was not declared as a function, but an attempt was made to use it as a function.
error 64: term does not evaluate to a function
An attempt is made to call a function through an expression that does not evaluate to a function pointer.
error 65: '*identifier*' : undefined
The given *identifier* is not defined.
error 66: cast to function returning... is illegal
An object cannot be cast to a function type.
error 67: cast to array type is illegal
An object cannot be cast to an array type.
error 68: illegal cast
A type used in a cast operation is not a legal type.
error 69: cast of 'void' term to non-void
The **void** type may not be cast to any other type.
error 70: illegal sizeof operand
The operand of a **sizeof** expression must be an identifier or a type name.
error 71: '*class*' : had storage class
The given storage *class* cannot be used in this context.
error 72: '*identifier*': initialization of a function
Functions may not be initialized.
error 73: *identifier* : cannot initialize array in function
Arrays can only be initialized at the external level.
error 74: cannot initialize struct/union in function
Structures and unions can only be initialized at the external level.

error 75: '*identifier*' : array initialization needs curly braces
The braces ({ }) around an array initializer are missing.

error 76: struct/union initialization needs curly braces
The braces ({ }) around a structure or union initializer are missing.

error 77: non-integral field initializer *identifier*
An attempt is made to initialize a bitfield member of a structure with a non-integral value.

error 78: too many initializers.
The number of initializers exceeds the number of objects to be initialized.

error 79: *identifier* is an undefined struct/union
The given *identifier* is declared as a structure or union type that has not been defined.

error 80: '*expression*' was the use of the struct/union
An undefined structure or union type variable is used in the given *expression*.

error 81: compiler limit : initializers too deeply nested
The compiler limit on nesting of initializers has been 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.

error 82: redefinition of formal parameter *identifier*
A formal parameter to a function is redeclared within the function body.

error 83: array '*identifier*' already has a size
The dimensions of the given array have already been declared.

error 84: function '*identifier*' already has a body
The given function has already been defined.

error 85: '*string*' : ignored
The given text appeared out of context and was ignored.

error 86: '*identifier*' : redefinition
The given *identifier* was defined more than once.

error 87: '*identifier*' : missing subscript
To reference an element of an array you must use a subscript.

error 88: use of undefined struct/union *identifier*
The given *identifier* was used to refer to a structure or union type that is not defined.

error 89: typedef specifies a near/far function
The near or far keyword is used in a typedef declaration.

error 90: function returns array
A function may not return an array. (It may return a pointer to an array.)

error 91: function returns function
A function may not return a function. (It may return a pointer to a function.)

error 92: array element type cannot be function
Arrays of functions are not allowed.

error 94: label '*identifier*' was undefined.
The function does not contain a statement labeled with the given *identifier*.

error 95: parameter has type void
Formal parameters and arguments to functions may not have **void** type.

error 96: struct/union comparison illegal
You cannot compare two structures or unions. (You can, however, compare individual members of structure and unions.)

error 97: illegal initialization
An initialization is illegal because of a previous error. (The previous error may not have produced an error message.)

error 98: non-address expression
An attempt was made to initialize an item that is not an lvalue.

error 99: non-constant offset
An initializer uses a non-constant offset.

error 100: illegal indirection.
The indirection operator (*) was applied to a non-pointer value.

error 101: '&' on constant
Only variables and functions can have their address taken.

error 102: '&' requires lvalue
The address-of operator can only be applied to lvalue expressions.

error 103: '&' on register variable
Register variables cannot have their address taken.

error 104: '&' on bit field ignored
Bitfields cannot have their address taken.

error 105: '*operator*' needs lvalue.
The given *operator* must have an lvalue operand.

error 106: *operator* : left operand must be lvalue
The left operand of the given *operator* must be an lvalue.

error 107: illegal index, indirection not allowed
A subscript was applied to an expression that does not evaluate to a pointer.

error 108: non-integral index
Only integral expressions are allowed in array subscripts.

error 109: subscript on non-array.
A subscript was used on a variable that is not an array.

error 110: '+' : 2 pointers
Two pointers may not be added.

error 111: pointer + non-integral value
Only integral values may be added to pointers.

error 112: illegal pointer subtraction
Only pointers that point to the same type may be subtracted.

error 113: '–' : right operand pointer
: The right-hand operand in a subtraction operation (–) is a pointer, but the left-hand operand is not.

error 114: '*operator*' : pointer on left; needs integral right
: The left operand of the given *operator* is a pointer; the right operand must be an integral value.

error 115: *identifier* : incompatible types
: An expression contains types that are not compatible.

error 116: *operator* : bad left *or* right operand
: The specified operand of the given *operator* is an illegal value.

error 117: '*operator*' : illegal for struct/union
: Structure and union type values are not allowed with the given *operator*.

error 118: negative subscript
: A value defining an array size was negative.

error 119: 'typedefs' both define indirection
: Two **typedef** types are used to declare an item and both **typedef** types have 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;
```

error 120: 'void' illegal with all types
: The **void** type cannot be used in operations with other types.

error 121: typedef specifies different enum
: Two different enumeration types defined with **typedef** are used to declare an item, but the enumeration types are different.

error 122: typedef specifies different struct
: Two structure types defined with **typedef** are used to declare an item, but the structure types are different.

error 123: typedef specifies different union
: Two union types defined with **typedef** are used to declare an item, but the union types are different.

error 124: code generation error
: The compiler could not generate code for an expression. Usually this occurs with a complex expression. Trying rearranging the expression.

error 125: allocation exceeds 64K for *identifier*
: The given item exceeds the limit of 64K. The only items that are allowed to exceed 64K on an 86/286 processor are **huge** arrays.

error 126: auto allocation exceeds 32K
: The space allocated for the local variables of a function exceeds the limit of 32 kilobytes.

error 127: parameter allocation exceeds 32K
The storage space required for the parameters to a function exceeds the limit of 32 kilobytes.

error 128: huge *'identifier'* cannot be aligned to segment boundary
The given array violates one of the restrictions imposed on **huge** arrays; review the discussion of these restrictions for details.

error 129: static procedure *'identifier'* not found.
A forward reference was made to a missing static procedure.

error 130: #line expected a string containing the filename
Invalid syntax for #line directive (missing file name).

error 131: attributes specify more than one near/far
More than one **near** or **far** attribute applied to an item.

error 132: syntax error: unexpected identifier
Identifier seen in a syntactically illegal context.

error 133: array '%s': unknown size
Attempt to declare unsized array as local variable.

error 134: symbol too large
Size of array exceeds compiler limit (2^16 in 80286 mode or 2^32 bytes in 80386 mode).

error 135: missing ')' in macro expansion
A macro reference with arguments is missing a closing parenthesis.

error 137: empty character constant
An illegal character constant was used.

error 138: unmatched close comment
Compiler detected */ without matching /*. This usually indicates an attempt to use illegal nested comments.

error 139: type following '%s' is illegal
Illegal type combination.

error 140: argument type cannot be function returning
A function is declared as a formal parameter of another function.

error 141: value out of range for enum constant
An enum constant has a value outside the range of values allowed for integer types.

error 142: ellipsis requires three periods
The compiler has detected the token .. and assumes ... was intended.

error 143: syntax error: missing '%s' before '%s'
Invalid token. Replace and recompile.

error 144: syntax error: missing '%s' before type '%s'
Invalid token. Replace and recompile.

error 145: syntax error: missing '%s' before identifier
Invalid token. Replace and recompile.

error 146: syntax error: missing '%s' before identifier '%s'
Invalid token. Replace and recompile.

error 147: array size unknown
Array size is not known.

B.2.6 Compiler Limits

To operate the 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.

The following table 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.

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 XENIX *C Language Reference* 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.	5 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 displays an error message to that effect.

B.3 Link Editor Error Messages

The error messages produced by the C linker fall into three categories:

- Fatal-error messages
- Linker error messages
- Warning messages

Fatal-error messages indicate a severe problem, one that prevents the linker from processing the object code. After printing a message about the fatal error, the linker terminates linking without producing an executable object file or checking for further errors. Fatal-error messages have the following form:

<location> : fatal error L1xxx: <messagetext>

Linker error messages indicate a problem in the executable object file. After printing a message, the linker produces the executable file and sets the error bit in the header. Linker error messages have the following form:

<location> : error L2xxx: <message text>

Warning messages are informational only; they do not prevent the linker from processing the relocatable object code into executable object code. Rather, warning messages just indicate possible problems in the executable object file. Warning messages have the following form:

<location> : warning L4xxx: <messagetext>

In the messages, *<location>* represents the input file, or path name of the linker if an input file is not present. The *xxx* represents the message number, and *<messagetext>* defines the message.

B.4 Linker Fatal-Error Messages

The following messages identify fatal errors. The linker can not recover from a fatal error, instead the linker terminates linking after printing the fatal-error message.

fatal error L1002: unrecognized option name
: An unrecognized character was given following – on the command line.

fatal error L1004: badly formed number
: An invalid numeric value was given for an option.

fatal error L1008: segment limit set too high
: The number following the -S option is larger than 1024, which is the largest number allowed.

fatal error L1011: badly formed hex number
: An invalid hexadecimal numeric value was given with an option.

fatal error L1012: number too large
A number was appended to an option greater than 2^32-1.
fatal error L1013: version number missing
The -v option was given without a version number.
fatal error L1014: unrecognized Xenix version number
The number following the -v option must be either 2, 3, or 5.
fatal error L1015: address missing
The -A option requires an appended number.
fatal error L1016: -A and -F are mutually exclusive
The -A and -F options are mutually exclusive.
fatal error L1018: Pagesize value missing
The -N option was given without a following pagesize number.
fatal error L1019: pagesize larger than 0xfe00
The -N option is given with a pagesize value larger than 0xfe00, which is the largest allowed.
fatal error L1020: no object modules specified
No object modules are specified on the command line. At least one must be specified for the linker to produce an output file.
fatal error L1023: terminated by user
An interrupt was issued while the linker was operating.
fatal error L1045: too many TYPDEF records
An object module contains more than 255 **TYPDEF** records.
fatal error L1046: too many external symbols in one module
An object module specifies more than the maximum limit of 1023 external symbols. Break the module into smaller parts or reduce the number of external references.
fatal error L1047: too many group, segment, and class names in one module
An object module contains too many group, segment, and class names. Reduce the number of groups, segments, or classes in the module.
fatal error L1048: too many segments in one module
An object module has more than the maximum limit of 255 segments. Split the module or combine segments.
fatal error L1049: too many segments
The program contains more than the default maximum limit of 128 segments. Relink the program using the -S option, assigning an appropriate number of segments.
fatal error L1050: too many groups in one module
The module contains more than the maximum limit of 21 group definitions (GRPDEF records). Split the module or redefine group definitions.
fatal error L1051: too many groups
The program contains more than the maximum limit of 20 groups, not counting DGROUP. Reduce the number of group definitions.

fatal error L1053: symbol table overflow

The program contains more than the maximum limit of 512K symbols, such as public, external, segment, group, class, and file names. Reduce the number of symbols.

fatal error L1054: requested segment limit too high

The linker does not have have sufficient memory to describe the number of segments requested by the -S option. Reduce the segment argument to a number below 1024.

fatal error L1057: data record too large

An **LEDATA** record contains more than the maximum limit of 1024 bytes of data. Note which translator, compiler or assembler produced the incorrect object module.

fatal error L1070: segment size exceeds 64K

A single 16-bit segment contains more than 64K of code or data. Reduce the size of the segment to less than 64K.

fatal error L1072: common area longer than 65536 bytes

The program has more than the maximum limit of 64K of communal variables allowed for 8086 and 80286 executable files. Note that this error is not generated by the macro assembler, but only by compilers supporting communal variables.

fatal error L1075: segment size exceeds <*number*>

A 32-bit segment exceeds the maximum limit of code or data imposed by the linker, which is indicated by *number*. Reduce the size of the segment.

fatal error L1076: common area longer than 4G-1 bytes

The program has more than the maximum limit of 4 gigabytes-1 of communal variables allowed for 80386 executable files. Note that this error is not generated by the macro assembler, but only by compilers supporting communal variables.

fatal error L1080: cannot open list file

The linker can not create the list (map) file.

fatal error L1081: out of space for run file

The disk on which the executable output file is being written to is full. Free more space on the disk and restart the linker.

fatal error L1083: cannot open run file

The disk on which the executable output file is being written to is full or the file already exists with read-only permissions. Free more space on the disk or change permissions.

fatal error L1085: cannot open temporary file

The disk on which the temporary file is being written to is full. Free more space on the disk and restart the linker.

fatal error L1086: scratch file missing

The linker is unable to open a temporary file recently created. Restart the linker.

fatal error L1087: unexpected end-of-file on scratch file

A temporary file recently created by the linker was unexpectedly reduced in size. Restart the linker.

fatal error L1088: out of space for list file

The disk on which the list file is being written to is full. Free more space on the disk and restart the linker.

fatal error L1091: unexpected end-of-file on library

All required data in the library file was not read before encountering the end-of-file. Replace the library file and restart the linker.

fatal error L1093: object not found

The object module specified on the command line does not exist. Restart the linker, verifying the correct object-module path name.

fatal error L1101: invalid object module

One of the object modules specified on the command line is invalid. Restart the linker. If the fatal error persists, contact your XENIX system administrator.

fatal error L1103: attempt to access data outside segment bounds

A data record in an object module specifies data extending beyond the end of the segment. Note which translator, assembler or compiler produced the incorrect object module and notify your XENIX system administrator.

fatal error L1113: unresolved COMDEF, internal error

An internal error has occurred; notify your XENIX system administrator.

fatal error L1120: use -i option

The program uses more than one segment and it is being linked as impure. Impure executable files can have only one segment.

fatal error L1121: *<name>*: group larger than 4G-1 bytes

The *name* 32-bit group contains segments larger than 4 gigabytes-1.

fatal error L1122: *<name>*: group larger than 64K bytes

The *name* 16-bit group contains segments larger than 64K.

fatal error L1123: *<name>*: both 16-bit and 32-bit segments in group

The *name* group contains both 16-bit and 32-bit segments.

fatal error L1124: relocation value missing

The **-Rt** or **-Rd** option was given without an argument.

fatal error L1125: stack size missing

The **-F** option was given without an argument.

B.4.1 Run-Time Library Error Messages

The following message may be generated at run time when your program has a serious error.

error 2000: Stack overflow

> Your program has run out of stack space. This can occur when a program uses a large amount of local data or is heavily recursive. The program is terminated with an exit status of 255. To correct the problem, allocate a larger stack size by using the **-F** option on the **cc** compile line.

Note

Attempting to divide an integer by 0 is an error. However, unless you make provisions to trap the signal, this error will be ignored. The error signal is 04, SIGILL, illegal instruction.

B.4.2 Run-Time Limits

Table B.2 summarizes the limits that apply to programs at run time. If your program exceeds one of these limits, an error message will inform you of the problem.

Program Limits at Run Time

Program Item	Description	Limit
Files	Maximum file size	$2^{32}-1$ bytes (4 gigabytes)
	Maximum number of open files (streams)	60[†]
Command Line	Maximum number of characters (including program name)	128
Environment Table	Maximum size	32K

[†] Three streams are opened automatically (*stdin*, *stdout*, and *stderr*) leaving 57 available for the program to open.

B.5 Linking Error Messages

The following messages identify errors in the executable object file. The linker continues processing upon encountering these errors.

error L2001: fixup(s) without data
: A FIXUPP record occurs without a data record immediately preceding it. Note which translator, compiler or assembler, produced the incorrect object file and notify your XENIX system administrator.

error L2002: fixup overflow near *num* in frame segment *name* target segment *segment name* target offset *number*
: Some possible causes are: (1) A group is larger than 64K; (2) the user's program contains an intersegment short jump or intersegment short call; (3) the user has a data item whose name conflicts with that of a subroutine in a library included in the link; and (4) the user has an EXTRN declaration inside the body of a segment. For example: CODE segment public 'code' extern main:far start proc far
  call main
  ret start endp CODE ends The following construction is preferred: extern main:far CODE segment public 'code' start proc far
  call main
  ret start endp CODE ends Revise the source and recreate the object file.

error L2011: *name* : NEAR/HUGE conflict
: Both the near and huge attributes are given for the *name* communal variable. This error only occurs in programs produced by compilers supporting communal variables.

error L2012: *name* : array-element size mismatch
: The far *name* communal array is declared with two or more different array-element sizes. This error only occurs in programs produced by compilers supporting communal variables.

error L2025: *name* : symbol defined more than once
: The public *name* symbol is defined more than once. Use only one declaration.

error L2029: unresolved externals
: One or more symbols are declared external, but they are not declared in any other module or library. A list of unresolved external references appears after the error messages in the following form:

  unresolved_external_symbol in file(s)

  file ...

  The *unresolved_external_symbol* is the symbol that is not resolved and *file* is the file(s) that references the symbol.

B.6 Linker Warning Error Messages

The following messages identify errors in the relocatable object file to be processed, or the path name of the linker, if the object file is not given.

warning L4020: <*name*> : code segment size exceeds 65500

The 16-bit code segment *name*, of length 65,501 to 65,536 bytes, is unreliable on the 80286.

warning L4031: <*name*> : segment declared in more than one group

The *name* segment is declared in more than one group.

warning L4032: <*name*> : segment defined both 16-, 32-bit, assuming 32

The *name* segment is defined both as a 16-bit and 32-bit segment and is marked as a 32-bit segment in the segment table.

warning L4050: too many public symbols

The maximum limit of 3072 public symbols has been defined, causing the -m option to not sort the symbols in the map file.

warning L4060: code group longer than 65530

A group containing 16-bit code segments, with a total length of 65,501 to 65,536 bytes, is unreliable on the 80286.

warning L4061: multiple code segments--should be medium model

The program defines more than one code segment and the **-Mm**, **-Ml**, **-Mh**, or **-Me** option was not given. Verify that all modules have the same memory model, or link with the **-Me** option.

warning L4062: multiple data segments--should be large model

The program defines more than one data segment and the **-Ml**, **-Mh**, or **-Me** option was not given. Verify that all modules have the same memory model, or link with the **-Me** option.

warning L4063: stack option ignored for 80386 executable

An **-F** option was given while linking an 80386 library; the linker ignored the **-F** option.

warning L4064: page-alignment option ignored for 80286 executable

An **-N** option was given while linking an 80286 library; the linker ignored the **-N** option.

Appendix C

sdb: The Symbolic Debugger

C.1 Introduction

This chapter describes a symbolic debugger, **sdb**, as implemented for C and assembly language programs on the XENIX operating system.

You can use the **sdb** program both for examining core images of aborted programs and for providing an environment in which execution of a program can be monitored and controlled. The **sdb** program allows you to interact with a debugged program at the source language level. It carefully controls the execution of a program while letting you examine and modify the program's data and text areas.

This chapter explains how to use **sdb**. In particular, it explains how to:

- Start and stop the debugger
- Display and manipulate instructions and data in source files
- Control and monitor program execution
- Debug machine language programs

A tutorial is provided at the end of this chapter that shows you how to work your way through your program using **sdb**.

C.2 Using sdb

The **sdb** program provides a comprehensive set of commands to let you examine, debug, and repair source files. To use all of the **sdb** features, it is necessary to compile the source program that will be debugged with the **-Zi** option using a command of the form:

```
cc -Zi filename.c -o filename
```

You must also link the executable object file with the **-I** option using the command:

```
ld -I objectfiles -o filename
```

This causes the compiler to generate additional information about the variables and statements of the compiled program. When the **-Zi** option has been specified, **sdb** can be used to obtain a trace of the called functions at the time of the abort and interactively display the values of variables.

There are two basic ways to use **sdb**: by running your program file under control of **sdb**, or by using **sdb** to examine the core image file left by a program that failed. The first way lets you run the program with **sdb** to see what is happening up to the point at which the program fails (or you can skip past the failure point and proceed with the run). The second method lets you check the status of the core file at the moment the program fails, which may or may not disclose the reason it failed. Both of these methods are discussed in detail in the following sections.

C.2.1 Starting sdb With a Program File

You can debug any executable C or assembly language program file by typing a command line of the form:

```
        sdb [filename]
```

where *filename* is the name of the program file to be debugged. **sdb** opens the file and prepares its text (instructions) and data for subsequent debugging. For example, the command:

```
        sdb sample
```

prepares the program named "sample" for examination and execution.

Once started, **sdb** prompts with an asterisk (*) and waits for you to type commands. If you have given the name of a file that does not exist or is in the wrong format, **sdb** will display an error message first, then wait for commands.

You may also start **sdb** without a filename. In this case, **sdb** searches for the default file, *a.out*, in your current working directory and prepares it for debugging. Thus, the command:

```
        sdb
```

is the same as typing:

```
        sdba.out
```

sdb displays an error message and waits for a command if the *a.out* file does not exist.

C.2.2 Starting sdb With a Core Image

The **sdb** program lets you examine the core image files of programs that caused fatal system errors. When debugging a core image from an aborted program, **sdb** reports which line in the source program caused the error and allows all variables to be accessed symbolically and to be displayed in the correct format.

To illustrate the process of debugging a core image file, we will use a hypothetical file called *prgm.c* and show a typical set of commands and responses for this process. First you must compile and execute the program by typing the command:

```
        cc -Zi prgm.c -o prgm
```

Execute the program by typing the command line:

```
prgm
```

In this example, we will assume that an error occurred while executing the program, causing a core dump. The output resulting from this error is:

```
Bus error - core dumped
```

Now invoke the **sdb** program and examine the core dump to determine the causes of the error using the command:

```
sdb prgm
```

A possible response from the **sdb** program is:

```
main:25: x[i]=0;
*
```

This output means that the bus error occurred in function **main** at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. The **sdb** program then prompts the user with an *, which shows that it is waiting for a command.

It is useful to know that **sdb** uses a notion of current function and current line. In this example, they are initially set to **main** and 25, respectively.

In the example shown in this section, **sdb** was called with one argument, **prgm**. In general, it takes three arguments on the command line:

1. The first is the name of the executable file that is to be debugged; it defaults to **a.out** when not specified.
2. The second is the name of the core file, defaulting to **core**. You can prevent **sdb** from opening this file by using the hyphen (-) in place of the core filename.
3. The third argument is the list of the directories (separated by colons) containing the source of the program being debugged. The default is the current working directory.

In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

If the error occurred in a function that was not compiled with the **−Zi** option, **sdb** prints the function name and the address at which the error occurred. The current line and function are set to the first executable line in **main**. If **main** was not compiled with the **−Zi** option, **sdb** will print an error message, but debugging can continue for those routines that were compiled with the **−Zi** option.

A sample **sdb** session with more examples is shown at the end of this chapter.

C.2.3 Printing a Stack Trace

When debugging a program, it is often useful to obtain a listing of the function calls that led to the error. You may obtain this listing by typing the **t** command in response to the **sdb** prompt:

```
*t
```

Possible output from the **t** command might be:

```
sub(x=2,y=3)    [prgm.c:25]
inter(i=16012)  [prgm.c:96]
main(argc=1,argv=0x7ffff54,envp=0x7ffff5c) [prgm.c:15]
```

This indicates that the program was stopped within the function **sub** at line 25 in file *prgm.c*. The **sub** function was called with the arguments **x**=2 and **y**=3 from **inter** at line 96. The **inter** function was called from **main** at line 15. The **main** function is always called by a startup routine with three arguments often referred to as *argc, argv*, and *envp*. Note that *argv* and *envp* are pointers, so their values are printed in hexadecimal.

C.2.4 Examining Variables

The **sdb** program can be used to display variables in the stopped program. Variables are displayed by typing their name followed by a slash. To display the value of variable *errflag*, type the following command:

```
*errflag/
```

Unless otherwise specified, variables are assumed to be either local to or accessible from the current function. To specify a different function, give the name of the function and the name of the variable you would like to list separated with a colon character(:). For example, to display variable *i* in function **sub**, type the following command:

```
*sub:i/
```

FORTRAN users can specify a common block variable in the same manner, provided it is on the call stack.

C.2.5 Matching Patterns for Variables and Functions

The **sdb** program supports a limited method of pattern matching for variable and function names. The symbol * is used to match any sequence of characters of a variable name and ? to match any single character.

To print the values of all variables beginning with the letter **x**, type the following command:

```
*x*/
```

To print the values of all two letter variables in function **sub** beginning with the letter **y**, type the following command:

```
*sub:y?/
```

To print all variables in the current function, type the following command:

```
**/
```

In the first and last examples, only variables accessible from the current function are printed. To display the variables for each function on the call stack, type the following command:

```
**:*/
```

C.2.6 Specifying Variable Formats

The **sdb** program normally displays the variable in a format determined by its type as declared in the source program. To request a different format, a specifier is placed after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are:

b	One byte
h	Two bytes (half word)
l	Four bytes (long word)

The length specifiers are effective only with the formats **d**, **o**, **x**, and **u**. If no length is specified, the word length of the host machine is used. A number can be used with the **s** or **a** formats to control the number of characters printed. The **s** and **a** formats normally print characters until either a null is reached or 128 characters have been printed. The number specifies exactly how many characters should be printed. The available format specifiers are described below:

c	character
d	decimal
u	decimal unsigned
o	octal
x	hexadecimal
f	32-bit single-precision floating point
g	64-bit double-precision floating point

s	Assume variable is a string pointer and print characters starting at the address pointed to by the variable until a null is reached.
a	Print characters starting at the variable's address until a null is reached.
p	Pointer to function.
i	Interpret as a machine-language instruction.

For example, to display the hexadecimal value of the variable *flag*, type the following command:

```
*flag/x
```

The **sdb** program also knows about structures, arrays, and pointers so that all of the following commands work:

```
*array[2][3]/
*sym.id/
*psym->usage/
*xsym[20].p->usage/
```

The only restriction in the above cases is that array subscripts must be numbers. There are also other special cases. For example, to display the structure pointed to by **psym**, type the following command:

```
*psym[0]
```

The result of this command will be in decimal format.

C.2.7 Displaying and Forming Addresses

Core locations can also be displayed by specifying their absolute addresses. To display location 1024 in decimal, type the command:

```
*1024/
```

As in C language, numbers may also be specified in octal or hexadecimal so the above command is equivalent to both the following commands:

```
*02000/
```

and

```
*0x400/
```

It is possible to mix numbers and variables. To refer to an element of a structure starting at address 1000, type the following command:

```
*1000.x/
```

To refer to an element of a structure whose address is at 1000, type the following command:

```
*1000->x/
```

For commands involving data structures, the **sdb** program uses the structure template of the last structure referenced.

To display the address of the variable *i*, type the following command:

```
*i=
```

To redisplay the last variable typed, use the following command:

```
*./
```

C.2.8 Leaving sdb

To exit **sdb** and return to the system shell, use the **q** or **quit** command.

C.3 Displaying and Manipulating Source Files

The **sdb** program has been designed to make it easy to debug a program without constant reference to a current source listing. Features are provided that perform context searches within the source files of the program being debugged and that display selected portions of the source files. The commands are similar to those of the UNIX system text editor **ed**(C). Like the editor, **sdb** uses a notion of current file and line within the current file. **sdb** also knows how the lines of a file are partitioned into functions, so it also uses a notion of current function. As noted in other parts of this document, the current function is used by a number of **sdb** commands.

C.3.1 Displaying the Source File

Four commands exist for displaying lines in the source file. They are useful for perusing the source program and for determining the context of the current line. The commands are:

p	Displays the current line.
w	Displays a window of ten lines around the current line.
z	Displays ten lines starting at the current line. Advances the current line by ten.
Ctrl-D	Scrolls ten lines down. Displays the next ten lines and advances the current line by ten. This command is used to cleanly display long segments of the program.

When a line from a file is displayed, it is preceded by its line number. This not only gives an indication of its relative position in the file, but it is also used as input by some **sdb** commands.

C.3.2 Setting the Current File or Function

You can use the **e** command to change the current source file. For example, you can change the current file to *file.c* by typing the following command:

```
*efile.c
```

In the above example, the current line is also set to the first line in the specified file.

You can also specify that you want a file containing a certain function to become the current file. For example, to change the current file to the file containing *function*, type the following command:

```
*e function
```

The above command also causes the first line of the function specified to become the current line.

To display the current function and file, use the **e** command with no arguments.

C.3.3 Setting the Current Line

There are several ways to change the current line in the source file. For many of the commands, **sdb** behaves the same as the line editor **ed**(C). It may be helpful for you to refer to the documentation on **ed** if you are unfamiliar with the concept of current line.

The + and − commands may be used to move the current line forward or backward by a specified number of lines. Typing advances the current line by one, and typing a number causes that line to become the current line in the file. For example, you can advance the current line by 15 and then print ten lines using the following command line:

```
*+15z
```

When you use the **z** or **Ctrl-D** commands to display data, they also set the current line to the last line displayed.

C.3.4 Searching for Regular Expressions

There are two commands for searching for instances of regular expressions in source files. To search forward through a file for a line containing a string that matches a regular expression, use a command of the form:

```
*/regular expression
```

To search backward through a file for a line containing a string that matches a regular expression, use a command of the form:

```
*?regular expression
```

The method used to match regular expressions in **sdb** is identical to that used by **ed**(C).

C.4 Controlling Program Execution

One very useful feature of **sdb** is breakpoint debugging. After entering **sdb**, breakpoints can be set at certain lines in the source program. The program is then started with an **sdb** command. Execution of the program proceeds as normal until it is about to execute one of the lines at which a breakpoint has been set. The program stops and **sdb** reports the breakpoint where the program stopped. At this point, **sdb** commands may be used to display the trace of function calls and the values of variables. If the user is satisfied the program is working correctly to this point, some breakpoints can be deleted and others set; and program execution may be continued from the point where it stopped.

C.4.1 Setting and Deleting Breakpoints

You can set breakpoints at any line in any function if your program has been compiled correctly. You will use the **b** command to set breakpoints. To set a breakpoint at line 12 in the current file, type the following command:

```
*12b
```

The line numbers are relative to the beginning of the file as printed by the source file display commands: **p**, **w**, **z**, and **Ctrl-D**.

To set a breakpoint at line 12 of function **proc**, type the following command:

```
*proc:12b
```

To set a breakpoint at the first line of function **proc**, type the following command:

```
*proc:b
```

To set a breakpoint at the current line, type the following command:

```
*b
```

You can delete breakpoints in the same way that you set them using the **d** command. To delete a breakpoint at line 12 in the current file, type the following command:

```
*12d
```

To delete a breakpoint at line 12 of function **proc**, type the following command:

```
*proc:12b
```

To delete a breakpoint at the first line of function **proc**, type the following command:

```
*proc:b
```

To delete breakpoints interactively, type the **d** command with no arguments. **sdb** prints the location of each breakpoint and waits for a response from the user. If you respond with a **y** or **d**, the breakpoint is deleted.

To print a list of all the current breakpoints, use the **B** command. To delete all the current breakpoints, use the **D** command.

You can also use the breakpoint command to automatically perform a sequence of commands at a breakpoint and then have execution continue. For example, if you want both a trace back and the value of 0 to be displayed each time execution gets to line 12, type the following command:

```
*12b t;x/
```

The **a** command also allows **sdb** to perform debugging functions and then automatically let the program continue execution. If you want a function name and its arguments to be printed each time it is called, type the following command:

**proc*:a

If you want a certain line in the source code to be printed each time it is about to be executed, type the following command:

**proc*:12a

When using the a command, execution continues after the function name or source line is printed.

C.4.2 Single Stepping Through a Program

A useful alternative to setting breakpoints is single stepping through a program. Sdb can be requested to execute the next line of the program and then stop using the s command. The command has the form:

[*count*]s

where *count* is the number of lines to execute in each step. This command is useful for slowly executing the program to examine its behavior in detail.

The S command is similar to the s command but it steps through the program function by function, rather than line by line. It is often used when one is confident that the called function works correctly but is interested in testing the calling routine.

Single stepping is especially useful for testing new programs, so they can be verified on a statement-by-statement basis. If an attempt is made to single step through a function that has not been compiled with the **-Zi** option, and linked with **-I** option, execution proceeds until a function is reached that has been compiled and linked in this way.

You can use the **i** command to run the program one machine level instruction at a time while ignoring the signal that stopped the program. Its uses are similar to the s command.

The **L** command loads the program to be debugged but does not run it. If you wish to examine the initial values of memory locations before the program has started to run, or to disassemble portions of the program without actually running it, you must enter the **L** command before you begin.

There is also an **I** command that causes the program to execute one machine level instruction at a time, but also passes the signal that stopped the program back to the program. These machine level commands are particularly useful when you have not compiled and linked your program with the necessary options.

Refer to the XENIX *Reference* for more information on the single stepping commands.

C.4.3 Running the Program

The r command is used to begin program execution. This command allows you to restart the program as if it were invoked from the shell. For example, to run the current program with given arguments, use a command of the form:

```
*r args
```

If no arguments are specified, then the arguments from the last execution of the program within **sdb** are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as INTERRUPT or QUIT occurs, or the program terminates. INTERRUPT and QUIT are represented as DEL and **Ctrl-D** in . In all cases after an appropriate message is printed, control returns to the user.

To continue execution of a stopped program, use the **c** command. You can also use the **c** command to insert a temporary breakpoint during execution. For example, to place a temporary breakpoint at line 12 and resume execution of a stopped program, type the following command:

```
*proc:12c
```

The temporary breakpoint is deleted when the **c** command finishes.

If you want the signal that stopped program execution to be passed back to the program, use the **C** command. This command is useful for testing user-written signal handlers.

Execution may be continued at a specified line with the g command. For example, you may continue program execution at line 17 of the current function with the following command:

```
*17g
```

This command is useful when you want to avoid a section of code that you already know is bad. You should not attempt to continue execution in a function different than that of the breakpoint.

C.4.4 Calling Functions and Procedures

It is possible to call any of the functions or procedures in the program from **sdb**. This feature is useful both for testing individual functions with different arguments and for calling a user-supplied function to display structured data. To simply execute a function or procedure, use a command of the form:

```
*proc(arg1, arg2, ...)
```

To call a function and display the value that it returns, use a command of the form:

```
*proc(arg1, arg2, . . .)/m
```

The value is displayed in decimal unless some other format is specified by m. Arguments to functions may be integer, character or string constants, or variables that are accessible from the current function.

If a function is called when the program is not stopped at a breakpoint (such as when a core image is being debugged), all variables are initialized before the function is started. This makes it impossible to use a function that formats data from a dump.

C.5 Debugging Machine Language Programs

The **sdb** program has facilities for examining programs at the machine language level. It is possible to print the machine language statements associated with a line in the source and to place breakpoints at arbitrary addresses. The **sdb** program can also be used to display or modify the contents of the machine registers.

C.5.1 Displaying Machine Language Statements

To display the machine language statements associated with line 25 in function **main**, use the command

```
*main:25?
```

The **?** command is identical to the **/** command except that it displays from text space. The default format for printing text space is the **i** format, which interprets the machine language instruction. The **Ctrl-D** command may be used to print the next ten instructions.

Absolute addresses may be specified instead of line numbers by appending a colon (:) to them. For example, to display the contents of address 0x1024 in text space, use the following command:

```
*0x1024:?
```

To display the instruction corresponding to line 0x1024 in the current function, use the following command:

```
*0x1024?
```

It is also possible to set or delete a breakpoint by specifying its absolute address. To set a breakpoint at address 0x1024, use the following command:

```
*0x1024:b
```

C.5.2 Manipulating Registers

The **x** command displays the values of all the registers. Also, individual registers may be named instead of variables by prefixing the register name with the character "@". For example, the command:

```
*@bx
```

displays the value of the register *bx*.

C.6 Using XENIX Shell Commands

The ! command (when used immediately after the * prompt) performs the same function that it does in **ed**(C). It allows you to temporarily escape from **sdb** to execute a shell command. When the command has finished executing, control is returned to the **sdb** program.

The ! can also be used to change the values of variables or registers when the program is stopped at a breakpoint. To set a specified variable to a specified value, use the following command:

**variable!value*

To set a named register **ax** to a specified value, use the following command:

*@ax!*value*

The value may be a number, character constant, register, or the name of another variable. If the variable is of type **float** or **double**, the value can also be a floating-point constant (specified according to the standard C language format).

C.7 A Sample sdb Session

This section provides a sample **sdb** session to show you how to to debug a program using **sdb**. The filenames and file contents used in this session are hypothetical and are not intended to represent the real world.

During this session, we are going to debug an executable object file called *myoptim*. The *myoptim* file has been compiled using the **-Zi** option and linked using the **-I** option so that we can use the full capabilities of the **sdb** commands during this session. We will examine the source file itself, rather than its core image resulting from program failure. To begin the **sdb** session, enter the following command:

```
sdb myoptim - .:../common
```

Please note that we do not want **sdb** to examine the core file associated with the *myoptim* file so a hyphen (-) has been specified here to prevent this. The **sdb** program will display the following messages:

```
Source path: .:../common
core file ignored
```

It is useful to set breakpoints anywhere in your program that you suspect might be the trouble spot. For example, set a breakpoint at the start of function **window** by typing the command line:

```
*window:b
```

The **sdb** program displays the following message:

```
0x3F042C (window:1459+2) b
```

Start executing the program by typing the command line:

```
*r < m.s > out.m.s
```

This redirects standard input to *m.s* and sends standard output to *out.m.s*. The **sdb** program displays the following message:

```
Breakpoint at
0x3F042C in window:1459: window(size, func) register int size;
boolean(*func)(); {
```

When debugging a program, it is often useful to obtain a listing of the function calls that led to the error. To do this, type the following command:

```
*t
```

The **sdb** program displays the following messages:

```
window(size= ,func=w opt) [optim.c:1459]
peep() [peep.c:34]
yylex() [local.c:229]
main(argc=0,argv=0x47BF86,-1073610300) [optim.c:227]
```

It is useful to peruse the source program to determine the context of the current line. To do this, display the next ten lines of the source file using the following command:

```
*z
```

The **sdb** program displays the following:

```
1459: window(size, func) register int size; boolean (*func)(); {
1460:
1461: extern NODE *initw();
146 : register NODE *pl;
1463: register int i;
1464:
1465: TRACE(window);
1466:
1467: /* find first window */
1468:
```

It is often useful to step through a program line by line while it is executing to examine its behavior in detail. To do this, type the following command:

```
*s
```

The **sdb** program displays the following: window:1459: window(size, func) register int size; boolean (*func)(); { Type the single step command again:

```
*s
```

The **sdb** program displays the following:

```
window:1465: TRACE(window);
```

Continue using this command until you have examined the lines of the program that are of interest to you:

```
*s
```

The **sdb** program displays the following:

```
window: 1469: wsize = size;
```

An important alternative to the s command is the S command. This command allows you to step through your program function by function, rather than line by line. To do this, type the following command line:

```
*S
```

The sdb program displays the following:

```
window:1475:  for(opf=pf->back; ;opf=pf->back){
```

The sdb program can be used to display the current value of variables used in the program. For example, display the value of the variable *pl* by typing the following command:

```
*pl
```

The sdb program displays the following:

```
0x476B38
```

You can use the **x** command to display the contents of all the registers in your program. To do this, type:

```
*x
```

The sdb program displays the following:

```
ax=2220 bx=0047 cx=0000 dx=0047 sp=1FDE bp=0000 si=0000 di=0000
ds=0047 es=0047 ss=0047 cs=003F ip=00FD NVUPEIPLZRNA PENC
003F:00FD 8BEC      mov    bp,sp
```

You can use the **sdb** program to dereference pointers in your program. To dereference the *pl* pointer, type the following command:

```
*pl[0]
```

The sdb program displays the following:

```
pl[0].forw/ 0x476B6C
pl[0].back/ 0x476AC8
pl[0].ops[0]/ pushw
pl[0].uniqid/ 0
pl[0].op/ 123
pl[0].nlive/ 3588
pl[0].ndead/ 4096
```

To dereference the *pl->forw* pointer, type the following command:

```
*pl->forw[0]
```

The sdb program displays the following:

```
        pl->forw[0].forw/0x476CA6
        pl->forw[0].back/0x476B38
        pl->forw[0].ops[0]/ call
        pl->forw[0].uniqid/0
        pl->forw[0].op/9
        pl->forw[0].nlive/3584
        pl->forw[0].ndead/4099
```

You can use **sdb** to change the values of variables when your program is stopped at a brea point. To replace the value of *pl* with the value of *pl->forw*, type the following command line:

```
        *pl!pl->forw
```

Display the new value of *pl*. To do this, type:

```
        *pl
```

The sdb program displays the following:

```
        0x476CA6
```

Continue executing the program until the next brea point is reached. To do this, type:

```
        *c
```

The sdb program displays the following:

```
        Breakpoint at
        0x3F042C in window:1459: window(size, func) register int size;
        boolean (*func)(); {
```

Single step through the program until you are confident you have examined the problem area. To do this, type:

```
        *s
```

The sdb program displays the following:

```
        window:1459: window(size, func) register intsize; boolean (*func)(); {
```

Continue single stepping through the program using the s command:

```
        *s
```

The sdb program displays the following:

```
        window:1465:  TRACE(window);
```

When you see a function that is of interest, you can display the size of the current function's argument. To do this, type:

```
*size
```

The **sdb** program displays the following:

```
3
```

When you have finished debugging your program, you will want to delete all the breakpoints in it. To do this, type:

```
*D
```

The **sdb** program displays the following:

```
All breakpoints deleted
```

Continue executing your program until it has finished running. To do this, type:

```
*c
```

When program execution is complete, the **sdb** program will display the following:

```
Process terminated normally (0)
```

The number that appears in parentheses is the exit status of the program being debugged. When you are ready to leave **sdb**, type the following command:

```
*q
```

The quit command ends your **sdb** session.

XENIX® System V

Development System

C Library Guide

This document was typeset with an IMAGEN® 8/300 Laser Printer.

SCO Document Number: XG-6-21-87-4.0

Contents

Chapter 1

Introduction

1.1 Introduction

The XENIX *C Library Guide* is designed to complement its companion volumes: the XENIX *C Language Reference* and the XENIX *C User's Guide*. While the *C Language Reference* serves as a reference for our implementation of the C language and the *C User's Guide* provides you with the knowledge to compile and run C programs, the *C Library Guide* provides you with information about the standard include files, tells you how to build user interfaces for C programs, provides a variety of tools to create programs that can be executed under control of the DOS operating system, and a full description of error messages.

This guide assumes that you have a basic familiarity with the C programming language and the XENIX environment.

1.2 About This Guide

This guide is organized as follows:

Chapter 1, "Introduction," summarizes the organization of this guide and the conventions used.

Chapter 2, "Run-Time Routines," contains summary descriptions of the routines available in the run-time library. See Section S of the XENIX *Reference*, for full descriptions of use and syntax.

Chapter 3, "Include Files," briefly describes each include file available and lists the routines that use it.

Chapter 4, "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 5, "Screen Processing," describes the library (*curses*) and functions which control screen updating and cursor movement.

Chapter 6, "Character and Screen Processing," describes the system-provided functions for character and string processing.

Chapter 7, "Using Process Control," describes the process control functions available with the standard C library.

Chapter 8, "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 9, "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.

Appendix A, "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.

Appendix B, "System Error Values," describes the standard error messages provided by XENIX.

1.3 Notational Conventions

This guide uses a number of notational conventions to describe the syntax of XENIX commands:

boldface	Boldface indicates a command, option, flag, or program name to be entered as shown. Boldface indicates the name of a library routine, global variable, standard type, constant, keyword, or identifier used by the C library. (To find more information on a given library routine consult the "Alphabetized List" in your XENIX *Reference* for the page that describes it.)
italics	Italics indicate a filename. This pertains to library include filenames (i.e. *stdio.h*), as well as, other filenames (i.e. */etc/ttys*). Italics indicate a placeholder for a command argument. When entering a command, a placeholder must be replaced with an appropriate filename, number, or option. Italics indicate a specific identifier, supplied for variables and functions, when mentioned in text. Italics indicate user named routines. User named routines are followed by open and close parentheses, (). Italics indicate emphasized words or phrases in text.

CAPITALS	Capitals indicate names of environment variables (i.e. TZ and PATH).
SMALL CAPITALS	Small capitals indicate keys and key sequences (i.e. RETURN).
[ ]	Brackets indicate that the enclosed item is optional. If you do not use the optional item, the program selects a default action to carry out.
...	Ellipses indicate that you can repeat the preceding item any number of times.
	Vertical ellipses indicate that a portion of a program example is omitted.
" "	Quotation marks indicate the first use of a technical term.
	Quotation marks indicate a reference to a word rather than a command.

Chapter 2

Run-Time Routines By Category

2.1 Introduction

This chapter contains a complete listing of all the routines available in the current C libraries. Routines are separated into categories to make them easier to use. Because of the number of routines, the descriptions of their functions are brief.

This chapter is provided in order to give you a summary of what is already available. Hence, arguments to the routines are not commonly given. On occasion, a number of routines are listed on one page. For help in finding a specific routine it may be necessary for you to look in the permuted index. For a complete discussion of a given routine, see the XENIX *Reference*.

2.2 Buffer Manipulation

The 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'). Thus, the buffer manipulation routines always take a length or count argument.

Function declarations for the buffer manipulation routines are given in the include file *memory.h*.

Routine	**Use**
memccpy	Copies characters from one buffer to another, until a given character is copied 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.

2.3 Character Classification and Conversion

The character classification and conversion routines let you test individual characters in a variety of ways, and convert between uppercase and lowercase characters. The classification routines identify a character by looking it up in a table of classification codes. Using these routines is generally faster than writing an equivalent test expression (such as if ((c >0) | |c <0x7f)) to classify a character.

Routine	Use
isalnum	Tests for alphanumeric character.
isalpha	Tests for alphabetic character.
isascii	Tests for ASCII character.
iscntrl	Tests for control character.
isdigit	Tests for decimal digit.
isgraph	Tests for printable character except space.
islower	Tests for lowercase character.
isprint	Tests for printable character.
ispunct	Tests for punctuation character.
isspace	Tests for whitespace character.
isupper	Tests for uppercase character.
isxdigit	Tests for hexadecimal digit.
toascii	Converts character to ASCII code.
tolower	Tests character and converts to lowercase if uppercase.
toupper	Tests character and converts to uppercase if lowercase.
_tolower	Converts character to lowercase (unconditional).
_toupper	Converts character to uppercase (unconditional).

The **tolower** and **toupper** routines are implemented both as functions and as macros; the remainder of the routines in this category are implemented only as macros. All of the macros are defined in *ctype.h*, and this file must be included or the macros will be undefined.

The **toupper** and **tolower** macros evaluate their argument twice and thus cause arguments with side effects to give incorrect results. For this reason, you may want to use the function versions of these routines instead.

The macro versions of **tolower** and **toupper** are used by default when you include *ctype.h*. To use the function versions instead, you must give #undef preprocessor directives for **tolower** and **toupper** *after* the **#include** directive for *ctype.h*, but *before* you call the routines. This procedure removes the macro definitions and causes occurrences of **tolower** and **toupper** to be treated as function calls to the **tolower** and **toupper** library functions.

If you want to use the function versions of **toupper** and **tolower** and you do not use any of the other character classification macros in your program, you can simply omit the *ctype.h* include file. In this case no macro definitions are present for **tolower** and **toupper**, so the function versions will be used.

2.4 Cursor Routines

The cursor control routines are available to the user when the cursor control library (**curses**) is specified on the compile line. For detailed information on these routines, their use, and their syntax, see Chapters 5 and 6 of this guide and the appropriate page in the S section of the *XENIX Reference*.

Routine	Use
addch	Adds a character to **stdscr**.
addstr	Adds a string to **stdscr**.
box	Draws a box around a window.
crmode	Sets **cbreak** mode.
clear	Clears **stdscr**.
clearok	Set clear flag for *win*.
clrtobot	Clears to bottom on **stdscr**.
clrtoeol	Clears to end-of-line on **stdscr**.
delch	Deletes a character from **stdscr**.
delwin	Delete *win*.
echo	Sets echo mode.
erase	Erase **stdscr**.
getch	Gets a character through **stdscr**.
getstr	Gets a string through **stdscr**.
gettmode	Gets tty modes.
getyx	Gets current (*y*,*x*) position of *win*.
inch	Gets character at current (*y*,*x*) coordinates.
***initscr**	Initializes screens.
insch	Inserts a character in **stdscr**.
insertln	Inserts a blank line in **stdscr**.
leaveok	Sets leave flag for *win*.
longname	Gets long name from *termbuf*.
move	Moves to (*y*,*x*) on stdscr.
mvaddch	Moves to (*y*,*x*) and adds character *ch*.
mvaddstr	Moves to (*y*,*x*) and adds string *str*.
mvcur	Moves cursor from (*lasty*,*lastx*) to (*newy*,*newx*).
mvdelch	Moves to (*y*,*x*) and deletes character from **stdscr**.
mvgetch	Moves to (*y*,*x*) and gets a character through **stdscr**.
mvgetstr	Moves to (*y*,*x*) and gets a string through **stdscr**.
mvinch	Moves to (*y*,*x*) and gets a character at the current coordinates.
mvinsch	Moves to (*y*,*x*) and inserts a character in **stdscr**.
mvwaddch	Moves to (*y*,*x*) in *win* and adds a character *ch*.
mvwaddstr	Moves to (*y*,*x*) in *win* and adds string *str*.
mvwdelch	Moves to (*y*,*x*) in *win* and deletes the character.

mvwgetch	Moves to (*y*,*x*) in *win* and gets a character.
mvwgetstr	Moves to (*y*,*x*) in *win* and gets a string.
mvwin	Moves upper corner of *win* to (*y*,*x*).
mvwinch	Moves to (*y*,*x*) in *win* and gets a character at current coordinates.
mvwinsch	Moves to (*y*,*x*) in *win* and inserts a character.
***newwin**	Creates a new window.
nl	Sets newline mappping.
nocrmode	Unsets cbreak mode.
noecho	Unsets echo mode.
nonl	Unsets newline mapping.
noraw	Unsets raw mode.
overlay	Overlays *win1* on *win2*.
overwrite	Overwrites *win1* on *win2*.
printw	Prints arguments on **stdscr**.
raw	Sets raw mode.
refresh	Makes current screen look like **stdscr**.
resetty	Resets tty flags to stored value.
savetty	Stores current tty flags.
scanw	Scans for arguments through **stdscr**.
scroll	Scrolls *win* one line.
scrollok	Sets scroll flag.
setterm	Sets terminal variables for *name*.
standend	Clears standout mode of **stdscr**.
standout	Sets standout mode for characters in subsequent output to **stdscr**.
***subwin**	Creates a subwindow in *win*.
touchwin	Prepares *win* for complete update on next refresh.
unctrl	Provides a printable version of *ch*.
waddch	Adds *ch* to *win*.
waddstr	Adds *str* to *win*.
wclear	Clears *win*.
wclrtobot	Clears to bottom of *win*.
wclrtoeol	Clears to end-of-line on *win*.
wdelch	Deletes current character from *win*.
wdeleteln	Deletes line from *win*.
werase	Erases *win*.
wgetch	Gets a character through *win*.
wgetstr	Gets a string through *win*.
winch	Gets a character at current (*y*,*x*) in *win*.
winsch	Inserts character *c* in *win*.
winsertln	Inserts a blank line in *win*.
wmove	Sets current (*y*,*x*) coordinates on *win*.
wprintw	Prints arguments on *win*.
wrefresh	Makes screen look like *win*.
wscanw	Scans for arguments through *win*.
wstandend	Clears standout mode for *win*.
wstandout	Sets standout mode for characters on subsequent output to *win*.

2.5 Database Manipulation Routines

These 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 the appropriate page in the *XENIX Reference*.

Routine	Use
dbminit	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.

2.6 Directory-Operation Routines

These routines provide control over the special files called directories. For a full description of their use, see **directory**(S) in the XENIX *Reference*. Programs referencing these routines must be linked with the **-lx** option.

The following lists the directory-operation routines supported by C:

Routine	Use
closedir	Causes the named directory stream to be closed and the structure associated with the directory pointer to be freed
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.

2.7 File Handling

These routines, coupled with the reading and writing and stream control routines, provide you with a great deal of control of files and devices at a low level. These routines are automatically available to all programs. See Chapters 3, 4 and 8 of this guide for more information. For details on the syntax and use of a specific routine, see the appropriate page in the *XENIX Reference*.

Routine	Use
access	Determines the accessibility of a specified file.
chdir	Changes the current working directory to the one specified by *path*.
chmod	Changes the access permission to *mode* on the file specified by *path*.
chown	Changes the owner and group of a file specified by *path*.
chroot	Changes the root directory to the one specified by *path*.
chsize	Sets the size of the file specified by *fildes* to exactly *size* bytes.
close	Closes a file descriptor.
creat	Creates a new file or re-writes an existing one.
dup,dup2	Duplicates an open file descriptor.
fcntl	Controls open files.
fstat	Returns the status of an open file (described by the file descriptor *fildes*).
getcwd	Returns a pointer, *pnbuf*, to the pathname of the current working directory.
ioctl	Provides control of character devices.
link	Makes a new link by creating a directory entry for the existing file using the new name.
lockf	Provides semaphores and record locking on files.
locking	Locks or unlocks a region of a file for reading or writing.
mknod	Creates a directory, special file, or ordinary file with a specified *path* and *mode*.
mount	Requests that a removable file system contained on the block special file, *spec*, be mounted on the directory *dir*.
open	Opens a file for reading or writing.
pipe	Creates an inter-process pipe.
stat	Returns the status of the named file.
umask	Sets and gets the file creation mask.
umount	Unmounts a file system mounted by *mount*. See *mount* above.
unlink	Removes a directory entry specified by the pathname pointed to by *path*.
ustat	Returns information about a mounted file system.
utime	Sets file access and modification times of the file specified by *path*.

2.8 Group and Password File Control

These 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 may still conduct searches of the files. See Chapter 3 of this guide for information on the format of both the group and the password files and see the appropriate page in the S section of the *XENIX Reference* for information on a specific routine.

Routine	Use
endgrent	Closes the group file.
endpwent	Closes the password file.
getgrent	Reads the next line of the group file.
getgrgid	Searches the group file from the beginning for a match to *gid*.
getgrnam	Searches the group file from the beginning for a match to *name*.
getpass	Reads a password from */dev/tty*, or the standard input if */dev/tty* cannot be opened.
getpw	Searches the password file for the specified *uid*, and returns the matching line to the *buf*.
getpwent	Reads the next line in the password file.
getpwnam	Searches from the beginning of the password file for a matching *name*.
getpwuid	Searches from the beginning of the password file for a matching *uid*.
putpwent	Writes a line on the stream *f*. Format matches that of */etc/passwd*.
setgrent	Rewind the group file.
setpwent	Rewind the password file.

2.9 Math Routines

To use the following routines, incorporate the file *math.h* into your program. For a detailed explanation of the nature and syntax of these routines, see the appropriate page in the S section of the *XENIX Reference*.

Routine	Use
abs	Returns the absolute value of an integer *i*.
acos	Returns the arc cosine of x.
asin	Returns the arc sine of x.
atan	Returns the arc tangent of x.
atan2	Returns the arc tangent of y/x in the range of $-\pi$ to π.
cabs	Determines Euclidean distance.
ceil	Returns the smallest integer not less than x.

cos	Returns the cosine of *x*.
cosh	Returns the hyperbolic cosine of *x*.
erf	Returns the function of *x*, defined as: {2oversqr pi}Intfrom0toxesup{- t suo 2}dt
erfc	Returns he value 1.0 −**erf**(*x*) because of the inherent loss of accuracy of **erf**.
fabs	Returns the absolute value of *x*.
floor	Returns the largest integer (as a **double**) not greater than *x*.
fmod	Returns the number *f* such that $x = iy + f$, for some inte r *i*, and $0 \leq f < y$.
frexp	Returns the mantissa of a **double** *value* as a **double** quantity, *x*, of magnitude less than 1, and stores an integer *n* such that *value* =*x**2***n*, indirec ly through an integer pointer, *eptr*.
gamma	Returns $ln\|\Gamma(\|x\|)\|$.
hypot	Returns Euclidean distance.
j0,j1,jn,y0,y1,yn	Calculates Bessel functions of the first and second kinds for real arguments and integer orders. See bessel in the index.
ldexp	Returns the quan ity *value**(2***exp*).
log	Returns the natural logari hm of *x*.
matherr	Handles errors returned by the ma h outines.
modf	Returns the positive fractional part of *value* and stores the integer part indirectly through the **double** pointer, *iptr*.
pow	Returns x^y.
rand,srand	Generates a pseudo-random number. *srand* is used to provide he seed.
sin	Returns the sine of *x*.
sinh	Returns the hyperbolic sine of *x*.
sqrt	Returns the square root of *x*.
tanh	Returns the hyperbolic tangent of *x*.

2.10 Memory Allocation

The following routines provide a means by which memory can be dynamically allocated or freed. These routines may be incorporated by referencing the include file *malloc.h*. These routines are explained in more detail in the S section of the *XENIX Reference*.

Routine	**Use**
calloc	Allocates space for an array. Space is initialized to zeros.
free	Stores a pointer to a block previously allocated by *malloc*.
mallinfo	Provides instrumenta ion describing space usage.

malloc	Allocates the first contiguous reach of free space found in a circular search from the last block allocated or freed.
mallopt	Provides for control over the allocation algorithm.
realloc	Changes the size of the block pointed to, and returns a pointer to the (possibly moved) block.

The following functions provide another means by which memory can be dynamically allocated or freed. These routines are explained in more detail in the S section of the *XENIX Reference*.

Routine	**Use**
calloc	Allocates space for an array. Space is initialized to zeros.
free	Stores a pointer to a block previously allocated by *malloc.*
malloc	Allocates the first contiguous reach of free space found in a circular search from the last block allocated or freed.
realloc	Changes the size of the block pointed to, and returns a pointer to the (possibly moved) block.

2.11 Message Control Routines

The following routines provide message control functions. These messages are the medium for inter-process communication. For more information, see **msgop**(S) in the XENIX *Reference.*

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.

2.12 Numeric Conversion

The following routines are useful because they provide you with the ability to convert numeric strings from one format to another.

Routine	**Use**
a64l	Takes a pointer to a null-terminated base 64 representation and returns a corresponding **long** value.
atoi	Converts a string pointed to by a pointer, *nptr*, to an integer number.

atof	Converts a string pointed to by a pointer, *nptr*, to a floatingpointnumber.
atol	Converts a string pointed to by a pointer, *nptr*, to a **lo**ng integer number.
ecvt	Converts *value* to a null-terminated ASCII string of *ndigit* length and returns a pointer to the string.
fcvt	Identical to *ecvt* except that the output has been rounded for FORTRAN F format output of the number of digits specified by *ndigit*.
gcvt	Converts a specified *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. Attempts to produce *ndigit* significant digits in FORTRANF format.
ltol3	Converts a list of **long** integers (packed into a character string) into a list of 3-byte integers.
l3tol	Converts a list of 3-byte integers (packed into a character string) into a list of **long** integers.
l64a	Takes a **long** argument and returns a pointer to the corresponding base 64 representation.
sgetl	Returns a **long**-integer datum stored in memory with **isputl**.
sputl	Stores a **long**-integer datum in memory in a machine-independent fashion.
strtod	Returns as a double-precision floating-point number the value represented by the character string pointed to by a string.
strtol	Returns as a **long** integer the value represented by the character string pointed to by a string.

2.13 Process Control

The following routines provide you with low-level control of XENIX processes.

Routine	**Use**
alarm	Sets the calling process' alarm clock to *sec* seconds.
brk	Dynamically changes the amount of space allocated to the calling process' data segment. The current break value is set to *addr*. See **sbrk**, below.
brkctl	Allocates data in a far segment.
execl	Transforms the calling process into a new process. Used when a known file with known arguments is being called.
execle	Transforms the calling process into a different process. Used when a new environment is to be passed to the new process.
execlp	Transforms the calling process into a new process. Allows the specification of the new process file.

execv	Transforms the calling process into a new process. Used when the number of arguments is unknown in advance.
execve	Transforms the calling process into a new process. Allows the specification of a new environment and an unknown number of arguments.
execvp	Transforms the calling process into a new process. Allows the specification of the new process file and an unknown number of arguments.
exit	Terminates a process and closes all file descriptors.
fork	Creates a new process.
getpgrp	Returns the process group ID of the calling process.
getpid	Returns the process ID of the calling process.
getppid	Returns the parent process ID of the calling process.
gsignal	Raises the signal *sig*. Used with the software signal facility, **ssignal**. See **ssignal** below.
kill	Sends a signal to a process or group of processes.
lock	Locks a process into main memory.
monitor	Serves as an interface to **profil**. Arranges to record a histogram of periodically-sampled values of the program counter and a count of calls to certain functions.
nap	Suspends execution of the current process for *period* milliseconds, or until a signal is received.
nice	Decreases the CPU priority of the process by the specified *incr*.
pause	Suspends a process until a specified signal is received.
proctl	Performs a variety of functions on active processes or process groups.
profil	Creates an execution time profile of a section of core memory.
ptrace	Allows a parent process to trace the execution of a child process.
rdchk	Checks to see if a process will block if it attempts to read the file designated by *fdes*.
sbrk	Dynamically alters the amount of space allocated to the calling process' data segment. Adds *incr* bytes to break value (the address of the first location beyond the end of the data segment).
setpgrp	Sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.
signal	Provides the calling process with a facility for handling signal trapping.
sleep	Suspends execution of the calling process for *seconds* seconds.
ssignal	Provides the user with a software signal facility to specify and trap her own signals.

times	Fills the structure **times** (declared in *<times.h>*) with the CPU times used by the calling process and the system.
ulimit	Provides control over process limits.
wait	Suspends the calling process until it traps a specified signal, or a child process stops or terminates.

2.14 Random-Number Generation Routines

The following routines generate pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic:

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}]

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 *Reference*.

2.15 Reading and Writing a File

Routine	Use
read	Reads from a file.
write	Writes to a file.
lseek	Moves the read/ write pointer within a file.

2.16 Search Routines

Routine	Use
bsearch	Performs a binary search and update.
ftw	Walks a hierarchical file tree
hcreate	Allocates sufficient space for the hash table
hdestroy	Destroys the hash table
hsearch	Returns a pointer to a hash table indicating the location at which an entry can be found

lsearch	Performs a linear search and update.
tdelete	Deletes anode from a binary search tree.
tfind	Searchs a binary tree for a datum.
tsearch	Builds and accesses a binary search tree.
twalk	Traverses a binary search tree.
qsort	Performs a quick sort.

2.17 Semaphore Control

The following routines control the semaphores that signal when a resource is available or locked. For detailed information, see **semctl**(S) and other appropriate pages in Section(S) in the XENIX *Reference*.

These are the XENIX semaphor routines:

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.

These are the UNIX System V semaphor routines:

Routine	**Use**
semctl	Provides a variety of semaphore control operations.
semget	Returns the semaphore identifier associated with a key.
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. **waitsem** puts the calling process to sleep if the resource is in use.

2.18 Shared Memory Routines

The following routines provide control functions for the use of shared data segments. See **shmop**(S) in the XENIX *Reference* for details.

These are the XENIX shared memory routines:

Routine	Use
sdenter	Indicates that the current process is about to access the contents of a shared data segment.
sdfree	Detachs the current process from the shared data segment that is attached at the specified *addr*.
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 *addr*.
sdleave	Indicates that the current process is done 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 *vnum*.

These are the UNIX System V shared memory routines:

Routine	Use
ftok	Forms a key to provide to the **msgget**, **semget**, and **shmget** system calls (for inter-process communication).
shmat	Attaches the shared memory segement associated with the shared memory identifier specified by *shmid* to the data segment of the calling process.
shmctl	Provides control of various shared memory operations.
shmdt	Detaches the calling process' data segment from the shared memory segment located at a specific address.
shmget	Gets a shared memory segment associated with a key.

2.19 Stream Control Routines

The original implementation of the C language did not provide any routines for I/O. The following routines are provided to define a user interface.

Routine	Use
clearerr	Resets the error indication on the named *stream*.
fclose	Causes any buffers for the named *stream* to be emptied, and the file to be closed. Buffers allocated by standard I/O are freed.

fdopen	Associates a stream with a file descriptor obtained from **open, dup, creat, pipe.** Returns the new stream.
feof	Returns non-zero when EOF is read on the named input *stream*, otherwise zero.
ferror	Returns non-zero when an error has occurred reading and writing the named *stream*, otherwise zero. Unless cleared by **clearerr**, the error indication lasts until the stream is closed.
fflush	Causes any buffered data for the named output *stream* to be written to that file. The stream remains open.
fgetc	Performs like **getc**, but is a function, not a macro. It may be used as an argument.
fgets	Reads characters from the *stream* until a newline character is encountered, or until *n*-1 characters have been read. Returns a pointer to *s*.
fileno	Returns the integer file descriptor associated with the *stream*.
fopen	Opens the *filename* and associates a stream with it.
pclose	Closes a *stream* opened by **popen**. Returns the exit status of the command.
popen	Opens the standard input, **stdin**, for writing or the standard output, **stdout**, for reading.
fprintf	Places output on the named output, *stream*.
fputc	Performs like **putc**, but is a function rather than a macro. It may be used as an argument.
fputs	Copies the null-terminated string *s* to the named output *stream*.
fread	Reads a number of items from the named input *stream*. Returns the number of items actually read.
freopen	Substitutes the named file in place of the open *stream*. Returns the original value of the *stream*, and closes it.
fscanf	Reads from the named input *stream*.
fseek	Sets the position of the next input or output operation on the *stream*.
ftell	Returns the current value of the offset relative to the beginning of the file associated with the named *stream*.
fwrite	Writes a number of items to the named output *stream*. Returns the number of items actually written.
getc,getchar	Returns the next character from the named input *stream*. **getchar** is identical to **getc (stdin)**.
gets	Reads a string into *s* from the standard input, **stdin**. Returns a pointer to *s*.
getw	Returns the next word from the named input *stream*.

printf	Places output on the standard output, **stdout**.
putc,putchar	Appends the character *c* to the named output *stream*. Returns the character written. **putchar** is the same as **putc** (*c*, **stdout**).
puts	Copies the null-terminated string *s* to the standard output, **stdout**, and appends a newline character.
putw	Appends the word *w* to the output *stream*.
rewind	Is equivalent to **fseek** (*stream*,0L,0).
scanf	Reads from the standard input, **stdin**.
setbuf	Assigns a specific *buf* to *stream*, rather than the automatically allocated buffer.
sprintf	Places output, followed by the null character (\0), in consecutive bytes starting at *s*.
sscanf	Reads from the character string *s*.
ungetc	Pushes the character *c* back on an input *stream*. Returns *c*.
vfprintf	Same as **fprintf** except that it is called with a variable-argument list.
vprintf	Same as **printf** except that it is called with a variable-argument list.
vsprintf	Same as **sprintf** except that it is called with a variable-argument list.

2.20 String Operations

These routines allow the user to compare and manipulate strings of ASCII characters.

Routine	**Use**
strcat	Appends a copy of string *s2* to the end of *s1*.
strchr	Returns a pointer to the first occurrence of character *c* in string *s*. NULL if *c* is not found.
strcmp	Performs a lexicographic comparison between *s1* and *s2* and returns an integer less than, equal to, or greater than zero, depending on whether *s1* is lexicographically less, equal, or greater than *s2*.
strcpy	Copies *s2* to *s1*.
strcspn	Returns the initial segment of *s1* which consists entirely of characters not from *s2*.
strdup	Returns a pointer to a duplicate copy of the string pointed to by *s*.
strlen	Returns the number of non-null characters in *s*.
strncat	Appends a copy of *s2* (to length *n*) to the end of *s1*.
strncmp	Makes the same lexicographic comparison as **strcmp**, but only to *n* characters.
strncpy	Copies exactly *n* characters of string *s2* to *s1*, truncating or null-padding *s2*.

strpbrk	Returns a pointer to the first occurrence in string *s1* of any character from string *s2*. NULL if no character from *s2* exists in *s1*.
strrchr	Returns a pointer to the last occurrence of character *c* in string *s*. NULL if *c* does not occur in the string.
strspn	Returns the length of the initial segment of string *s1* which consists entirely of characters from string *s2*.
strtok	Considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The string *s1* is recursively searched and on each successive search, a pointer to the first character of the next token encountered is returned.

2.21 System Accounting Control

These routines are typically used by the system administrator to check/manipulate the contents of the system accounting files.

Routine	**Use**
acct	Enables or disables system accounting.
cuserid	Returns a pointer to a string which 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 */etc/utmp*.
getutid	Searches from the current file position forward, until it encounters an entry of the specified *id*.
getutline	Searches forward from the current file position until an entry of the specified *line* is encountered.
pututline	Writes an entry (in the *utmp* 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 */etc/utmp*.
ttyslot	Returns the index of the current user's entry in the */etc/utmp* file.

2.22 Terminal Control Routines

Routine	**Use**
tgetent	Extracts the entry for terminal *name*, a buffer pointed to by *bp*.

tgetflag	Returns 1 if the specified capability, *id*, is present in the terminal's entry in the */etc/termcap* file, 0 if it is not.
tgetnum	Returns the numeric value of the specified capability, *id*, returning (-1) if it is not given for the terminal's entry in the */etc/termcap* file.
tgetstr	Gets the string value of the specified capability, *id*, placing it in a buffer.
tgoto	Returns a cursor-addressing string.
tputs	Decodes the leading padding information of the string *cp*.

2.23 Time Control Routines

These routines provide the system administrator with control over the system time.

Routine	**Use**
asctime	Converts the times returned by **localtime** () and **gmtime** () to a 26-character ASCII string, and returns a pointer to the string.
ctime	Converts a time pointed to by *clock* into ASCII, and returns a pointer to the 26-character string.
ftime	Returns the time in a structure (**timeb** found in *<sys/timeb.h>*).
gmtime	Converts time, pointed to by *clock*, to Greenwich Mean Time (the XENIX system time).
localtime	Converts time, pointed to by *clock*, to the correct local time and adjust for possible daylight savings.
stime	Sets the system's time.
time	Returns the current system time in seconds since 00:00:00 GMT, January 1, 1970.
tzset	Sets the external variables *timezone* and *daylight* from the environment variable TZ.

2.24 Miscellaneous Routines

Routine	**Use**
abort	Generates an I/O trap signal which causes the calling process to abort.
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.

dial,undial	**dial** allocates a terminal device for reading and writing, and returns a file descriptor to the device. **undial** is called when communication is ended to deallocate the semaphore set during the allocation of the device.
defopen	Opens the default file specified by *filename*. Calling **defopen** () with NULL closes the default file.
defread	Reads the previously-opened file from the beginning until it encounters a line beginning with *pattern*. It then returns a pointer to the first character in the line following *pattern*.
flist	Performs the same function as **xlist**, below, except that **flist** accepts a pointer to a previously-opened file instead of *filename*.
getenv	Searches the environment list for a string *name* and returns the associated *value*.
getopt	Returns the next option letter in *argv* that matches a letter in *optstring*, where *optstring* is a string of recognized option letters.
logname	Returns a pointer to the null-terminated login name (determined from the environment variable).
longjmp	Restores the environment saved by the last call of **setjmp**. See **setjmp** below.
mktemp	Makes a unique filename from the specified *template*.
nlist	Examines the executable output file, *filename*, and extracts a list of values which is matched to a specified name list. Matches to name cause *type* and *value* to be inserted into the next two fields in the outputfile.
perror	Produces a short message on the standard error, **stderr**, describing the last error encountered during a system call from a C program.
regex	Executes a compiled regular expression against a *subject* string.
putenv	Changes or adds the value of an environment varaible.
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 to *gid*.
setjmp	Performs a non-local **goto.** Saves its stack environment in *env*. Returns value zero.
setuid	Sets the real and effective user IDS of the callingprocess to *uid*.
shutdn	Flushes all information in 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.
system	Passes the *string* to a new invocation of a shell.
tmpfile	Creates a temporary file and returns a corresponding FILE pointer.
tmpname	Generates a unique filename for a temporary file.
ttyname,isatty	Returns a pointer to the null-terminated pathname of the terminal device associated with the file descriptor *fildes*. **isatty** () returns one if *fildes* is associated with a terminal device, zero otherwise.
uname	Returns a null-terminated character string naming the current XENIX system.
xlist	Functions identically to **nlist**, with the additional feature of inserting a segment value (if it exists) in the executable output file. See **nlist**.

Chapter 3

Include Files

3.1 Overview

The include files in the XENIX system are divided into three groups - those which reference system information (kept in */usr/include/sys*), those which may be useful to individual users (kept in */usr/include*) and those useful in development for a DOS environment, (kept in */usr/include/dos*). Note that this separation is not absolute and you may find yourself using a number of the include files in any of the directories.

This chapter contains summary descriptions of all the XENIX include files. Following the system implementation, these descriptions are divided into three sections.

3.2 /usr/include Files

The following sections contain descriptions of the function of each include file and a list of 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.

Where these declarations or definitions are of special interest, they will be noted. For detailed information on a particular routine, see the appropriate page in the (S), (F) or (DOS) section of the XENIX *Reference*.

3.2.1 a.out.h

The file *a.out.h* determines the structure of the object file.

3.2.2 ar.h

Each file placed in an archive file is preceded by a header (which is determined by the structure **ar_hdr**). *ar.h* also sets the value of the archive magic number.

3.2.3 assert.h

Defines a macro which is useful in verifying the validity of a specified C statement. See **assert** in the XENIX *Reference* for more information.

3.2.4 core.h

Defines the location and size of a core-image file. See **core** in the XENIX *Reference* for detailed information on the structure of core files.

3.2.5 ctype.h

Defines a number of macros which classify ASCII-coded integer values by doing a table lookup. For a complete list of the available macros, see **ctype** in the XENIX *Reference*.

3.2.6 curses.h

Provides a number of routines which control screen and cursor functions. See **curses** in the XENIX *Reference* for a complete list of all the functions available.

3.2.7 dbm.h

Defines the functions:

dbminit	**fetch**	**store**
delete	**firstkey**	**nextkey**

for database manipulation. These routines are used for handling very large (billion block) databases. See the *dbm*() manual page in the XENIX *Reference* for more detailed information.

3.2.8 dial.h

Defines the routines **dial** and *undial*() which are used in communication over phone lines between XENIX and UNIX systems. It also defines the structure CALL, which stores information about the communication attempt.

3.2.9 dumprestor.h

When incremental dumps are done onto magnetic tape, the files that are being dumped are preceded by information defined by the structure **spcl**. This structure defines the format of the header record and the first record of each description.

The structure **idates** describes an entry to the file where the dump history is kept.

3.2.10 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 Appendix B of the XENIX *C Library Guide*.

See the **perror** system call in the XENIX *Reference* for information on error handling.

3.2.11 execargs.h

Provides information for the shell. Not for use by the user.

3.2.12 fcntl.h

Provides the values for the file control function, **fcntl**. For a description of the values, see **fcntl** in the XENIX *Reference*.

3.2.13 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.

3.2.14 grp.h

Defines a structure **group**, which returns pointers to information about entries in the file */etc/group*. See the **getgrent** system call in the XENIX *Reference* for more information.

3.2.15 lockcmn.h

Common lock type definitions. Included by locking.h and by fcntl.h.

3.2.16 macros.h

Defines a number of useful macros (some for string handling, others for library routines).

3.2.17 malloc.h

Defines the **mallinfo** structure (which contains information on memory allocation). Defines the routines:

malloc	**free**	**realloc**
mallopt	**mallinfo**	

See **malloc** in the XENIX *Reference* for more information.

3.2.18 math.h

Defines the math routines listed below:

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.

See **bessel**, **exp**, **floor**, **hypot**, **gamma**, **sinh**, and **trig** in the XENIX *Reference* for detailed information on the math functions.

For information on **matherr** return values, see Appendix B of the XENIX *C Library Guide*.

3.2.19 memory.h

Defines the routines:

memccpy	**memchr**	**memcmp**
memcpy	**memset**	**movedata**

that are used for buffer manipulation.

3.2.20 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. See **mount** in the XENIX *Reference* for more information.

3.2.21 mon.h

Defines two structures: **monhdr** and **mon**. 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** and **profil** in the XENIX *Reference*.

3.2.22 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. See **getpwent** in the XENIX *Reference* for details on the structure of the entries.

3.2.23 regexp.h

Defines the functions:

compile	**step**	**advance**
getrnge	**ecmp**	

that compile regular C language expressions and return pointers to the compiled forms. For a detailed description, see **regexp** in the XENIX *Reference*.

3.2.24 sd.h

Defines the shared data table and the following shared data flags:

sdenter
sdget
sdleave
sdfree
sdgetv
sdwaitv

For more information, see section (S) in the XENIX *Reference*.

3.2.25 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.

3.2.26 setjmp.h

Provides data to ensure that the **setjmp** and **longjmp** system calls are machine-independent.

3.2.27 sgtty.h

Defines the structure **sgttyb** for the **stty** and *gtty*() system calls. It also defines the **stty** and *gtty*() system calls, terminal modes, delay algorithms, speeds, and **ioctl** arguments. Additionally, it defines the structure tchars, for dealing with special characters. For more information, see **ioctl**, **tty**, and **stty** in the XENIX *Reference*. This is included for compatibility with Version 7.

3.2.28 signal.h

Defines the values that may 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** in the XENIX *Reference*.

3.2.29 stand.h

Provides the necessary information and structures for the operation of the system in standalone mode.

3.2.30 stdio.h

Defines the standard buffered input and output routines. The files **stdin**, **stdout**, and **stderr** are defined. The routines:

getchar **putchar** **ftell**
rewind **setbuf**

are defined. Macros are defined for **clearerr**, **feof**, **ferror**, and **fileno**.

For details on how to use the standard I/O routines, see the following manual pages in the XENIX *Reference*: **open**, **close**, **read**, **write**, **ctermid**, **cuserid**, **fclose**, **ferror**, **popen**, **printf**, **putc**, **puts**, **scanf**, **system**, **tmpnam**.

3.2.31 string.h

Defines the following string manipulation routines:

strcmp **strncmp** **strlen**
strspn **strcspn**

For details, see **string** in the XENIX *Reference*.

3.2.32 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** and **tty** in the XENIX *Reference*.

3.2.33 time.h

Defines the structure for the conversion of time to ASCII. Defines the routine **tzset** and the variables **timezone**, **daylight**, and **tzname**. See **ctime**(S) in the XENIX *Reference* for details.

3.2.34 unistd.h

Defines the flag values for the **lock** system call. See **lock** in the XENIX *Reference* for details.

3.2.35 ustat.h

Defines the structure **ustat** which returns information about a given mounted file system. For details, see **ustat** in the XENIX *Reference*.

3.2.36 utmp.h

Defines the format for the */etc/utmp* system accounting file. See **utmp** in the XENIX *Reference* for details.

3.2.37 values.h

Defines various values for machine-dependent variables.

3.2.38 varargs.h

Contains macros for use in variable argument functions. Provided in order to allow portability of C language code.

3.3 /usr/include/sys Files

The following include files are system files. Many of them define system parameters, or contain information used by the kernel.

3.3.1 a.out.h

Declares the structures:

xexec	**xext**	**xseg**	**xiter**
xlist	**aexec**	**nlist**	**bexec**

which define (respectively): the **x.out** header, the **x.out** header extension, the **x.out** segment table entry, the **x.out** iteration record, the structure for the **xlist** system call, the **a.out** header, and the **b.out** header.

See Chapter 7 of the XENIX *C User's Guide*, and the **a.out** manual page in the XENIX *Reference* for more detailed information.

3.3.2 acct.h

Defines the structure **acct** for use in system accounting.

See **acct** and **accton** in the XENIX *Reference* for more information.

3.3.3 ascii.h

Definitions of ascii standard values and names.

3.3.4 assert.h

Defines the ASSERT macro. See **assert** in the XENIX *Reference* for more details.

3.3.5 brk.h

Defines the commands for break control.

3.3.6 buf.h

Defines the structures **buf** and **hbuf** that (respectively) provide access to an I/O buffer for device drivers, and fast access to the buffers through hashing.

3.3.7 callo.h

Defines the structure **callo**, which is provided to allow a clock interrupt for a specific period.

3.3.8 comcrt.h

Definitions used by crt driver and by stty.

3.3.9 conf.h

Defines the structures **linesw**, **bdevsw**, and **cdevsw**, which are the declarations (respectively) for the line discipline switch, the block device switch, and the character device switch.

3.3.10 console.h

User level include file for PC Console keyboard related defines and variables.

3.3.11 crtctl.h

Defines the cursor control codes.

3.3.12 dio.h

Header file for standard badtrack scheme.

3.3.13 dir.h

Defines the structure **direct**, which contains the value for the maximum directory size.

3.3.14 errno.h

Contains the values for the **errno** variable. The kernel sets the **errno** variable upon an error. Math routines also set it.

See **perror** in the XENIX *Reference* for more information.

3.3.15 fblk.h

Defines the structure **fblk**, which contains the address of the next free block.

3.3.16 fcntl.h

Controls open files and file locking.

3.3.17 file.h

Defines a structure **file**, which holds the read/write pointer associated with each open file.

3.3.18 filsys.h

Defines the structure of the super-block and a number of fundamental system variables.

3.3.19 ino.h

Defines the structure of the inode as it appears on a disk block.

3.3.20 inode.h

Contains definitions of the structures **iisem**, **iisd**, and **inode**, which (respectively) provide information about the semaphores related to a given inode, provide information about the shared data segments allocated to the inode, and provide information about the inode itself.

3.3.21 iobuf.h

Defines the structure of the I/O buffer for each block device.

3.3.22 ioctl.h

Defines macros for I/O control.

3.3.23 ipc.h

Provides constant definitions for the inter-process communications (IPC) report. See **ipc(S)** in the XENIX *Reference* for more information.

3.3.24 kmon.h

The monitor buffer starts with the structure in this file.

3.3.25 lock.h

Defines flag values for the locking of resources.

3.3.26 lockcmn.h

Common lock type definitions. Included by locking.h and by fcntl.h.

3.3.27 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.

3.3.28 machdep.h

Defines machine-dependent variables (e.g., number of descriptor table entries, clock timing).

3.3.29 map.h

Defines the structure **map**, to hold the location of the swapmap.

3.3.30 mmu.h

Defines constants for the descriptor tables for memory management purposes.

3.3.31 mount.h

Defines the structure **mount**. One is allocated for every device mounted. See **mount** in the XENIX *Reference* for more information.

3.3.32 msg.h

Defines the structures **msqid_ds**, **msg**, **msgbuf**, and **msqinfo**, which provide (respectively) the data structure for inter-process messages, a structure for each message in the queue, the user message buffer template for the **msgsnd** and **msgrecv** system calls, and a structure containing information about the state of the message queue.

See **ipcs**, **msgctl**, **msgget**, and **msgop**, in the XENIX *Reference*, for more information on inter-process communication.

3.3.33 ndir.h

This file allows programs that use a BSD directory structure to run on XENIX.

3.3.34 ndp.h

Defines structures for the Numeric Data Processor.

3.3.35 nfs.h

This file provides necessary information to install networking software.

3.3.36 param.h

Contains a number of parameters vital to the system: the system's adjustable parameters, priorities, signals, MMU constants, macros for unit conversion, and definitions of the fundamental constants of the implementation.

3.3.37 preadi.h

This file is for storing physical transfer requests.

3.3.38 proc.h

Defines the structures **proc** and **xproc** to hold all the vital information on processes while they may be swapped out.

3.3.39 proctl.h

A library file used by proctl.

3.3.40 reg.h

Defines constants that provide an index of the available registers relative to AX.

3.3.41 relsym.h

Provides definitions for the following structures:

sym	**reloc**	**xreloc**
asym	**bsym**	

sym defines the structure of the symbol table for **x.out**, **reloc** defines a relocation table entry for the long form of **x.out**, **xreloc** defines the relocation table entry for the short form of **x.out**, and the structures **asym** and **bsym** are provided for compatibility with other systems. Definitions for:

sym.s_type	**reloc.r_desc**
asym.sa_type	**nlist.n_type**

are also provided.

3.3.42 relsym86.h

Contains the declarations for the 8086/80286/80386 symbol table and relocation record structures. The structures **dosexec**, **desctab**, and **srel86** are defined. **dosexec** is provided for DOS support, **desctab** provides the structure of the descriptor table, and **srel86** provides the structure for segment relocation (which is necessary for middle and larger model memory support).

3.3.43 sd.h

Defines the shared data table.

3.3.44 sem.h

Defines the structures used by the semaphore operations system call, **semop**. The structures are:

semid_ds	**sem**	**sem_undo**
seminfo	**sembuf**	

See the **semop**(S) manual page in the XENIX *Reference* for detailed information.

3.3.45 shm.h

Defines values, and the *shmid_ds* structure for shared memory operations. See the **shmop** manual page in the XENIX *Reference* for detailed information.

3.3.46 signal.h

Defines the values for the SIGNAL constants.

See **signal**(S) in the XENIX *Reference* for more information.

3.3.47 sites.h

Provides values for system constants (that are used in the structure defined in *utsname.h*).

3.3.48 space.h

Defines data structures for the XENIX kernel.

3.3.49 stat.h

Defines the structure **stat**, which returns a structure to both the **stat** and **fstat** system calls. It also defines a number of constants.

3.3.50 sysinfo.h

Defines the structures **sysinfo** and **syswait**, which hold information about the state of the system and its processes.

3.3.51 sysmacros.h

Defines a number of machine-dependent macros.

3.3.52 systm.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.

3.3.53 termio.h

This file defines structures to control i/o for serial terminals.

3.3.54 text.h

Defines the structure **text**, which provides the format for text segments. It also defines a number of constants.

3.3.55 timeb.h

Defines the structure **timeb**, returned by the **ftime** system call.

See **time(S)** in the XENIX *Reference* for more information.

3.3.56 times.h

Defines the structure **tms**, returned by the routine **times**.

See **times** in the XENIX *Reference* for more information.

3.3.57 ttold.h

Defines the structures **sgtty** and **tc**, which contain information for the **stty** and *gtty* system calls. It also defines the terminal modes.

3.3.58 tty.h

Defines the structures:

tty	**clist**	**cblock**
chead	**inter**	

The **tty** structure formats the I/O information for each character device. It defines a number of internal state variables, and device commands.

3.3.59 types.h

Defines the structure **saddr** and numerous machine-dependent variables.

3.3.60 ulimit.h

Defines values passed to the **ulimit** system call.

3.3.61 user.h

Defines the structure **user**, which contains all the data on a user process that doesn't need to be referenced (and so is swapped with the process). The standard error codes are also redefined here.

3.3.62 utsname.h

Defines the structure **utsname**, which provides general information about system characteristics.

3.3.63 var.h

Defines the structure **var**.

3.4 /usr/include/dos Files

These files are for use in XENIX to DOS cross development. Some of these files are compatible with XENIX and some are for use only in the DOS environment.

3.4.1 assert.h

Defines the **assert** macro.

3.4.2 conio.h

This include file contains the function declarations for the Microsoft C V2.03 compatable console and portIO routines

3.4.3 ctype.h

Defines the **ctype** macros as well as the character conversion macros (**toupper**, etc).

3.4.4 direct.h

This include file contains the function declarations for the library functions related to directory handling and creation.

3.4.5 dos.h

Defines the structs and unions used to handle the input and output registers for the DOS interface routines defined in the V2.0 to V3.0 compatability package. It also includes macros to access the segment and offset values of Microsoft C "far" pointers, so that they may be used by these routines.

3.4.6 errno.h

Defines the system-wide error numbers (set by system calls). Conforms to XENIX standard. extended for compatibility with uniforum standard. See **perror** for corresponding error messages. This list must always agree with the one in perror.c.

3.4.7 fcntl.h

Defines file control options used by the **open** system call. This include file contains the function declarations for the low level file handling and IO functions.

3.4.8 malloc.h

This include file contains the function declarations for the memory allocation functions.

3.4.9 math.h

Constant definitions and external subroutine declarations for the math subroutine library.

3.4.10 memory.h

This include file contains the function declarations for the System V compatable buffer (memory) manipulation routines.

3.4.11 process.h

Define modeflag values for *spawnxx ()* calls. Only P_WAIT and P_OVERLAY are currently implemented on DOS. Also contains the function argument declarations for all process control related routines.

3.4.12 register.h

Definitions for register variable specifiers. Defined for 8086 and 68000.

3.4.13 search.h

This include file contains the function declarations for the sorting and searching routines.

3.4.14 setjmp.h

Defines the machine dependant buffer used by **setjmp** and **longjmp** routines to save the program state.

3.4.15 share.h

File sharing modes for sopen.

3.4.16 signal.h

Define signal values. Only SIGINT is recognized on DOS.

3.4.17 spawn.h

Define modeflag values for **spawnxx** () calls. Only P_WAIT and P_OVERLAY are currently implemented on DOS. Also contains the function argument declarations for all process control related routines.

3.4.18 stdio.h

Defines the structure used by the level 2 I/O ("standard I/O") routines and some of the associated values and macros.

3.4.19 stdlib.h

This include file contains the function declarations for commonly used library functions which either don't fit somewhere else, or, like **toupper** and **tolower**, can't be declared in the normal place (**ctype.h** in the case of **toupper** and **tolower**) for other reasons.

3.4.20 string.h

This include file contains the function declarations for the string manipulation functions.

3.4.21 time.h

Defines the structure returned by the *localtime*() and *gmtime*() routines and used by **asctime**.

3.4.22 v2tov3.h

Defines a group of macros which can be used to ease the problems of porting Microsoft C version 2.0 programs to Microsoft C version 3.0.

3.5 /usr/include/dos/sys Files

These files are DOS system files. Many of them define system parameters, or contain information used by the kernel.

3.5.1 locking.h

Flags for **locking** system call.

3.5.2 stat.h

Defines the structure returned by the **stat** and **fstat** routines.

3.5.3 timeb.h

Structure returned by **ftime** system call.

3.5.4 types.h

Defines types used in defining values returned by system level calls for file status and time information.

3.5.5 utime.h

Defines the structure used by the utime routine to set new file access and modification times. Note thatMS-DOS 2.0 does not recognize access time, so this field will always be ignored and the modification time field will be used to set the new time.

Chapter 4

Using the Standard I/O Functions

4.1 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. Inparticular, it describes:

- Command line arguments
- Standard input and output files
- Stream functions for ordinary files
- Low-level functions for ordinary files
- Random access functions

4.1.1 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.

4.1.2 Special Names

The standard I/O library uses many names for special purposes. In general, these names can he used in any program that has included the *stdio.h* file. The following is a list of the special names:

stdin	The name of the standard input file.
stdout	The name of the standard output file.
stderr	The name of the standard error file.
EOF	The value returned by the read routines on an end-of-file or an error.
NULL	The null pointer, returned by pointer-valued functions, to indicate an error.
FILE	The name of the file type used to declare pointers to streams.
BSIZE	The size in bytes (default is 1024) suitable for an I/O buffer supplied by the user.

4.1.3 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.

4.2 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

```
display3 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.

4.3 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 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 stream functions described in the section "Using 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.

4.3.1 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 (\0) 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.

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 *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 octal digits, and hexadecimal numbers 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 the **raw** function call described in "Setting a Terminal Mode" in Chapter 5 of this Guide.

4.3.2 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.

The function is typically used in a conditional loop to write a string of characters to the standard output. For example, the function:

```
writen (p, cnt)
char p[ ];
int cnt;
{
    int i;

    for (i=0; i<=cnt; i++)
        putchar( (i != cnt) ? p[i] : '\n');
}
```

writes *cnt* number of characters plus a newline character to the standard output.

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 (\0). 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*.

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 *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*.

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.

4.3.3 Redirecting the Standard Input

You can change the standard input from the terminal keyboard to an ordinary file by using the normal shell redirection symbol, <. This symbol directs the shell to open, for reading, the file whose name immediately follows the symbol. For example, the following command line opens the file *phonelist* as the standard input to the program **dial**:

```
dial <phonelist
```

The **dial** program may then use the **getchar**, **gets**, and **scanf** functions to read characters and values from this file. If the file does not exist, the shell displays an error message and stops the program.

Whenever **getchar**, **gets**, or **scanf** are used to read from an ordinary file, they return the value EOF if the end of the file or an error is encountered. It is useful to check for this value to make sure you do not continue to read characters after an error has occurred.

4.3.4 Redirecting the Standard Output

You can change the standard output of a program from the terminal screen to an ordinary file by using the shell redirection symbol, >. The symbol directs the shell to open, for writing, the file whose name immediately follows the symbol. For example, the command line:

```
dial >savephone
```

opens the file *savephone* as the standard output of the program **dial**, and not the terminal screen. You may use the **putchar**, **puts**, and **printf** functions to write to the file.

If the file does not exist, the shell automatically creates it. If the file already exists and the user has write permission, the file will be truncated. If the file exists, but the program does not have permission to change or alter the file, the shell displays an error message and does not execute the program.

4.3.5 Piping the Standard Input and Output

Another way to redefine the standard input and output is to create a pipe. A pipe simply connects the standard output of one program to the standard input of another. The programs may then use the standard input and output to pass information from one to the other. You can create a pipe by using the standard shell pipe symbol, | .

For example, the command line:

```
dial | wc
```

connects the standard output of the program **dial** to the standard input of the program **wc**. (The standard input of **dial** and standard output of **wc** are not affected.) If **dial** writes to its standard output with the **putchar**, **puts**, or **printf** functions, **wc** can read this output with the **getchar** and **scanf** functions.

Note that 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. Pipes are described in more detail in Chapter 8, "Creating and Using Pipes."

4.3.6 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.

```
#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 Chapter 8.

4.4 Using the 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 "stream" of characters. For this reason, the functions are called the stream functions.

Unlike the standard input and output files, a file to be accessed by a 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.

4.4.1 Using File Pointers

Every file opened for access by the 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 FILE type, they are constants, not variables. They must not be assigned values.

4.4.2 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 stream functions to read from or write to the file. See **fopen**(S) in the XENIX *Reference* Guide.

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.

4.4.3 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.

4.4.4 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 be 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 (\0) 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*.

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*.

4.4.5 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 structrure). 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
    {
        charname[20];
        int age;
    } record;

main()
{
    FILE *database, *fopen();
    record person[2];

    if ((database = fopen(dbname, "w")) == NULL) {
        printf("Cannotopen %s0,dbname);
        exit(1);
    }
    fwrite(person, sizeof(record), 2, database);
    printf("0ecord is %d0, sizeof(record));
    printf("person is %d0,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.

4.4.6 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 described in **scanf** (S) in the XENIX *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*.

4.4.7 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 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.

4.4.8 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.

4.4.9 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 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*.

4.4.10 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(&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.

4.4.11 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, size(name), 1, stream);
while(!feof( stream));
```

4.4.12 Testing For File Errors

The **ferror** function tests a given 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*.

```
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.

4.4.13 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.

4.4.14 Program Example

This section shows how you can use the 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 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 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.

4.5 Using More Stream Functions

The 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.

4.5.1 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 mainfest constant BSIZE, which is defined in the **stdio.h** file.

When a file has a buffer, the 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.

4.5.2 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,"r",stdout);
```

This has the same effect as using the redirection symbols in the command line of the program.

4.5.3 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 descriptor 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 **BSIZE** bytes in length, where **BSIZE** is a manifest constant found in *stdio.h*.

The function is typically used to 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(BSIZE);
setbuf(stdout, p);
```

The new buffer is BSIZE 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.

4.5.4 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;
charname[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.

4.5.5 Flushing a File Buffer

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);
```

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.

4.6 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 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 has the complete burden of handling input and output.

The low-level functions, like the 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.

4.7 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.

4.7.1 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 *Reference*.) The logical OR operator (**|**) may be used to combine the constants. The *mode* is required 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 open the new file named */usr/tmp/scratch* for reading and 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 permission for everyone else.

Note that if **O_CREAT** is given and the file already exists, the function destroys the file's old contents.

4.7.2 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.

4.7.3 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.

4.7.4 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.

```
int fd;

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.

4.7.5 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.

The first program copies its standard input to its standard output:

```
#define  BUFSIZE       BSIZE

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** 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        BSIZE

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*.

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 BSIZE
#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.

4.7.6 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 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.

4.7.7 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.

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));
}
```

4.7.8 Moving the Character Pointer in a Stream

The **fseek** function, a 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.

4.7.9 Rewinding a File

The **rewind** function, a 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.

4.7.10 Getting the Current Character Position

The **ftell** function, a 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 stream file.

Chapter 5

Screen Processing

5.1 Introduction

This chapter explains how to use the screen updating and cursor movement library named **curses**. The library provides 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.

5.1.1 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.

The **termcap curses** routines are summarized in the **curses**(S) manual page. The **terminfo curses** routines are completely summarized in the **terminfo**(S) manual page. The extensions provided by **terminfo curses** over **termcap curses** are not described in this chapter.

The **terminfo curses** package available with XENIX includes a number of extensions over other versions of **curses**. These extensions provide for superior handling of *typeahead* and *function keys*. These extensions require that programs using **terminfo curses** be compiled with the XENIX extensions library (**-lx**).

5.1.2 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 and output files. Screen processing does not rely on the terminal's type. Instead, the screen processing functions use a XENIX terminal capability description database, either */etc/termcap* or */usr/lib/terminfo*, to tailor their actions for any given terminal. This makes a screen processing independent. The program can be run with any terminal as long as that terminal is described by the appropriate terminal description database.

The screen processing functions access a terminal screen by working through intermediate "screens" and "windows" in memory. A screen is a representation of what the entire terminal screen should look like. A window is a representation of what some portion of the terminal screen should look like. A screen can be made up of one or more windows. 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 by using the *newwin*() or *subwin*() functions. 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, may 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* (for "standard screen"). The *stdscr* may be used without first creating it. The function also creates *curscr* (for "current screen") which contains a copy of what is currently on the terminal screen.

To display characters at 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.

5.2 Using the Library

To use **curses** with your program, you must first decide which version of **curses** to use. The five basic options are described below.

5.2.1 System Default

The first option is to use the default **curses** library and terminal description database. The default library is chosen at installation time and can be one of three values:

- **termcap curses** using either */etc/termcap* or **terminfo** (depending on how you link your program).
- **terminfo curses** using **terminfo**.
- No default **curses**.

Since your system's default may not be the same as any other system's default, using the default system **curses** is not recommended.

To use the default **curses** library, add the line:

```
#include <curses.h>
```

to the beginning of your program. When you link your program, you should use the command line:

cc *files* **-lcurses** [*other libraries*]

The *other libraries* you include on your command line depend on which version of **curses** is the default. If the default is **termcap curses**, include either the library:

-ltermcap

to use */etc/termcap*, or

-ltinfo

to use **terminfo**.

If the default is **terminfo curses**, include the library:

```
-lx
```

The command line:

```
cc files -lcurses -ltermcap -lx
```

should work regardless of the default **curses** and uses either **termcap curses** with */etc/termcap* or **terminfo curses** with **terminfo**.

5.2.2 termcap curses Using /etc/termcap

In this method **termcap curses** is used with the */etc/termcap* terminal description database. You must place the line:

```
#include <tcap.h>
```

in the program's source instead of:

```
#include <curses.h>
```

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.

5.2.3 termcap curses Using terminfo

In this method you use **termcap curses** with the **terminfo** database. You must place the line:

```
#include <tcap.h>
```

at the beginning of your program's source instead of:

```
#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**.

5.2.4 terminfo curses Using terminfo

In this method you use **terminfo curses** with the **terminfo** database. You must place the line:

```
#include <tinfo.h>
```

at the beginning of your program source instead of:

```
#include <curses.h>
```

and link your program with the command:

```
cc files -ltinfo [-lx]
```

The XENIX extensions library (**-lx**) may or may not be required depending on which features of **curses** you have used. This technique is specific to and is not portable outside of XENIX.

5.2.5 System Independent curses

You use your choice of the **curses** libraries and the terminal description utilities in a fashion that is portable outside of XENIX. With this technique, you still use the line:

```
#include <curses.h>
```

at the beginning of the source of your program. Your program's source does not need to change with this method.

To choose which **curses** to use, you must define either **M_TERMCAP** or **M_TERMINFO**. If you define **M_TERMCAP**, **termcap curses** is used. If you define **M_TERMINFO**, **terminfo curses** is used. To define one or the other, you should use the command line:

cc -DM_*curses file*

when compiling your program. In the above example, **M_***curses* is one of the two names **M_TERMCAP** or **M_TERMINFO**. Or you may add the lines:

```
#ifdef M_XENIX
#define M_curses
#endif
```

to the beginning of your program before you include **curses.h**.

When you compile your program, you should use one of the following command lines:

cc *files* **-ltcap -ltermcap**	**termcap curses** using */etc/termcap*.
cc *files* **-ltcap -ltinfo**	**termcap curses** using **terminfo**.
cc *files* **-ltinfo [-lx]**	**terminfo curses** using **terminfo**.

This technique is not specific to and is portable outside of XENIX. This is the recommended method of including the **curses** library. Note that the only change required is to your **cc** command line. Your program source should not require any changes with this method. The examples in this chapter assume that you are using this method.

5.2.6 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>
```

5.2.7 Predefined Names

The screen processing library has a variety of predefined names. These names refer to variables, manifest constants, and types that can be used with the library functions. The following is a list of these names:

Variables

Type	Name	Description
WINDOW*	**curscr**	A pointer to the current version of the terminal screen.
WINDOW*	**stdscr**	A pointer to the default screen used for updating when no explicit screen is defined.
char	**Def_term**	A pointer to the default terminal type if the type cannot be determined.
bool	**My_term**	The terminal type flag. If set, it causes the terminal specification in "Def_term" to be used, regardless of the real terminal type.
char	**ttytype**	A pointer to the full name of the current terminal.
int	**LINES**	The number of lines on the terminal.
int	**COLS**	The number of columns on the terminal.
int	**ERR**	The error flag. Returned by functions on an error.
int	**OK**	The okay flag. Returned by functions on successful operation.

Types and Constants

Name	Description
reg	A storage class. It is the same as **register** storage class.
bool	A type. It is the same a **char** type.
TRUE	The Boolean true value (1).
FALSE	The Boolean false value (0).

5.3 Preparing the Screen

The **initscr** and **endwin** functions perform the operations required to initialize and terminate programs that use the screen processing functions. The following sections describe these functions and how they affect the terminal.

5.3.1 Initializing the Screen

The **initscr** function initializes screen processing for a program by allocating the required memory space for the screen processing functions and variables, and by setting the terminal to the proper modes. The function call has the form:

```
        initscr()
```

No arguments are required.

The **initscr** function must be used to prepare the program for subsequent calls to other screen processing functions and for the use of the screen processing variables. For example, in the following program fragment, **initscr** initializes the screening processing functions.

```
        #include <curses.h>

        main ()
        {

        initscr();
        if ( strcmp(ttytype,"dumb") )
        fprintf(stderr, "Terminal type can't displayscreen.");
```

In this example, the predefined variable *ttytype* is checked for the current terminal type.

The function returns (WINDOW*) ERR if memory allocation causes an overflow.

5.3.2 Using Terminal Capability and Type

The **initscr** function uses the terminal capability descriptions given in the XENIX system's terminal description database (either */etc/termcap* or *terminfo*) to prepare the screen processing functions for creating and updating terminal screens. 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 library when screen processing is initialized, so direct access by a program is not required.

The **initscr** function uses the "TERM" environment 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 one of the description utilities.

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 "dumb" (for "dumb terminal"), but the user may 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."

Terminal capabilities, types, and identifiers are described in detail in **termcap**(M) and **terminfo**(M) in the XENIX *Reference*.

5.3.3 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 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 the section "Setting a Terminal Mode" in this chapter. If the modes are changed, they must be changed immediately after calling **initscr**. Terminal modes are described in detail in **termio**(M) in the XENIX *Reference*.

Note

The terminal mode functions should only be used in conjunction with other screen processing functions. They should not be used alone.

5.3.4 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 refreshing from 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, and the clear flag prevents the characters on the screen from being cleared before being updated. The flags may be changed by using the functions described in the section "Setting Window Flags," in this chapter.

5.3.5 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.

5.3.6 Terminating Screen Processing

The **endwin** function terminates screen processing in a program by freeing all memory resources allocated by the screen processing functions and restoring the terminal to its previous state, prior to screen processing. The function call has the form:

```
endwin()
```

No arguments are required.

The **endwin** function must be used before leaving a program that has called the **initscr** function to restore the terminal to its previous state. 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>

main ()
{
initscr();
/*
/* Program body... */
/*
endwin();
}
```

Note that **endwin** must not be called if **initscr** has not been called. Also, **endwin** should be called before any call is made to the **exit** function. The **endwin** function must also be called if the **gettmode** and **setterm** functions have been called, even if **initscr** has not.

5.4 Using the Standard Screen

The following sections explain how to use the standard screen to display and edit characters on the terminal screen.

5.4.1 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 form:

```
addch(ch)
```

where *ch* gives the character to be added and must have **char** type. For example, if the current position is (0, 0), the function call:

```
addch('A')
```

places the letter "A" at this position and moves the pointer to (0, 1).

If a newline ('\n') character is given, the function deletes all characters from the current position to the end of the line and moves the pointer one line down. If the newline flag is set, the function deletes the characters and moves the pointer to the beginning of the next line. If a return ('\r') is given, the function moves the pointer to the beginning of the current line. If a tab ('\t') is given, the function 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 function returns ERR if it encounters an error, such as illegal scrolling.

5.4.2 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 form:

```
addstr(str)
```

where *str* is a character pointer to the given string. For example, if the current position is (0,0), the function call:

```
addstr("line");
```

places the beginning of the string *line* at this position and moves the pointer to (0,4).

If the string contains newline, return, or tab characters, the 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 function returns ERR if it encounters an error such as illegal scrolling.

5.4.3 Printing Strings, Characters, and Numbers

The **printw** function prints one or more values on the standard screen, where a value may be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the form:

printw(*fmt*[, *arg*]...)

where *fmt* is a pointer to a string that defines the format of the values, and *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*. A format may be *%s* for string, *%c* for 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 *%s* is given, the corresponding *arg* must be a character pointer. For other formats, the actual value or a variable containing the value may be given.

The function is typically used to copy both numbers and strings to the standard screen at the same time. For example, if the current position is (0,0), the function call:

```
printw("%s %d", name, 15);
```

prints the name given by the variable *name*, starting at position (0,0). It then prints the number 15 immediately after the name.

The function returns ERR if it encounters an error such as illegal scrolling.

5.4.4 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 form:

```
    c = getch()
```

where *c* is the variable to receive the character.

The 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 newline or the end of the file 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** copies 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 later in the chapter.

The function returns ERR if it encounters an error such as illegal scrolling.

5.4.5 Reading a String From the Keyboard

The **getstr** function reads a string of characters from the terminal keyboard and copies the string to a given location. The function call has the form:

```
    getstr(str)
```

where *str* is a character pointer to the variable or location to receive the string. When entered at the keyboard, the string must end with a newline character or with the end-of-file character. The extra character is replaced by a null character when the string is stored. It is the programmer's responsibility to ensure that *str* has adequate space to store the entered string.

The function is typically used to read 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 termina is set to ECHO mode, **getstr** 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. Terminal modes are described later in the chapter.

The function returns ERR if it encounters an error such as illegal scrolling.

5.4.6 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 may be a string, character, or decimal, octal, or hexadecimal number. The function call has the form:

scanw(*fmt*, *argptr* ...)

where *fmt* is a pointer to a string defining the format of the values to be read, and *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 may be *%s* for string, *%c* for character, and *%d*, *%o*, or *%x* for a decimal, octal, or hexadecimal number, respectively. (Other formats are described in **scanf**(S) in the XENIX *Reference*.)

The function is typically used to read 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.

5.4.7 Moving the Current Position

The **move** function moves the pointer to the given position. The function call has the form:

```
move (y,x)
```

where *y* is an integer value giving the new row position, and *x* is an integer value giving the new column position. For example, if the current position is (0,0), the function call:

```
move(5,4)
```

moves the pointer to line 5, column 4.

The function returns ERR if it encounters an error such as illegal scrolling.

5.4.8 Inserting a Character

The **insch** function inserts a character at the current position and shifts the existing character (and all characters to its right) one position to the right. The function call has the form:

```
insch (c)
```

where *c* is the character to be inserted.

The 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.

5.4.9 Inserting a Line

The **insertln** 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 form:

```
insertln()
```

No arguments are required.

The 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.

5.4.10 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 form:

```
delch()
```

No arguments are required.

The 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 0:

```
int cnt;

while ( cnt != 0 ) {
    delch();
    cnt-- ;
    }
```

5.4.11 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 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();
```

5.4.12 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();
```

5.4.13 Clearing a Part of the Screen

The **clrtobot** 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 **clrtobot** function clears the screen from the current position to the bottom of the screen. For example, if the current position is (10,0), the function call:

```
clrtobot();
```

clears all characters from line 10 and all lines below line 10.

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 function call:

```
clrtoeol();
```

clears all characters from (10,10) to (10,79). The characters at the beginning of the line remain unchanged.

Note that both the **clrtobot** and **clrtoeol** functions do not change the current position.

5.4.14 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 form:

```
refresh()
```

No arguments are required.

The 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 been 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.

5.5 Creating and Using Windows

The following sections explain how to create and use windows to display and edit text on the terminal screen.

5.5.1 Creating a Window

The **newwin** function creates a window and returns a pointer that may be used in subsequent screen processing functions. The function call has the 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, respectively, in the window, and *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 have type *WINDOW**.

The 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 and assigns the 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 position (9,35) on the terminal screen.

If either *lines* or *cols* is zero, the function automatically creates a window that has "**LINES** - *begin_y*" lines or "**COLS** - *begin_x*" columns, where "**LINES**" "**COLS**" are the predefined constants giving the total number of lines and columns on the terminal screen. For example, the function call:

```
newwin(0, 0, 0, 0)
```

creates a new window whose upper left corner is at position (0,0) and that has "**LINES**" lines and "**COLS**" columns.

Note

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.

5.5.2 Creating a Subwindow

The **subwin** function creates a subwindow and returns a pointer to the new window. A subwindow is a window which 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 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, respectively, in the subwindow, and *begin_y* and *begin_x* are integer values that give the line and column position, respectively, of the upper left corner of the subwindow when displayed on the terminal screen. The *swin* variable must have type *WINDOW**.

The 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.

5.5.3 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 form:

```
waddch(win, ch)
```

where *win* is a pointer to the window to receive the character, and *ch* gives the character to be added. *ch* must have **char** type. For example, if the current position in the window *midscreen* is (0,0), the function call:

```
waddch(midscreen, 'A')
```

places the letter *A* at this position and moves the pointer to (0,1).

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 form:

```
waddstr(win, str)
```

where *win* is a pointer to the window to receive the string, and *str* is a character pointer to the given string. For example, if the current position is (0,0), the function call:

```
waddstr(midscreen, "line");
```

places the beginning of the string "line" at this position, and moves the pointer to (0,4).

The **wprintw** function prints one or more values on the given window, where a value may be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the form:

```
wprintw(win, fmt[, arg]...)
```

where *win* is a pointer to the window to receive the values, *fmt* is a pointer to a string that defines the format of the values, and *arg* is a value to be printed. If more than one *arg* is given, each must be separated from the preceding with a comma (,). For each *arg* given, there must be a corresponding format given in *fmt*. A format may be *%s* for string, *%c* for 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 *%s* is given, the corresponding *arg* must be a character pointer. For other formats, the actual value or a variable containing the value may be given.

The 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 newline, 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 errors such as illegal scrolling.

5.5.4 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 form:

c = wgetch(*win*)

where *win* is a pointer to a window, and *c* is the character variable to receive the character.

The 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 form:

wgetstr(*win*, *str*)

where *win* is a pointer to a window, and *str* is a character pointer to the variable or location to receive the string. When entered at the keyboard, the string must end with a newline character or with the end-of-file character. The extra character is replaced by a null character when the string is stored.

It is the programmer's responsibility to ensure that *str* has adequate space for storing the typed string.

The function is typically used to read 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 may be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the form:

wscanw(*win*, *fmt* [, *argptr*] ...)

where *win* is a pointer to a window, *fmt* is a pointer to a string defining the format of the values to be read, and *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 may be *%s* for string, *%c* for character, and *%d*, *%o*, or *%x* for a decimal, octal, or hexadecimal number, respectively. (Other formats are described in **scanf**(S) in the XENIX *Reference*.)

The function is typically used to read 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(midscreen, "%s %d", name, &id);
```

In this example, the name is stored in the character array *name* and the number in the integer variable *id*.

If the terminal is set to ECHO mode, the function copies the string to the given window. 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 functions return ERR if they encounter errors such as illegal scrolling.

5.5.5 Moving the Current Position in a Window

The **wmove** function moves the current position in a given window. The function call has the form:

```
    wmove(win,y,x)
```

where *win* is a pointer to a window, *y* is an integer value giving the new line position, and *x* is an integer value giving the new column position. For example, the function call:

```
    wmove(midscreen, 4, 4)
```

moves the current position in the window *midscreen* to (4,4). The function returns ERR if it encounters an error such as illegal scrolling.

5.5.6 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 existing character (and all characters to its right) one position to the right. The function call has the form:

```
    winsch ( win, c )
```

where *win* is a pointer to a window, and *c* is the character to be inserted.

The function is typically used to edit the contents of the given window. For example, the function call:

```
    winsch(midscreen, 'X');
```

inserts the character *X* at the current position in the window *midscreen*.

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 form:

```
    winsertln( win )
```

where *win* is a pointer to the window to receive the blank line.

The 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.

5.5.7 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 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 with a space. The function call has the form:

    wdelch(*win*)

where *win* is a pointer to a window.

The function is typically used to edit the contents of the standard screen. For example, the function call:

```
    wdelch(midscreen);
```

deletes the character at the current position in the window *midscreen*.

The **wdeleteln** 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 in the screen blank. The function call has the form:

    wdeleteln(*win*)

where *win* is a pointer to a window.

The 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--;
        }
```

5.5.8 Clearing the Screen

The **wclear**, **werase**, **wclrtobot**, 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 form:

```
wclear(win)
```

where *win* is the window to be cleared.

The **werase** function clears the given window, 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 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 form:

```
wclrtohot(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 function call:

```
wclrtobot( midscreen );
```

clears all characters from line 10 and all lines below line 10.

The **wclrtoeol** function clears the standard screen from the current position to the end of the current line. The function call has the 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 function call:

```
wclrtoeol( midscreen );
```

clears all characters from (10,10) to the end of the line. The characters at the beginning of the line remain unchanged.

Note that the **wclrtobot** and **wclrtoeol** functions do not change the current position.

5.5.9 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 form:

```
    wrefresh(win)
```

where *win* is a pointer to a window.

The function is used solely to display 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 been 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.

5.5.10 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 form:

```
    overlay( win1, win2)
```

where *win1* is a pointer to the window to be copied, and *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*.

The 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();
```

5.5.11 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. The function call has the form:

```
    overwrite( win1, win2)
```

where *win1* is a pointer to the window to be copied, and *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*.

The function is typically used to display the contents of a temporary window in the middle of a larger window. For example, in the following program fragment, **overwrite** is used to copy the contents of a work window to the standard screen.

```
    WINDOW *work;

    overwrite(work, stdscr);
    refresh();
```

5.5.12 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 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, and *x* is an integer value giving the column to which the corner is to be moved.

The 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 a error such as an attempt to move part of a window off the edge of the screen.

5.5.13 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 form:

```
c = 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 form:

```
c = winch(win)
```

where *win* is the pointer to the window containing the character to be read.

The 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.

5.5.14 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 form:

touchwin(*win*)

where *win* is a pointer to the window to he touched.

The function is typically used when two or more overlapping windows makeup the terminal screen. For example, the function call:

```
touchwin(leftscreen);
```

is used to touch the window named *leftscreen*. A subsequent **refresh** copies all characters in *leftscreen* to the terminal screen.

5.5.15 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 form:

delwin(*win*)

where *win* is the pointer to the window to be deleted.

The function is typically used to remove temporary windows from a program or to free memory space for other uses. For example, the function call:

```
    delwin(midscreen);
```

removes the window named *midscreen*.

5.6 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 window.

5.6.1 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 form:

box(*win*, *vert*, *hor*)

where *win* is the pointer to the desired window, *vert* is the vertical character, and *hor* is the horizontal character. Both *ver* and *hor* must have **char** type.

The 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.

5.6.2 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 form:

```
standout()
```

No arguments are required.

The **wstandout** function sets the standout attribute of characters added to the given window or screen. The function call has the form:

wstandout(*win*)

where *win* is a pointer to a window.

The 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. See **termcap**(M) and **terminfo**(M) in the XENIX *Reference*.

5.6.3 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 form:

```
standend()
```

No arguments are required.

The **wstandend** function restores the normal attribute for a given window or screen. The function call has the form:

wstandend(*win*)

where *win* is a pointer to a window.

The 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();
    }
```

5.6.4 Getting the Current Position

The **getyx** function copies the current line and column position of a given window pointer to a corresponding pair of variables. The function call has the 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, and *x* is the integer variable to receive the column position.

The 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 position in the variables *line* and *column*:

```
int line, column;

getyx(stdscr, line, column);
```

5.6.5 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 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 form:

```
    leaveok(win, state)
```

where *win* is a pointer to the window containing the flag to be set, and *state* is a Boolean value defining the state of the flag. If *state* is TRUE, the flag is set; if FALSE, the flag is cleared. For example, the function call

```
    leaveok(stdscr, TRUE);
```

sets the cursor flag.

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, then no scrolling is allowed. The function call has the form:

```
    scrollok(win, state)
```

where *win* is a pointer to a window, and *state* is a Boolean value defining how the flag is to be set. If *state* is TRUE, the flag is set; if 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 form:

```
    clearok(win, state)
```

where *win* is a pointer to the desired screen, and *state* is a Boolean value. The function sets the flag if *state* is TRUE, and clears the flag if FALSE. For example, the function call:

```
    clearok(stdscr, TRUE)
```

sets the clear flag for the standard screen.

When the clear flag is set, each **refresh** call to the given terminal 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.

5.6.6 Scrolling a Window

The **scroll** function scrolls the contents of a given window upward by one line. The function call has the form:

```
    scroll( win )
```

where *win* is a pointer to the window to be scrolled. The function should be used in special cases only.

5.7 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 form:

```
    mvfunc ( [ win, ]y, x [, arg ]... )
```

where *func* is the name of a function, *win* is a pointer to the window to be operated on (*stdscr* is used if none is given), *y* is an integer value giving the line to move to, *x* is an integer value giving the column to move to, and *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 function call:

```
    mvaddch(10, 5, 'X');
```

moves the position to (10,5) and adds the character *X*. The operation is the same as moving the position with the **move** function and then adding a character with **addch**.

A complete list of the functions which may be used with the movement prefix is given in **curses**(S) and **terminfo** (S) in the XENIX *Reference*.

5.8 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 only be used when using other screen processing functions.

5.8.1 Setting a Terminal Mode

The **crmode**, **echo**, **nl**, and **raw** functions set the terminal mode, causing subsequent input from the terminal's keyboard to be processed accordingly.

The **crmode** function sets the CBREAK mode for the terminal. The 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 form:

```
    crmode()
```

No arguments are required.

The **echo** function sets the ECHO mode for the terminal, causing each character entered at the keyboard to be displayed at the terminal screen. The function call has the form:

```
    echo()
```

No arguments are required.

The **nl** function sets a terminal to NEWLINE mode, causing all newline characters to be mapped to a corresponding newline and return character combination. The function call has the form:

```
    nl()
```

No arguments are required.

The **raw** function sets the RAW mode for the terminal, causing each character entered 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 newline characters into newline and return combinations. The function call has the form:

```
    raw()
```

No arguments are required.

5.8.2 Clearing a TerminalMode

The **nocrmode**, **noecho**, **nonl**, and **noraw** functions clear the current terminalmode, 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 form:

```
    nocrmode()
```

No arguments are required.

The **noecho** function clears a terminal from the ECHO mode. This mode prevents characters entered at the keyboard from being displayed on the terminal screen. The function call has the form:

```
noecho()
```

No arguments are required.

The **nonl** function clears a terminal from NEWLINE mode, causing newline characters to be mapped into themselves. This allows the screen processing functions to perform better optimization. The function call has the form:

```
nonl()
```

No arguments are required.

The **noraw** function clears a terminal from RAW mode, restoring normal editing and signal generating function to the keyboard. The function call has the form:

```
noraw()
```

No arguments are required.

5.8.3 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 form:

mvcur(*last_y*, *last_x*, *new_y*, *new_x*)

where *last_y* and *last_x* are integer values giving the last line and column position of the cursor, and *new_y* and *new_x* are integer values giving the new line and column position of the cursor. For example, the function call:

```
mvcur(10, 5, 3 , 0)
```

moves the cursor from (10,5) to (3,0) on the terminal screen.

Note

The **mvcur** function should only be used 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.

5.8.4 Getting the Terminal Mode

The **gettmode** function returns the current tty mode. The function call has the form:

s = gettmode()

where *s* is the variable to receive the status.

The function is normally called by the **initscr** function.

5.8.5 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 **initscr** and **endwin** functions. They are not required when performing ordinary screen processing.

5.8.6 Setting a Terminal Type

The **setterm** function sets the terminal type to the given type. The function call has the 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 may be used in special cases.

5.8.7 Reading the Terminal Name

The **longname** function converts a given **termcap** or **terminfo** identifier into the full name of the corresponding terminal. The function call has the form:

longname(*termbuf*, *name*)

where *termbuf* is a pointer to the string containing the terminal type identifier, and *name* is a character pointer to the location to receive the long name. The terminal type identifier must exist in the */etc/termcap* file or *terminfo* library.

The 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.

Chapter 6

Character and String Processing

6.1 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.

6.2 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.

6.2.1 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 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.

6.2.2 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.

6.2.3 Testing for Alphanumerics

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.

6.2.4 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.

6.2.5 Testing for Control Characters

The **iscntrl** 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:

iscntrl (*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, **iscntrl** 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 ( !iscntrl(c))
            fputc( c, outfile );
```

The **fputc** function is ignored if the character is a control character.

6.2.6 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;
```

6.2.7 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.

6.2.8 Testing for Printable Characters

The **isprint** function tests for printabl 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).

6.3 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).

6.3.1 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).

6.3.2 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.

6.3.3 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.

6.4 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.

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.

6.4.1 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 (\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");
```

6.4.2 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.

6.4.3 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 (\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.

6.4.4 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 *inname* are short enough to be stored in *name*:

```
char*inname;
charname[MAX ;

if ( strlen(inname) < MAX)
    strcpy(name, inname);
```

6.4.5 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*.

6.4.6 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.

6.4.7 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*.

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*.

6.4.8 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 *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*.

6.4.9 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 *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 7 Using Process Control

7.1 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 *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.

7.2 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.

7.3 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("Enterfilename: ");
        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 Chapter 4.

7.4 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.

7.5 Starting a New Program

The **execl** and **execv** 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.

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 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[ ];

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 pathname 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.

Note that the **execl** and **execv** functions will not expand metacharacters (e.g., <, >, *, ?, and []) given in the argument list. If a program needs these features, it can use **execl** or **execv** to call a shell as described in the next section.

7.6 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.

7.7 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.

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 overlayed 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.

7.8 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 *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.

7.9 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**.

7.10 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 8

Writing and Using Pipes

8.1 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.

8.2 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.

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.

8.3 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 Chapter 4.

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*.

8.4 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);
```

8.5 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.

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.

8.5.1 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 Chapter 4.

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.

8.5.2 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 Chapter 4. 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.

8.6 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.

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 hangup signals. The last set restores the signals to their original status. The **signal** function is described in detail in Chapter 6 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.

8.7 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 deamon or server program creates and reads from a named pipe, while programs associated with the deamon 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 followingroutine creates a named pipe:

```
# nclude  <sys/stat.h>
extern interrno;

/* ma  e a named pipe, mode 666 */
if(m  nod("/u/eric/pipe", S_IFIFO | 0666, 0)== -1){
    perror("/u/eric/pipe");      /* An error occurred. */
    exit (errno);
}
```

Use **unlin** (S) to remove a named pipe.

```
if (unlink("/u/eric/pipe") == -1) {
    perror("/u/eric/pipe");
    exit(errno);
}
```

Chapter 9

Using System Resources

9.1 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.

In particular, this chapter explains how to:

- Allocate memory for dynamically required storage.
- Lock a file to ensure exclusive use by a program.
- Use semaphores to control access to a resource.
- Share data space 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). These are referred to 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. It is generally recommended that these operations be used instead of the XENIX operations. The XENIX operations are compatible with previous versions of XENIX and should be used only if one is working with software which uses XENIX style operations.

The UNIX operations include file locking, shared data, and semaphores. Programs using the XENIX operations must be linked with the XENIX library, using the **-lx** option. The two memory allocation packages described are available in both UNIX System and XENIX.

It is not possible to use both XENIX and UNIX operations in a compatible fashion. The operations mentioned are valid for one type of operation and cannot be mixed with the other type.

9.2 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 space, then free this space after it has finished using it.

There are four basic memory allocation functions: **malloc**, **calloc**, **realloc**, and **free**. The **malloc** and **calloc** functions are used to allocate space for the first time. The functions allocate a given number of bytes and return a pointer to the new space. The **realloc** function reallocates an existing space, allowing it to be used in a different way. The **free** function returns allocated space to the system.

9.2.1 Allocating Space for a Variable

The **malloc** function allocates space for a variable containing a given number of bytes. The function call has the form:

```
        malloc (size);
```

where *size* is an unsigned number which gives the number of bytes to be allocated. For example, the function call

```
        table=malloc (4);
```

allocates four bytes of storage. The function normally returns a pointer to the starting address of the allocated space, but will return a null pointer if there is not enough memory.

The function is typically used to allocate storage for a group of strings that vary in length. For example, in the following program fragment **malloc** is used to allocate space for ten different strings, each of different length.

```
        int i;
        char temp[100];
        char *string[10];
        char *malloc();
        unsigned isize;

        for (i=0; i < 10; ++i) {
                gets(temp);
                string[i] = malloc(strlen(temp) + 1);
                if (string[i] == NULL) {
                        perror("malloc failed");
                        exit(1);
                }
                strcpy (string[i], temp);
        }
```

In this example, the strings are read from the standard input. Note that 1 must be added to *strlen(temp)* because space is needed at the end of the string for the NULL character.

9.2.2 Allocating Space 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 form:

```
        calloc (n, size);
```

where *n* is the number of elements in the array, and *size* is the number of bytes in each element. 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 memory. For example, the function call

```
table= calloc (10,4);
```

allocates sufficient space for a 10 element array. Each element has 4 bytes.

The function is typically used in programs which 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 longs read from the standard input.

```
int i;
long *table;
unsigned inum;

scanf("% d", &inum);
table= (long *)calloc (inum, sizeof(long));
for (i=0; i<inum;i++)
      scanf("%ld", &table[i]);
```

Note that the number of elements is read from the standard input before the elements are read.

9.2.3 Reallocating Space

The **realloc** function reallocates the space at a given address without changing the contents of the memory space. The function call has the form:

realloc (*ptr*, *size*);

where *ptr* is a pointer to the starting address of the space to be reallocated, and *size* is an unsigned number giving the new size in bytes of the reallocated space. The function normally returns a pointer to the starting address of the allocated space, but will return a null pointer if there is not enough memory.

This function is typically used to keep storage as compact as possible. For example, in the following program fragment, **realloc** is used to remove table entries.

```
int i;
long *table;
unsigned inum;

for (i=inum-1; i >= -1; --i) {
        printf("%ld\n", table[i]);
        table = (long *)realloc(table, 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 space from its current length to the length given by "i*4".

9.2.4 Freeing Unused Space

The **free** function frees unused memory space that had been previously allocated by a **malloc**, **calloc**, or **realloc** function call. The function call has the form:

```
free (ptr);
```

where *ptr* is the pointer to the starting address of the space to be freed. This pointer *must* be the return value of a **malloc**, **calloc**, or **realloc** function.

The function is used exclusively to free space which is no longer used or to free space to be used for other purposes. For example, in the following program fragment **free** frees the allocated space pointed to by the elements of "string."

```
int i;
char *string[10];

for (i=0; i < 10; ++i)
        free (string[i]);
```

9.2.5 Tunable Memory Allocation

The memory allocation package which is 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.

A new memory allocation package, which uses a different time/space tradeoff, is included in System V. The new library routines 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 will most likely not be easily changed to use the new package.

The new library package contains the same basic routines with the same functionality; **malloc**, **free**, **realloc**, and **calloc**. It also contains the new routines **mallopt** and **mallinfo**.

The major benefit of the new package is performance improvement for programs that make heavy use of dynamic memory allocation. This memory allocation module is intended to be used by a sophisticated programmer concerned about performance. No allocation algorithm is perfect for all applications. Special knowledge of the distribution of the size of requests for memory can be used to optimize an algorithm. This algorithm provides several tunable parameters to allow customization of the algorithm. Instrumentation is provided to help in the choice of values for these parameters. This package usually performs better than the standard **malloc** package; for optimum performance, however, these parameters should be tuned.

The new package is provided in *libmalloc.a*, and is accessed via the **-lmalloc** flag at the end of the **cc** command line. It is standard System V, available in UNIX as well as XENIX. The standard package is kept in the standard C library, *libc.a*. Thus developers who want the new package must take a positive step to use it. Programs left alone will work exactly the way they used to work.

The interface to both packages is described in the **malloc**(S) manual page. The new interface is similar to the standard interface with routines having the same names. This allows the use of the new package without the changes to code that would be required if the routines had different names, while protecting naive users from the interface change that would occur if the new package replaced the standard package. In addition, it assures that the same allocation routines will be used by libraries that are used by an application, avoiding fragmentation problems.

The most serious 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. An option allows this property to be used, at the cost of two extra words of overhead per block.

The new malloc has two additional functions: **mallopt** and **mallinfo**. The **mallopt** function provides for control over the allocation algorithm. The **mallinfo** function provides instrumentation describing space usage. This information can be used to determine optimal malloc operation using **mallopt**.

9.2.6 Optimizing Memory Allocation

The **mallopt** function provides control over the allocation algorithm. The function call has the form:

```
mallopt (cmd, value);
```

where *cmd* sets variables that allow tuning the algorithm to allocate memory in the most efficient manner for the application. The values available for *cmd* are: **M_MXFAST** which sets the maximum size of blocks that are very quickly allocated in "large groups", **M_NLBLKS** which sets the size of the "large groups" allocated when blocks less than the size of *maxfast* are encountered, **M_GRAIN** which sets the *grain* used when rounding values, and **M_KEEP** which preserves the data in a freed block until the next allocation (for compatibility with the other malloc). **mallopt** may be called repeatedly, until the first block is allocated.

9.2.7 Gathering Memory Allocation Information

The **mallinfo** function provides instrumentation describing space usage. It returns a structure which is defined in *<malloc.h>*. The function call has the form:

```
#include <malloc.h>

struct mallinfo mi;

mi = mallinfo();
```

Refer to the **malloc**(S) manual page and the *<malloc.h>* header file for more information.

9.2.8 Accessing Additional Memory Segments

brkctl(S) is a XENIX routine that allows 8086/80286 programs to access additional data segments. It is not available under UNIX System V, and so is not portable. Use it only when malloc(S) is not sufficient.

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 library simulates the use of an additional segment, via shared memory, if the **brkctl** fails.

brkctl(S) also has several 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** as shown in the following program 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 fars 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]);
}
```

This example allocates 40,000 bytes in a far data segment and fills this memory with the integers from 1 to 20,000.

Be aware that since *fp* is a far pointer, it cannot be passed to the standard small or medium model library functions (e.g. **strcpy**) that expect near pointers.

9.3 Locking Files

Locking a file is a way to synchronize file use when several processes may require access to a single file. The standard C library and the XENIX library provide three file locking functions: **locking**, **lockf**, and **fcntl**. These functions lock any given section of a file, preventing all other processes which wish to use the section from gaining access. A process may lock the entire file or only a small portion. In any case, only the locked section is protected; all other sections may be accessed by other processes as usual. See the **locking**(S), **lockf**(S), and **fcntl**(S) manual pages in the XENIX *Reference*.

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. Before a file can be locked, however, it must be prepared using the **open** and **lseek** functions described in Chapter 2, "Using the Standard I/O Functions." This section describes in detail how to use the **locking** function. The other file locking functions are used in a similar manner.

This is a brief summary of differences between file locking techniques. **lockf**(S) and **fcntl**(S) are UNIX System V style file locking routines and their use is recommended over the XENIX style routine, **locking**(S).

locking and **fcntl** can do a write lock, allowing reads of locked areas. **lockf** only does read and write locks.

The syntax and arguments for **lockf** and **locking** are essentially the same. The only difference is the #defines for the commands (for example, F_LOCK vs. LK_LOCK). The manner of calling **fcntl** is similar to **ioctl**(S). There is no need to do a **seek** before the lock with **fcntl**, since the data structure passed to **fnctl** includes the file offset at which locking is to start.

All three locking styles are *enforced*. When another process tries to access an area that is locked, that process suspends execution or gets an error return. Be aware that it is possible to lock a region beyond the end of a file in all three styles.

To use the **locking** function, you must add the lines

```
#include <sys/types.h>
#include <sys/locking.h>
```

to the beginning of the program. The file *sys/locking.h* contains definitions for the modes used with the function.

9.3.1 Preparing a File for Locking

Before a file can be locked, it must first be opened using the **open** function, then properly positioned by using the **lseek** function to move 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 for writing. The file need not be opened for reading. The file */usr/include/fcntl.h* must be included when using **open**. The **lseek** function may 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 *reservations* file at file position 1024.

```
#include <fcntl.h>

fd = open("reservations", O_WRONLY);
lseek(fd, 1024L, 0);
```

9.3.2 Locking a File

The **locking** function locks one or more bytes of a given file. The function call has the form:

```
locking (filedes, mode, size);
```

where *filedes* is the file descriptor of the file to be locked, *mode* is an integer value which 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* may be "LK_LOCK" for locking the given bytes, or "LK_UNLK" for unlocking them. For example, in the following program fragment, **locking** locks 100 bytes at the beginning of the file given by "fd".

```
#include <fcntl.h>

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.

9.3.3 Program Example

This section shows how to lock and unlock a small section in an existing file using the *locking* function. In the following program, the function locks 50 bytes in the file *data* which is opened for reading and writing. The locked portion of the file is accessed, then **locking** is used again to unlock the file.

```
#include <sys/types.h>
#include <sys/locking.h>
#include <fcntl.h>

main()
{
        int fd, err;
        char *data;

        fd = open("data",O_RDWR);
        if(fd == -1)
                perror("open failed");
        else {
                /*
                 * seek to position 100 and lock 50 bytes
                 */
                lseek(fd, 100L, 0);
                err = locking(fd, LK_LOCK, 50L);
                if(err == -1){
                        /* process error return */
                }

                /* read or write bytes 100 - 150 in the file */

                /*
                 * unlock the region
                 */
                lseek(fd, 100L, 0);
                locking(fd, LK_UNLCK, 50L);
        }
}
```

9.4 Using Semaphores

The standard C library and the XENIX library provide a group of functions, called the semaphore functions, which may be used to control the access to a given system resource. These functions create, open, and request control of "semaphores."

XENIX System V supports two sets of system calls for dealing with semaphore operations. These are referred to in subsequent descriptions as either XENIX semaphores or UNIX System V semaphores.

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. A process that wishes to take control of a semaphore away from another process must wait until that process relinquishes control. Semaphores can be used 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.

The XENIX semaphore operations are compatible with previous releases of XENIX. The system calls for manipulating XENIX semaphores are: **opensem**(S), **creatsem**(S), **sigsem**(S), **waitsem**(S), and **nbwaitsem**(S). Programs using these operations must be linked with the XENIX library, using the **-lx** option.

The **creatsem** function creates a semaphore. The semaphore may then be opened and used by other processes. A process can open a semaphore with the **opensem** function and request control of a semaphore with the **waitsem** or **nbwaitsem** function. Once a process has control of a semaphore it can carry out tasks using the given resource. All other processes must wait. When a process has finished accessing the resource, it can relinquish control of the semaphore with the **sigsem** function. This lets other processes get control of the semaphore and use the corresponding resource.

The UNIX System V semaphore operations are compatible with AT&T UNIX System V. The system calls for manipulating UNIX semaphores are: **semop**(S), **semctl**(S), and **semget**(S).

It is not possible to use both XENIX and UNIX semaphores in a compatible fashion. The operations mentioned are valid 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, and UNIX System V semaphore operations briefly.

9.4.1 Creating a Semaphore

The **creatsem** function creates a semaphore, returning a semaphore number which may be used in subsequent semaphore functions. The function call has the form:

```
creatsem (sem_name, mode);
```

where *sem_name* is a character pointer to the name of the semaphore, and *mode* is an integer value which defines the access mode of the semaphore. Semaphore names have the same syntax as regular file names. The names must be unique. The function normally returns an integer semaphore number which may be 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.

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 preceding with its tasks.

```
main()
{
        int tty1;
        FILE *ftty1;

        tty1 = creatsem("tty1", 0777);
        ftty1 = fopen("/dev/tty01", "w");
              /* Program body. */
}
```

Note that **fopen** is used immediately after **creatsem** to open the file */dev/tty01* 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 may 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 may be accessed only by using the **waitsem** **nbwaitsem** or **sigsem** functions. The **creatsem** function may 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**(S) function.

9.4.2 Opening a Semaphore

The **opensem** function opens an existing semaphore for use by the given process. The function call has the 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 may 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.

The function is typically used by a process just 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*.

```
        int sem1;

        if ( (sem1 = opensem("semaphore1") ) != -1)
                waitsem(sem1);
```

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.

A semaphore must not be opened 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.

9.4.3 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 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 -1 if it encounters an error such as requesting a semaphore that does not exist or requesting a semaphore that is locked to a dead process.

The 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".

```
        int tty1;
        FILE *ftty1;

        waitsem( tty1 );
        fprintf( ftty1, "Changing tty driver\n");
```

The function waits until current controlling process relinquishes control of the semaphore before returning to the next statement.

9.4.4 Checking the Status of a Semaphore

The **nbwaitsem** function checks the current status of a semaphore. If the semaphore is not available, the function returns the error value ENAVAIL. Otherwise, it gives immediate control of the semaphore to the calling process. The function call has the form:

nbwaitsem (*sem_num*);

where *sem_num* is the semaphore number of the semaphore to be checked. The function returns −1 if it encounters an error such as requesting a semaphore that does not exist. The function also returns −1 if the process controlling the requested semaphore terminates without relinquishing control of the semaphore.

The function is typically used in place of **waitsem** to take control of a semaphore.

9.4.5 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 form:

sigsem (*sem_num*)

where *sem_num* is the semaphore number of the semaphore to relinquish. The semaphore must have previously created or opened by the process. Furthermore, the process must have been previously taken control of the semaphore with the **waitsem** or **nbwaitsem** function. The function returns −1 if it encounters an error such as trying to take control of a semaphore that does not exist.

The function is typically used after a process has finished accessing the corresponding device or system resource. This allows waiting processes to take control. For example, in the following program fragment, **sigsem** signals the end of control of the semaphore "tty1".

```
int tty1;
FILE *temp, *ftty1;

waitsem(tty1);
while ((c=fgetc(temp)) != EOF)
        fputc(c, ftty1);
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 signal the end of the control before terminating. In such a case, the semaphore must be reset by using the **creatsem** function.

9.4.6 Program Example

This section shows how to use the semaphore functions to control the access of a system resource. The following program creates five different processes which 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 program performs no meaningful work, it clearly illustrates the use of semaphores.

```
#define     NPROC     5

char    mf[]= "_kesemfXXXXXX";
int     sem_num;
int     holdsem = 5;

main()
{
        r gister i, child;

        mktemp(semf);
        if ((sem_num = creatsem(semf, 0777)) < 0)
                err("creatsem");
        for (i = 1; i < NPROC; ++i) {
                if((child = fork()) < 0)
                        err("No fork");
                elseif(child ==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)
int id;
{
        while(holdsem—) {
                if(waitsem(sem_num) < 0)
                        err("waitsem");
                printf("%d\n", id);
                sleep(1);
                if(sigsem(sem_num) < 0)
                        err("sigsem");
        }
}

 rr(s)
char*s;
{
        perror(s);
        exit(1);
}
```

The program contains a number of global variables. The array "semf" contains the semaphore name. The name is used by the **creatsem** and **opensem** functions. The variable "sem_num" is the semaphore number. This is the value returned by **creatsem** and **opensem** and eventually used 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(S)** 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, it 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 *doit*(), waits for termination of each child process and finally deletes the semaphore with the **unlink(S)** function.

doit() calls the **waitsem** function to request control of the semaphore. The function waits until the semaphore is available, it then prints an integer between 0 and 4 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.

9.4.7 UNIX System V Semaphores Program Example

The following program example is essentially the same as the previous semaphore example except that this program uses the UNIX System V semaphores.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define NPROC 5
#define KEY (key_t)7

int sem_id;
int holdsem = 5;
union {
        int val;
        struct semid_ds *buf;
        ushort *array;
} arg;

main()
{
        register i, child;

        /*
         * create the semaphore
         */
        if ((sem_id = semget(KEY, 1, IPC_CREAT |0777)) < 0)
                err ( "semget" );
        /*
         * set the number of resources it is controlling
         */
        arg.val = 1;
        if ( semctl ( sem_id, 0, SETVAL, arg) < 0 ) {
                err ( "semctl setval");
        }
        for ( i = 1; i < NPROC; ++ i ) {
                if ( (child = fork() ) < 0 )
                        err ("No fork");
                else if ( child == 0 ) {
                        if ( ( sem_id = semget ( KEY, 1, 0777) ) < 0 )
                                err ( "childsemget");
                        doit(i);
                        exit(0);
                }
        }
        doit(0);
        for (i = 1; i < NPROC; ++i)
                while (wait ( (int *)0 ) < 0)
                        ;
        /*
         * remove the semaphore
         */
        if ( semctl ( sem_id, 0, IPC_RMID ) < 0 )
                err ( "semctl removal");
}

doit(id)
int id;
{
        struct sembuf sops[1];

        sops[0].sem_num = 0;        /* there is only one semaphore */
        sops[0].sem_flg = 0; /* wait until semaphore available */
        while (holdsem--) {
                /*
                 * the value of the member sem_op
                 * determines the action of semop()
                 */
                sops[0].sem_op = -1;        /* request the semaphore */
                if ( semop ( sem_id, sops, 1) != 0 )
                        err ( "semop request");
                printf("%d\n", id);
                sleep(1);
                sops[0].sem_op = 1; /* release the semaphore */
                if ( semop (sem_id, sops, 1 ) < 0 )
                        err ( "semop release" );
        }
}

err(s)
char*s;
{
        perror(s);
        exit(1);
}
```

9.5 Using Shared Memory

Shared memory is a method by which one process shares its allocated data space with another. Shared memory allows processes to pool information in a central location and directly access that information without the burden of creating pipes or temporary files.

The standard C library and the XENIX library provide several functions to access and control shared memory. These functions 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**(S), **sdfree**(S), **sdenter**(S), **sdgetv**(S), and **sdwait**(S). Programs using these operations must be linked with the XENIX library, using the **-lx** option.

The UNIX System V shared memory operations are compatible with AT&T UNIX System V. The system calls for manipulating UNIX shared memory are **shmop**(S), **shmctl**(S), and **shmget**(S).

It is not possible for you to use both XENIX and UNIX shared memory in a compatible fashion. The operations mentioned are valid only for one type of shared memory, and cannot be mixed with the other type. This section mainly describes the use of the XENIX shared memory operations in detail since these system calls are unique to the XENIX operating system. A discussion of UNIX System V shared memory and a program example is included.

XENIX shared memory uses a semaphore type mechanism to control access to the segment. The shared memory is part of the near data segment. Each process attached to a XENIX shared memory segment maintains a separate copy of the data. It is slower than the UNIX System V shared memory, because after every **sdleave** the kernel copies the updated data to each process that is attached to the shared memory.

Since XENIX shared memory is kept in the near data segment for small and medium model programs, the heap space is reduced. There is less space available for the process to dynamically allocate.

Access to UNIX System V shared memory must be controlled by the program. It is faster than XENIX shared memory because only a single copy of the data is kept; all processes attached to the shared memory access the same physical memory. It does not impact the amount of heap space available to small and medium model 8086/80286 programs because the shared memory is in a separate far segment. Use of UNIX System V shared memory can allow small and medium model 8086/80286 programs to have more than 64K of data.

If there is not enough free memory available on the system, UNIX System V shared memory operations do not cause swapping of other processes in order to get the needed memory. Because of this, the performance of programs using UNIX System V shared memory varies depending on the system load.

There are five XENIX shared memory functions: **sdget**, **sdfree**, **sdenter**, **sdgetv**, and **sdwait**.

The **sdget** function creates and/or adds a shared memory segment to a given process's data space. To access a segment, a process must signal its intention with the **sdenter** function. Once a segment has completed its access, it can signal that it is finished using the segment with the **sdleave** function. The **sdfree** function is used to remove a segment from a process's data space. The **sdgetv** and **sdwaitv** functions are used to synchronize processes when several are accessing the segment at the same time.

To use the shared data functions, you must add the lines

```
#include <sys/types.h>
#include <sd.h>
```

at the beginning of the program. The *sd.h* file contains definitions for the manifest constants and other macros used by the functions.

9.5.1 Creating a Shared Data Segment

The **sdget** function creates a shared data segment for the current process and attaches the segment to the process's data space. The function call has the form:

```
sdget (path, flag, size, mode);
```

where *path* is a character pointer to a valid pathname, *flag* is an integer value which defines how the segment should be created, *size* is a long integer value which defines the size in bytes of the segment to be created, and *mode* is an integer value which defines the access permissions to be given to the segment. The *flag* may be a combination of SD_CREAT for creating the segment, SD_RDONLY for attaching the segment for reading only, or SD_WRITE for attaching the segment for reading and writing. You may also use SD_UNLOCK for allowing simultaneous access by multiple processes. The values can be combined by logically ORing them. The function returns the address of the segment if it has been successful. Otherwise, the function returns -1.

The function is typically used by just one process to create a segment that it will share with several other processes. For example, in the following fragment, **sdget** is used to create a segment and attach it for reading and writing. The address of the new segment is assigned to *shared*.

```
char *shared;

shared = sdget("/tmp/share",
                    SD_CREAT |SD_WRITE, 512L, 0777);
```

When the segment is created, the size "512" 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 that the SD_UNLOCK flag used on systems without hardware support for shared data may severely degrade the execution performance of the program.

9.5.2 Attaching a Shared Data Segment

The **sdget** function can also be used to attach an existing shared data segment to a process's data space. In this case, the function call has the form

```
        sdget(path, flag);
```

where *path* is a character pointer to the pathname of a shared data segment created by some other process, and *flag* 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.

The function can be used to attach any shared data segment a process may wish to access. For example, in the following fragment, the program 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*.

```
        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 data segment without the necessary permissions. The segment permissions are defined when the segment is created.

9.5.3 Entering a Shared Data Segment

The **sdenter** signals a process's intention to access the contents of a shared data segment. A process cannot access the contents of the segment unless it enters the segment. The function call has the form:

```
        sdenter (addr, flag);
```

where *addr* is a character pointer to the segment to be accessed, and *flag* is an integer value which defines how the segment is to be acces ed. 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 may also be combined by logically ORing them. 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 the SD_UNLOCK flag and another process is currently accessing it.

Once a process has entered a segment, it can examine and modify the contents of the segment. For example, in the following fragment, the program uses **sdenter** to enter the segment for reading and writing, then sets the first value in the segment to 0 if it is equal to 255.

```
char *share;

share = sdget( "/tmp/share", SD_WRITE );

sdenter(share, SD_WRITE);
if ( share[0] == 255 )
        share[0] = 0;
```

In general, it is unwise to stay in a shared data 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 data 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 data segment that cannot be shared simultaneously, the program must not call the **fork**(S) function when it is also accessing that segment.

9.5.4 Leaving a Shared Data Segment

The **sdleave** function signals a process's intention to leave a shared data segment after reading or modifying its contents. The function call has the form:

sdleave (*addr*);

where *addr* is a character 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.

The function should be used after each access of the shared data to terminate the access. If the segment's lock flag is set, the function must be used after each access to allow other processes to access the segment. For example, in the following program fragment, **sdleave** terminates each access to the segment given by "shared".

```
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);
}
```

9.5.5 Getting the Current Version Number

The **sdgetv** function returns the current version number of the given data segment. The function call has the 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.

The function is typically used 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".

```
char *shared;

if (sdgetv(shared) > 10)
        sdenter(shared);
```

In this example, the segment is entered if the current version number of the segment is greater than "10".

9.5.6 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 form:

```
sdwaitv (addr, vnum);
```

where *addr* is a character pointer to the desired segment, and *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.

The function is typically used to synchronize the actions of two separate processes. For example, in the following program fragment the program waits while the program corresponding to the version number "vnum" performs its operations in the segment.

```
char *share;
int vnum;

vnum = sdgetv( share );
if( sdwaitv(share, vnum) == -1)
        fprintf(stderr, "Cannot find segment\n");
else
        sdenter( share );
```

If an error occurs while waiting, an error message is printed.

9.5.7 Freeing a Shared Data Segment

The **sdfree** function detaches the current process from the given shared data segment. The function call has the form:

```
sdfree(addr);
```

where *addr* is a character pointer to the segment to be set free. If the segment is freed, the function returns the integer value 0. 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.

9.5.8 Program Example

This section shows how to use the shared data functions to share a single data segment between two processes. The following program attaches a data segment named */tmp/share* and then uses it to transfer information to between the child and parent processes.

```
#include <sys/types.h>
#include <sd.h>

#define SIZE 30

main()
{
        char *share, message[SIZE];
        int i, vnum;

                        share = sdget("/tmp/share",
                        SD_CREAT | SD_WRITE | SD_UNLOCK,
                        (long)SIZE, 0777);

        if (fork() == 0){
                for (i=0; i < 4; i++) {
                        sdenter(share, SD_WRITE);
                        strcpy(message, share);
                        strcpy(share, "Child leaving message");
                        vnum = sdgetv(share);
                        sdleave(share);
                        sdwaitv(share, vnum+1);
                        printf("Child: %d - %s\n", i, message);
                }
                sdenter(share, SD_WRITE);
                strcpy(message, share);
                strcpy(share, "Child leaving last message");
                sdleave(share);
                printf("Child: %d - %s\n", i, message);
                exit(0);
        }
```

```
                for (i=0; i < 5; i++) {
                        sdenter(share, SD_WRITE);
                        strcpy(message, share);
                        strcpy(share, "Parent leaving message");
                        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 data segment created by the parent process. Each process accesses the segment 5 times. During the access, a process copies the current contents of the segment to the variable *message* 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 argument to **sdwaitv** is "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 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.

9.5.9 UNIX System V Shared Memory

The UNIX System V shared memory operations are compatible with AT&T UNIX System V. The system calls for manipulating shared memory are **shmctl**(S), **shmget**(S), and **shmop**(S).

To use the shared memory functions, you must add the lines:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

at the beginning of the program. The *sys/ipc.h* and *sys/shm.h* files contain definitions for the manifest constants and other macros used by the functions. Be sure to use the **-Me** option to cc(CP) when compiling 8086 or 80286 programs containing these #include files to enable the use of the **far** keyword.

The full extent of UNIX System V shared memory operations are not described here in detail. This section briefly describes some of the issues to be aware of when using shared memory.

Each process gets a shared memory *id* by calling *shmget* with the proper key. The key field is the means for determining which memory segment is being shared. Each process that needs to attach to the same memory segment must use the same key. The process that creates the shared memory must also use the **IPC_CREAT** command when calling **shmget**. All other processes that use the memory also call **shmget** but without using **IPC_CREAT**.

Each process attaches to the shared memory segment using **shmat**. The shared memory address, *shmaddr*, must be 0 on systems with segmented architecture. **shmat** returns a far pointer on 8086/80286 processors. This is important to remember when comparing quantities. For example:

```
#define FNULL          (char far *) 0
#define FAILURE        (char far *) -1
char far *fp;

if ((fp = shmat(id, FNULL, 0)) == FAILURE)
        ...
```

Note that the comparisons use far pointers, and that they check for *equality* with -1. Checks for less than zero (< 0) will not work as expected.

Shared memory operations do not clean up after themselves. The programmer must be careful to removed shared memory segments when they are no longer required. The last process to detach from the shared memory segment should use a **shmctl** call with the **IPC_RMID** command. This (implicitly) detaches the process and removes the shared memory segment. If the last process detaches from the shared memory segment and does not remove it, the shared memory remains inaccessible until a process calls **shmget** using the same key and attaches to the shared memory.

9.5.10 Program Example

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <signal.h>

#define KEY (key_t) 60
```

```
#define FNULL      (char far *) 0
#define FAILURE (char far *) -1
#define FALSE 0
#define TRUE 1

int shmid;
char far *base, far *shmat();          /*both fars are necessary
                                          on 8086 or 80286 systems */
int clean();

main()
{
        int created;
        char buf[50];

        signal(SIGINT, clean);
        created = TRUE;
        if((shmid = shmget(KEY, 50, IPC_CREAT |0666)) < 0){
                created = FALSE;
                if ((shmid = shmget(KEY, 50, 0)) < 0) {
                        perror("shmget failed");
                        exit(1);
                }
        }
        if ((base = shmat(shmid, FNULL, 0)) == FAILURE) {
                perror("shmat failed");
                exit(1);
        }
        /*
         * if we created the shared memory segment load it with data.
         * in a real situation the data would come from a file.
         */
        if (created)
                fn_strcpy(base, "acfhijknpz");
        while (gets(buf) != NULL)
                if (f_strchr(base, buf[0]))
                        printf("found\n");
                else
                        printf("not found\n");
        clean();
}
```

```
/*
 * same as strchr(S), but takes a far pointer
 */
int
f_strchr(s, c)
char far *s;
char c;
{
        char d;

        while(d = *s++)
                if(c == d)
                        return(TRUE);
                else if (c < d)
                        return(FALSE);
        return (FALSE);
}

/*
 * copy a near string to a far string
 */
fn_strcpy(s, t)
char far *s;
char *t;
{
        while (*s++ = *t++)
                ;
}

/*
 * detach and remove the shared memory segment
 */
clean()
{
        struct shmid_ds shmds;

        shmdt(base);
        shmctl(shmid, IPC_RMID, &shmds);
        exit(0);
}
```

This program illustrates getting a shared memory segment in which a database of information is being stored. Different processes share the database. The process that creates the database loads it with data. Each process reads characters from the standard input and checks the shared memory to see if the character read is in the database. This program can be executed several times simultaneously. Only one shared memory segment is created and the same segment is referenced by each process. Since the shared memory segment is not in the same segment as the current process, they must be accessed using far pointers.

It is necessary to use special handling to copy data to far pointers. The standard library routines work only with large model programs because then all data is far. In small and middle model programs, programmers must write their own functions to copy data between near and far segments. Casting a near pointer to a far pointer does not achieve the same results.

9.6 Message Queues

Message queues are another way in which one process can communicate with another. They are especially useful when different types of messages are being passed between processes, and the action required is dependent on the type of message being passed.

The standard C library provides the following functions to access and control message queues: **msgctl**(S), **msgget**(S), and **msgop**(S). These operations are all UNIX System V operations which are compatible with AT&T UNIX System V.

To use these message queue functions, you must add the lines:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

at the beginning of the program. The *sys/ipc.h* and *sys/msg.h* files contain definitions for the manifest constants and other macros used by the functions. Be sure to use the **-Me** option on **cc**(CP) when compiling programs containing these #include files to enable the use of the **near** keyword.

9.6.1 Program Example

Program example for process 1.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/signal.h>
#include <ctype.h>

#define KEY (key_t)60

int msgid;

main()
{
        int i,n;
        struct {
```

```
        long mtype;
        char mtext[40]; /* max of 40 chars per msg */
}buf;
char s[100];
int clean();

signal(SIGINT, clean);
/*
 * Create the msg queue.
 * Use key #60. Everyone else will use this too.
 * The key can be any value that everyone agrees to.
 */
msgid = msgget(KEY, IPC_CREAT |0666);
if (msgid < 0) {
        perror("1 msgget failed");
        exit(1);
}
/*
 * This is the main program.
 * We will send messages to two processes.
 * The messages will consist of integers.
 * One of the processes will multiply the integer by3.
 * The other will add 6.
 * This program receives the answers back
 * and prints the results.
 */
while (gets(s) != NULL) {
        strcpy(buf.mtext, s+1);
        /*
         * Send it to one of the two processes.
         * The process that multiplies
         * integers by3 will have a 'type' of 3.
         * The other will be type 6.
         * We have a type of 1.
         *
         * A letter and string of digits are read
         * from standard input. If the letter is
         * uppercase send the digits to 3 else send
         * them to 6.
         */
        buf.mtype = isupper(s[0]) ? 3 : 6;
        if (msgsnd(msgid, &buf, strlen(buf.mtext),
                    ~IPC_NOWAIT) < 0){
                perror("1 msgsnd failed");
                exit(1);
        }
        /*
         * Now get the answer back and print it out.
         */
        if (msgrcv(msgid, &buf, strlen(buf.mtext), (long) 1,
                                IPC_NOWAIT) < 0) {
                        perror("1 msgrcv failed");
                        exit(1);
                }
                printf("%s\n", buf.mtext);
        }
        clean();
}

/*
 * All communication is done, remove the message queue.
 */
clean()
{
        if (msgctl(msgid, IPC_RMID,
                        (struct msquid *) NULL) < 0) {
                perror("1 msgctl failed");
                exit(1);
        }
        printf("msgqueue removed\n");
        exit(0);
}
```

Program example for process 6.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>

#define KEY (key_t) 60

/*
 * We are process #6.
 * Read messages of type 6 from msg queue #60,
 * take the integer in the message,
 * add 6 to it and send it back to process #1.
 */
main()
{
        int i, msgid, val;
        struct {
                long mtype;
                char mtext[40];
        } buf;

        /*
         * Get msg queue #60.
         * the 0 argument means no create, r/w perms
```

```
          */
          msgid = msgget(KEY, 0);
          if (msgid < 0) {
                perror("6 msgget failed");
                exit(1);
          }
          /*
           * Read numbers from the msgqueue, add 6 to them,
           * and send them back to process #1.
           */
          while (msgrcv(msgid, &buf, 40, (long) 6,
                              ~IPC_NOWAIT) >= 0) {
                val = atoi(buf.mtext);            /* extract integer */
                sprintf(buf.mtext, "%d + 6 = %d", val, val+6);
                buf.mtype = 1;
                if (msgsnd(msgid, &buf, 40, ~IPC_NOWAIT) < 0) {
                      perror("6 msgsnd failed");
                      exit(1);
                }
          }
          if (errno != EIDRM)
                perror("6 msgrcv failed");
    }
```

This example consists of three programs, two of which are given. The third is very similar to the second. All three processes execute at the same time; process 1 needs to start first because it creates the message queue.

The first program creates a message queue, and sends messages of types 3 or 6, which contain integers, to the other two processes. Process 6 receives messages of type 6, extracts the integer from the message, adds 6 to the integer, and sends the sum back to process 1. Process 3 acts in similar fashion, except that it receives type 3 messages and multiplies the integer by 3 before sending the product back to process 1.

Process 1 uses processes 3 and 6 to perform tasks. It can continue with the main body of its function while subprocessing is handled by the other processes.

Note that the *msg.h* file contains the following declaration of *struct msgbuf*:

```
          struct msgbuf {
                long mtype;          /* message type */
                char mtext[1];       /* message text */
          };
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/msg.h>
```

This is only a template or model. Declaring a message buffer with a type of *struct msgbuf* will only allow messages of length 1 to be sent. The preferred method is as illustrated in the example where a buffer is specifically declared with a maximum size specified. Note that the

```
        char mtext[40];
```

cannot be replaced with

```
        char *mtext;
```

Appendix A

XENIX to DOS: A Cross Development System

A.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.

A.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 pathname 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.

A.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 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.

A.3.1 DOS Floating Point Flags

The **-FP***n* 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. (*/usr/lib/dos/*[SML]*dlibcfa.a*) 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. (*/usr/lib/dos/em.a*) 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. (*/usr/lib/dos/87.a*) A coprocessor must be installed to run programs compiled with this flag.
-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.

A.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* and the *Macro Assembler Reference*. 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*.

A.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 *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.

A.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.

A.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.

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 **doscp** 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 *Reference* and Chapter 3 "Using XENIX and DOS On the Same Disk" in the XENIX *Installation Guide*.

A.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.

A.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.

A.9.1 Common Routines

The following is a list of the common routines for DOS and XENIX.

abort*	**ctime**	**fprintf**	**isascii**	**modf**	**sscanf**	**_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**	
atol	**exp**	**ftime***	**localtime**	**rand**	**strncpy**	
bessel†,††	**fabs**	**fwrite***	**locking***	**read***	**strpbrk**	
bsearch	**fclose**	**gcvt**	**log†**	**realloc**	**strrchr**	
cabs	**fcvt**	**getc**	**log10†**	**rewind**	**strspn**	
calloc	**fdopen**	**getchar**	**longjmp**	**sbrk**	**strtok**	
ceil	**feof**	**getcwd**	**lseek***	**scanf**	**swab**	
chdir*	**ferror**	**getenv**	**malloc**	**setbuf**	**system***	
chmod*	**fflush**	**getpid***	**matherr**	**setjmp**	**tan†**	
chsize	**fgetc**	**gets**	**memccpy**	**signal***	**tanh†**	
clearerr	**fgets**	**getw**	**memchr**	**sin†**	**time**	
close	**fileno**	**gmtime**	**memcmp**	**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**.

A.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 A.9.1.

A.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	**fputchar**	**labs**	**spawnl**	**strupr**
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.

A.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.

A.10.1 Common Variables

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.

A.10.2 Common Variables

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.

A.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

A.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 A.11.1 and A.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 A.11.3.

A.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:

assert.h	*setjmp.h*	*sys/stat.h*
ctype.h	*signal.h*	*sys/timeb.h*
errno.h	*stdio.h*	*sys/types.h*
fcntl.h	*time.h*	
math.h	*sys/locking.h*	

A.11.2 Common Include Files

The XENIX-compatible include files listed in section A.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

A.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.

conio.h	*io.h*	*stdlib.h*
direct.h	*process.h*	*sys/utime.h*
dos.h	*share.h*	*v2tov3.h*

A.12 Differences Between Common Routines

Sections A.12.1 through A.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*.

A.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 coredump occurs on DOS.

A.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.

A.12.3 chdir

In case of error, only the **ENOENT** value may be returned for **errno** on DOS.

A.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.

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.

A.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.

A.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 **EMFILE** value may be used; under DOS, the file must be opened to determine whether it is executable.

A.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 in the XENIX *Reference*. **fopen** and **freopen** routines to obtain more information on the default mode setting.

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.

A.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.

A.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**.

A.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.
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.

A.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.

A.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 an 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 *Reference* for details on the TZ variable.

A.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.

A.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.

A.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.

A.12.16 lseek

In case of error, only the **EBADF** and **EINVAL** values may be returned for **errno** on DOS.

A.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.

A.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.

A.12.19 signal

The DOS version of the **signal** routine can only handle the **SIGINT** signal. In DOS, **SIGINT** is defined to be INT23H (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**.

A.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
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_ino	Not used.
st_dev	Drive number of the disk containing the file.
st_rdev	Drive number of the disk containing the file.
st_nlink	Always 1.
st_uid	Not used.
st_gid	Not used.
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 **ENOENT** value may be returned for **errno** on DOS.

A.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.

A.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.)

A.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.

A.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.

A.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.

In case of error, only the **EBADF** and **ENOSPC** values may be returned for **errno** on DOS.

A.13 Differences in Definitions

Many of the special definitions given in *intro*(S) in the XENIX *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.

Appendix B
System Error Values

B.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 the XENIX *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. Thus, you should test for errors and print error mesages immediately after a function call to obtain accurate results.

The include file, *errno.h*, contains the definitions of the **errno** values.

B.2 errno Values

The following list shows the **errno** values used in the XENIX environment, the system error message corresponding to each value, and a brief description of the circumstances that cause each error.

Value	**Message**	**Description**
1 EPERM	Not owner.	Indicates an attempt to modify a file when the user is not the owner or the super-user. This message is also returned for attempts to do things only allowed to a super-user.
2 ENOENT	No such file or directory.	Occurs when a specified file does not exist, or when a directory specified in a pathname does not exist.
3 ESRCH	No such process.	The process specified by *pid* in *kill* or *ptrace* cannot be found.

4 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.
5 EIO	I/O error.	A physical I/O error. May occur on a call following the one to which it actually applies.
6 ENXIO	No such device or address.	I/O on a special file refers to a subdevice which doesn't exist, or 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.
7 E2BIG	Arg. list too long.	An argument list longer than 5,120 bytes is presented to a member of the *exec* family.
8 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)).
9 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).
10 ECHILD	No child processes.	A **wait** was executed by a process which had un-*wait*ed-for child processes.
11 EAGAIN	No more processes.	A **fork** failed because the process table is full, or the user has the maximum number of processes.

12 ENOMEM	Not enough space.	During an **exec** or **sbrk**, a request is made for more space than is available. Space available 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**.
13 EACCES	Permission denied.	Attempt to access a file in a way denied by the protection system.
14 EFAULT	Bad address.	The system encountered a hardware fault while attempting to use an argument from a system call.
15 ENOTBLK	Block device required.	A non-block file was specified where a block device was required, e.g., **mount**.
16 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, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled.
17 EEXIST	File exists.	An existing file was specified in an inappropriate context, e.g., *link*.
18 EXDEV	Cross-device link.	A *link* to a file on another device was attempted.
19 ENODEV	No such device.	An attempt was made to apply an inappropriate system call to a device, e.g., a write-only device.
20 ENOTDIR	Not a directory.	A non-directory was specified where a directory was required (e.g., in a path prefix or as an argument to **chdir** (S)).
21 EISDIR	Is a directory.	An attempt to write on a directory.

22 EINVAL	Invalid argument.	An invalid argument (i.e. mentioning an undefined signal to *signal*) was passed to a routine. Also set by the math functions described in the (S) section of the XENIX *Reference*.
23 ENFILE	File table overflow.	The system's table of open files is full, and (temporarily) no more **opens** can be be accepted.
24 EMFILE	Too many open files.	No process may have more than 60 file descriptors open at a time.
25 ENOTTY	Not a typewriter.	Not a typewriter.
26 ETXTBSY	Text file busy.	An attempt to execute a pure-procedure program which is currently open for writing (or reading). Also, an attempt to open a pure-procedure program that is being executed for writing.
27 EFBIG	File too large.	The size of a file exceeded the maximum file size (1,082,201,088 bytes) or ULIMIT; see **ulimit** (S).
28 ENOSPC	No space left on device.	During a **write** to an ordinary file, there is no free space left on the device.
29 ESPIPE	Illegal seek.	An **lseek** was issued to a pipe.
30 EROFS	Read-only file system.	An attempt to modify a file or directory was made on a device mounted read-only.
31 EMLINK	Too many links.	An attempt was made to make more than the maximum number of links (1000) to a file.
32 EPIPE	Broken pipe.	A write 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.

33 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.
34 ERANGE	Math result not representable.	The value of a function in the math package is not representable within machine precision.
35 EUCLEAN	File system needs cleaning.	An attempt was made to **mount** (S) a file system whose super-block is not flagged clean.
36 EDEADLK	Would deadlock.	A process' attempt to lock a file region would cause a deadlock between processes vying for control of that region. Also, sometimes shown as EDEADLOCK.
37 ENOTNAM	Not a name file.	A **creatsem** (S), **opensem** (S), **waitsem** (S), **sigsem** (S), was issued using an invalid semaphore identifier.
38 ENAVAIL	Not available.	An **opensem** (S), **waitsem** (S), or **sigsem** (S) was issued to a semaphore that has not been initialized by a call to **creatsem** (S). A **sigsem** was issued to a semaphore out of sequence (i.e., before the process has issued the corresponding **waitsem** to the semaphore). An **nbwaitsem** (S) 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 (i.e., without using a **waitsem** on the semaphore).
39 EISNAM	A name file.	A name file (semaphore, shared data, etc.) was specified when not expected.

B.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). **Errno** will be set if the calculation to be performed is inappropriate or would result in an overflow.

The following is a list of the math library functions, and the values that they return:

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 non-positive and *y* is not an integer, or when *x*, *y* are both zero.
log, log10	EDOM	Returns a huge negative value when the argument given is non-positive.
sqrt	EDOM	Returns 0 when argument is negative.
hypot, cabs	-	Both return: **sqrt(x*x + y*y)** Overflows are precluded.
sinh, cosh, tanh	-	Both sinh and cosh will return a huge value of appropriate sign when the correct value would overflow.

Index

XENIX® System V

Development System

Macro Assembler User's Guide

This document was typeset with an IMAGEN® 8/300 Laser Printer.

SCO Document Number: XG-5-1-87-4.0

Contents

Chapter 1
Introduction

1.1 Overview

This guide describes the usage, syntax, and structure of the XENIX Macro Assembler, MASM. MASM is an assembler for the Intel® 8086, 80186, 80286, or 80386 families of microprocessors. It can assemble the instructions of the 8086, 8088, 80186, 80286, and 80386 microprocessors, and the 8087 and 80287 floating-point coprocessors. It has a set of powerful assembly-language directives that gives you complete control of the segmented architecture of the 8086, 80186, 80286, and 80386 microprocessors. MASM instruction syntax allows a wide variety of operand data types, including integers, strings, packed decimals, floating-point numbers, structures, and records.

The assembler produces 8086, 8088, 80186, 80286, and 80386 relocatable object modules from assembly-language source files. The relocatable object modules can be linked to create executable programs.

MASM is a macro assembler. It has a full set of macro directives that lets you create and use macros in a source file. The directives instruct MASM to repeat common blocks of statements, or replace macro names with the blocks of statements they represent. MASM also has conditional directives that provide for selective exclusion of portions of a source file from assembly, or inclusion of additional program statements by simply defining a symbol.

MASM performs strict syntax checking of all instruction statements, including strong typing for memory operands, and detects questionable operand usage that could lead to errors or unwanted results.

MASM produces relocatable object modules that can be linked with other high-level-language object modules. Programs can be constructed by combining MASM relocatable object modules with object modules created by C, Pascal, FORTRAN, or other high-level-language compilers.

1.2 Getting Started

Before you start developing assembly-language programs, you need to verify the following:

- The current operating system is the XENIX System V/386.
- The MASM executable files are located in the /usr/bin directory.
- You know how to use the 8086, 80286 and 80386 instruction sets.
- Your text editor creates ASCII text files.

If the current operating system is not the XENIX® System V/386, determine the operating-system version and use the corresponding MASM manuals.

If the MASM executable files are not located in the /usr/bin directory, ask your system administrator for their location.

To create assembly-language programs, you need to know how to use the 8086, 80286, and 80386 instruction sets. The directives, operands, operators, and expressions of MASM are explained in this guide. However, this guide does not explain how to use the instruction sets.

To assemble assembly-language programs, the source file must be in ASCII format. If your text editor does not produce ASCII files, switch to an editor that produces ASCII files.

1.3 Notational Conventions

This guide uses the following notational conventions to define the assembly-language syntax:

Roman	Indicates command, keyword, directive, or parameter names that must be typed as shown. In most cases, uppercase roman represents keywords and directives, and lowercase roman represents commands and parameters. Uppercase and lowercase letters can be freely mixed in some cases.
Bold	Indicates command-line options and arguments used to call the assembler and assembler options.
Italics	Indicates placeholders or parameters; i.e., a name that you must replace with the value or file name required by the program. Also indicates file names and path names in text.
Ellipsis dots...	Indicate that you can repeat the preceding item any number of times.
Commas ,,,	Indicate that you can repeat the preceding item any number of times, as long as you separate the items with a comma.
[Brackets]	Indicate that the enclosed item is optional. If you do not use the optional item, the program selects the default action.
Vertical bar |	Indicates that only one of the separated items can be used. You must make a choice between the items.
"Quotation marks"	Indicate text from a source-code example.
I/O	Indicates source code created with a text editor.

Chapter 2
Command-Line Options

2.1 Introduction

This chapter describes how to use the MASM command and command-line options to create relocatable object files. The MASM command and command-line options are briefly described in the next section. Each following section describes a specific command-line option.

2.2 Assembling Source Files

The MASM command line initiates the assembly of assembly-language source files. To assemble one or more source files, type the MASM command, the desired command-line options, if any, and the name of the file(s) you want to assemble.

The MASM command line has the form:

masm [*options*] *filename*

The *options* can be any combination of MASM options described in the following sections.

The *filename* is the name of the source file to be assembled. The *filename* must have the *.s* extension. If the source file is successfully assembled, then a relocatable object file is created with the *.o* extension and the same file name as the source file.

For a complete list and brief description of MASM command-line options, see **masm**(CP) in the XENIX *Reference*.

2.3 Outputting Segments in Alphabetical Order: -a

Syntax

-a

The **-a** option directs MASM to place the assembled segments in alphabetical order before copying them to the object file. If this option is not given, MASM copies the segments in the order encountered in the source file.

The following example creates an object file named *filename.s* whose segments are arranged in alphabetical order. Therefore, if the source file *filename.o* contains definitions for the segments DATA, CODE, and MEMORY, the assembled segments in the object file have the order CODE, DATA, and MEMORY.

Example

```
masm -a filename.s          ; assembles to filename.o
```

2.4 Specifying the I/O Buffer Size: -b

Syntax

-b*num*

The **-b** option directs MASM to define the I/O buffer size for source, include, and object files, but not cross-reference or listing files. The size of the buffer is defined in 1K (Kilobyte) increments by the *num* argument and can be any integer value in the range 1–64. If this option is not placed on the command line, the default buffer size of 64K is defined.

The following example directs MASM to define an I/O buffer size of 45K.

Example

```
masm -b45 filename.s
```

2.5 Outputting Cross-Reference Data: -c, -C

Syntax

-c
-C

The **-c** and **-C** options print cross-reference data for each assembled file and for each set of assembled files, respectively. If either the **-c** or **-C** option is used, then the cross-reference data for each assembled file is output to a file with the *.crf* extension and the same file name as the assembled file.

The following two examples direct MASM to output cross-reference data for each specified file.

Examples

```
masm -c filename.s
masm -C filename1.s filename2.s filename3.s
```

2.6 Creating a Pass 1 Listing: -d

Syntax

-d

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

The **-d** option does not create a Pass 1 listing unless you also direct MASM to create an assembly listing. It does direct MASM to display error messages for both Pass 1 and Pass 2 of the assembly, even if no assembly listing is created.

The following example directs MASM to create a Pass 1 listing for the source file *filename.s* into the file *filename.lst* .

Example

```
masm -d -l filename.s      ; creates filename.lst
```

2.7 Defining Symbols: -D

Syntax

-D*symbol*

The **-D** option directs MASM to define the *symbol* argument appended to the **-D** option as a text macro with a null value. (See the *Macro Assembler Reference*, "EQU Directive," in the chapter "Types and Declarations" for a discussion of text macros.) The *symbol* will be defined with the case in effect at that point in the command line. Any number of **-D** options can be used. The defined symbol can be tested with the IFDEF and IFNDEF directives during the assembly.

The following example directs MASM to define *symbol* as a null text macro. The default conversion to uppercase occurs in this example.

Example

```
masm –Dsymbol filename.s
```

2.8 Creating Floating-Point Emulator Code: -e

Syntax

-e

The -e option directs MASM to generate floating-point instruction codes that can be "fixed up" to software interrupts at link time.

This is the default option under XENIX. If an 80287 is present, the XENIX system changes the software interrupts into real 80287 instructions. If the 80287 chip is not present, a software emulator in the XENIX system is used to process the software interrupts as if the chip were actually present. The emulator does not handle all valid 80287 instructions. Unemulated instructions will give a SIGILL signal.

The following example directs MASM to create emulation code for any floating-point instructions it finds in the program.

Example

```
masm –e filename.s
```

2.9 Controlling Include-File Path Names: -I

Syntax

-I*pathname*

The -I option directs MASM to use the specified *pathname* argument as a prefix to the file names given in the INCLUDE directives in an assembly-language program. Up to 10 -I options can be specified on the command line to force searching of the given directories in a specific order.

The following example forces the INCLUDE directives to search */usr/include*, then the current directory, for the given file name.

Example

```
masm -I /usr/include -I. filename.s
```

2.10 Producing an Assembled-Source Listing: -l

Syntax

-l [*filename*]

The -l option directs MASM to generate a listing file to the standard output file, which is usually the console device. If the -l option has a *filename* argument appended to it, then the listing is written to the file *filename* rather than the default listing file (whose name is the same as that of the first input file, except that it has the extension *.lst*).

The following example directs MASM to generate a source listing into the file *listfilename.lst* .

Example

```
masm -l listfilename.s              ; creates listfilename.lst
```

2.11 Preserving Lowercase Internal Names: -Ml

Syntax

-Ml

The -Ml option directs MASM to preserve lowercase letters in label, variable, and symbol names. When this option is given, names that have the same spelling but use different letter case are considered different. For example, with the -Ml option, "DATA" and "data" are different. Under XENIX this is the default case mapping option.

The -Ml option is typically used when a source file is to be linked with object modules created by a case-sensitive compiler.

The following example directs MASM to preserve lowercase letters in any names defined in the source file *filename.s*.

Example

```
masm -Ml filename.s
```

2.12 Converting Names to Uppercase: -Mu

Syntax

-Mu

The **-Mu** option directs MASM to convert all lowercase letters in all symbols to uppercase.

The following example directs MASM to convert lowercase letters in all symbols defined in the source file *filename.s*.

Example

```
masm -Mu filename.s
```

2.13 Preserving Lowercase Public and External Names: -Mx

Syntax

-Mx

The **-Mx** option directs MASM to preserve lowercase letters in public and external names only when copying these names to the object file. For all other purposes, MASM converts the lowercase letters to uppercase.

Public and external names are any label, variable, or symbol names that have been defined using the EXTRN or PUBLIC directive. Since MASM converts the letters to uppercase for assembly, these names must have unique spellings. That is, the names "DATA" and "data" are considered the same.

The **-Mx** option is used to ensure that the names of routines or variables copied to the object module have the correct spelling. The option is used with any source file that is to be linked with object modules created by a case-sensitive compiler, and is particularly useful for transporting assembler files from MS-DOS® to XENIX when working with C.

The following example directs MASM to preserve lowercase letters in any public or external names defined in the source file *filename.s*.

Example

```
masm -Mx filename.s
```

2.14 Suppressing Symbol Table Information: -n

Syntax

-n

The **-n** option directs MASM to suppress information about the symbols used in the assembled program. This option must be used in conjunction with the **-l** option.

The following example directs MASM to generate a listing file without any symbol information in the file *filename.lst*.

Example

```
masm -l -n filename.s
```

2.15 Outputting Object-Code Files: -o, -O

Syntax

-o*objectfile*
-O*objectfile*

The **-o** and **-O** options direct MASM to generate an object-code file. If the **-o** or **-O** option has an *objectfile* argument appended to it then object code is written to the file *objectfile* rather than the default file (whose name is the same as that of the first input file, except that it has the extension *.o*). The **-o** option without a file name suppresses the generation of an object file. The **-o** and **-O** options output assembled instructions in octal and binary formats, respectively.

The following two examples direct MASM to generate object code in the file *objectfile.*

Examples

```
masm -oobjectfile filename.s
masm -Oobjectfile filename.s
```

2.16 Checking for Impure Memory References: -p

Syntax

```
-p
```

The **-p** option directs MASM to check for impure memory references. This option ensures that you don't declare any explicit stores into memory via the CS segment overide operator. If you want your code to run in 80286 protected mode, use the **-p** option to avoid errors due to impure memory references. For example, a typical violation might look like the following, and if assembled with the **-p** option an error message would be generated.

```
        .286
code    segment
        assume cs:code
        ...
codewrd         dw ?
        ...
        mov cs:codewrd, <data>
```

The following example directs MASM to check the source file *filename.s* for any impure memory references.

Example

```
masm -p filename.s
```

2.17 Creating Floating-Point Coprocessor Code: -r

Syntax

```
-r
```

The **-r** option directs MASM to generate floating-point instruction code that can be executed by an 8087 or 80287 coprocessor. Programs created using the **-r** option can run only on machines having an 8087 or 80287 coprocessor.

The following example directs MASM to assemble the source file *filename.s* and create actual 8087 or 80287 instruction code for floating-point instructions.

Example

```
masm -r filename.s
```

2.18 Outputting Assembler Statistics: -v

Syntax

```
-v
```

The **-v** option directs MASM to print statistical information about the assembled file to the output listing. The number of source lines, lines assembled, symbols (in addition to the standard statistic of bytes of symbol space available), and warning and fatal error messages are printed to the output listing.

This example directs MASM to print statistical information about the assembled file *filename.s* to the output listing.

Example

```
masm -v filename.s
```

2.19 Outputting Error Messages: -x

Syntax

-x

The -x option directs MASM to print error messages on the standard error channel in addition to the messages generated in the listing file, without displaying the source line in error. If the -l option is given, then the -x option has no effect. By using this option the MASM will assemble faster. Error messages can be completely suppressed by using the -x option which makes assemblies "silent," i.e., they send no output to the standard error channel.

The following example directs MASM to print copies of error messages only, written to the standard error file.

Example

```
masm -x filename.s
```

2.20 Listing False Conditionals: -X

Syntax

-X

The -X option directs MASM to copy to the assembly listing all statements forming the body of an IF directive whose expression (or condition) evaluates to false. If you do not give the -X option in the command line, MASM suppresses all such statements. The -X option lets you display conditionals that do not generate code. This option applies to all IF directives: IF, IFE, IF1, IF2, IFDEF, IFNDEF, IFB, IFNB, IFIDN, and IFDIF.

The -X option behaves like an initial .TFCOND directive in a source file. The .SFCOND and .LFCOND directives supercede the -X option and .TFCOND directive.

The -X option does not affect the assembly listing unless you direct MASM to create an assembly-listing file.

If the source file “filename.s” does not contain a .TFCOND directive, MASM lists all false conditionals found in the source file.

The following example directs MASM to copy all statements forming the body of an IF directive to the assembly-listing file.

Example

```
masm –X –l filename.s
```

Chapter 3
Source-File Listings

3.1 Introduction

MASM creates an assembly listing of your source file whenever you give an assembly-listing option on the MASM command line (see "Assembling Source Files" in the chapter "Command-Line Options" of the Macro Assembler User's Guide. The assembly listing contains a list of the statements in your program and the object code generated for each statement. The listing also lists the names and values of all labels, variables, and symbols in your source file. MASM creates one or more tables for macros, structures, records, segments, groups, and other symbols and places these tables at the end of the assembly listing.

MASM lists symbols only if it encounters any in the program. If there are no symbols in your program for a particular table, the given table is omitted. For example, if you use no macros in your program, you will not see a macro section in the symbol table.

The assembly listing will also contain error messages if any errors occur during assembly. The error message follows the statement that caused the error. At the end of the listing, MASM displays the number of error and warning messages it issued.

The following sections explain the format of the assembly listing and the meanings of special symbols used in the listing.

3.2 Reading Program Code

MASM lists the program code generated from the statements of a source file. Each line has the form:

[*line-number*] *offset* *code* *statement*

The *line-number* is from the first statement in the assembly listing. The line numbers are given only if a cross-reference file is also being created. The *offset* is the offset from the beginning of the current segment to the code. The *code* is the actual instruction code or data generated by MASM for the statement. MASM gives the actual numeric value of the code, if possible. Otherwise, it indicates what action needs to be taken to compute the value. The *statement* is the source statement shown exactly as it appears in the source file, or after processing by a MACRO, IRP, or IRPC directive.

If any errors occur during assembly, the error message will be printed directly below the statement where the error occurred, displaying the source file and line number in addition to the error number and error message.

MASM uses the following special characters to indicate addresses that need to be resolved by the linker or values that were generated in a special way.

Character	Meaning
R	Relocatable address; linker must resolve
E	External address; linker must resolve
----	Segment/group address; linker must resolve
=	EQU or = directive
nn:	Segment override in statement
nn/	REP or LOCK prefix instruction
nn [*xx*]	DUP expression; nn copies of the value *xx*
+	Macro expansion
C	Included line from INCLUDE file

Example

```
XENIX Macro Assembler          Page 1-1    06-25-87

                         extrn go:near

    0000                 data segment public 'data'
                              assume es:data
    0000 0002            s2   dw    2
    0002                 data ends

    0000                 code segment public 'code'
                              assume cs:code
    0000                 start:
    0000 E8 0000 E                  call    go
    0003 26:A1 0000 R         mov   ax, s2
    0007 B4 4C                mov   ah, 4ch
    0009 CD 21                int   21h
    000B                 code ends

                              end
```

3.3 Reading a Macro Table

MASM lists the names and sizes of all macros defined in a source file. The list has two columns, Name and Length.

The Name column lists the names of all macros. The macro names are listed in alphabetical order and are spelled exactly as given in the source file. Names longer than 31 characters are truncated.

The Length column lists the size of the macro in terms of nonblank lines. The macro size is listed in hexadecimal.

Example

```
Name            Length
BIOSCALL        0002
DISPLAY         0005
DOSCALL         0002
KEYBOARD        0003
LOCATE          0003
SCROLL          0004
```

3.4 Reading a Structure and Record Table

MASM lists the names and dimensions of all structures and records in a source file. The table contains two sets of overlapping columns. The Width and # Fields columns list information about the structure or record. The Shift, Width, Mask, and Initial columns list information about the structure or record members.

The Name column lists the names of all structures and records. The names are listed in alphabetical order and are spelled exactly as given in the source file. Names longer than 31 characters are truncated.

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

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

Example

```
        Name            Width # Fields
                        Shift  Width Mask  Initial

PARMLIST                001C  0004
  BUFSIZE               0000
  NAMESIZE              0001
  NAMETEXT              0002
  TERMINATOR                  001B
```

For a record, the Width column lists the size (in bits) of the record. The # Fields column lists the number of fields in the record.

For fields in a record, the Shift count lists the offset (in bits) from the low order bit of the record to the first bit in the field. The Width column lists the number of bits in the field. The Mask column lists the maximum value of the field, expressed in hexadecimal format. The Initial column lists the initial value of the field, if there is one. For each field, the table shows the mask and initial values as if they were placed in the record and all other fields were set to 0.

Example

```
          Name        Width # Fields
                      Shift  Width Mask  Initial

REC0                  0008   0003
          FLD1        0006   0002  00C0  0040
          FLD2        0003   0003  0038  0000
          FLD3        0000   0003  0007  0003
REC1                  000B   0002
          FL1         0003   0008  07F8  0400
          FL2         0000   0003  0007  0002
```

3.5 Reading a Segment and Group Table

MASM lists the names, sizes, and attributes of all segments and groups in a source file. The list has five columns: Name, Size, Align, Combine, and Class.

The Name column lists the names of all segments and groups. The names in the list are given in alphabetical order, except that the names of segments belonging to a group are placed under the group name. Names are spelled exactly as given in the source file. Names longer than 31 characters are truncated.

The Size column lists the size (in bytes) of each segment. Since a group has no size, only the word GROUP is shown. The size, if given, is in hexadecimal format.

The Align column lists the alignment type of the segment. The type can be any of the following:

BYTE
WORD
PARA
PAGE
DWORD
FWORD

If the segment is defined with no explicit alignment type, MASM lists the default alignment for that segment.

The Combine column lists the combine type of the segment. The type can be any one of the following:

NONE
PUBLIC
STACK
MEMORY
COMBINE
COMMON
AT *address*

NONE is given if no explicit combine type is defined for the segment.

The Class column lists the class name of the segment. The name is spelled exactly as given in the source file. If no name is given, none is shown.

Example

Name	**Size**	**Align**	**Combine**	**Class**
AAAXQQ	0000	WORD	NONE	'CODE'
DATA	0024	WORD	PUBLIC	'DATA'
STACK	0014	WORD	STACK	'STACK'
CONST	0000	WORD	PUBLIC	'CONST'
HEAP	0000	WORD	PUBLIC	'MEMORY'
MEMORY	0000	WORD	PUBLIC	'MEMORY'
ENTXCM	0037	WORD	NONE	'CODE'
MAIN_STARTUP	007E	PARA	NONE	'MEMORY'

3.6 Reading a Symbol Table

MASM lists the names, types, values, and attributes of all symbols in the source file. The table has four columns: Name, Type, Value, and Attr.

The Name column lists the names of all symbols. The names in the list are given in alphabetical order and are spelled exactly as given in the source file. Names longer than 31 characters are truncated.

The Type column lists each symbol's type. The type can be any one of the following:

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

If Type is Number, Opcode, Alias, or Text, the symbol is defined by an EQU directive or an = directive. The Type column also lists the symbol's length if it is known. A length is given as one of the following:

Length	Meaning
BYTE	One byte (8-bits)
WORD	One word (16-bits)
DWORD	Doubleword (2 words)
FWORD	Tripleword (3 words)
QWORD	Quadword (4 words)
TBYTE	Ten bytes (5 words)

A length can also be given as a number. In this case, the symbol is a structure, and the number defines the length (in bytes) of the structure. For example, the following type identifies a label to a structure that is 31 bytes long.

L 0031

The Value column shows the numeric value of the symbol. For absolute symbols, the value represents an absolute number. For labels and variable names, the value represents that item's offset from the beginning of the segment in which it is defined. If Type is Number, Opcode, Alias, or Text, the Value column shows the symbol's "value," even if the "value" is simple text. Number shows a constant numeric value. Opcode shows a blank (the symbol is an alias for an instruction mnemonic). Alias shows the name of another symbol. Text shows the text the symbol represents. "Text" is any operand that does not fit one of the other three categories.

The Attr column lists the attributes of the symbol. The attributes include the name of the segment in which the symbol is defined, if any, the scope of the symbol, and the code length. A symbol's scope is given only if the symbol is defined using the EXTRN or PUBLIC directive. The scope can be External or Global. The code length is given only for procedures.

Example
Symbols:

Name	**Type**	**Value**	**Attr**
SYM	Number	0005	
SYM1	Text	1.234	
SYM2	Number	0008	
SYM3	Alias	SYM4	
SYM4	Text	5[BP][DI]	
SYM5	Opcode		
SYM6	L BYTE	0002	DATA
SYM7	L WORD	0012	DATA Global
SYM8	L DWORD	0022	DATA
SYM9	L QWORD	0000	External
LAB0	L FAR	0000	External
LAB1	L NEAR	0010	CODE

3.7 Reading a Pass 1 Listing

When you specify the **-d** option in the MASM command line, MASM adds a Pass 1 listing to the assembly-listing file, making the assembly-listing file show the results of both assembler passes. The listing is intended to help locate the sources of phase errors.

The following examples illustrate the Pass 1 listing for a source file that assembled without error. Although an error was produced on Pass 1, MASM corrected the error on Pass 2 and completed assembly correctly.

During Pass 1, a JLE instruction to a forward reference produces an error message.

```
00177E 00JLESMLSTK
file(line) : error 9: Symbol not defined SMLSTK
0019    BB 1000                 MOV    BX,4096
001C               SMLSTK:
```

MASM displays this error since it has not yet encountered the definition for the symbol SMLSTK.

By Pass 2, SMLSTK has been defined and MASM can fix the instruction so no error occurs.

```
0017    7E 03                   JLE    SMLSTK
0019    BB 1000                 MOV    BX,4096
001C               SMLSTK:
```

The JLE instruction's code now contains 03 instead of 00. This is a jump of 3 bytes.

Since MASM generated the same amount of code for both passes, there was no phase error. If a phase error had occurred, MASM would have displayed an error message.

In the following program fragment, a mistyped label creates a phase error. In Pass 1, the label *GO* is used in a forward reference and creates a *Symbol not defined* error. MASM assumes that the symbol will be defined later and generates 3 bytes of code, reserving 2 bytes for the symbol's actual value.

```
0000                code  segment
0000 E9 0000  U     jmp   GO
file(line) : error 9: Symbol not defined GO
0003          GO    label byte
0003 B8 0001        mov   ax, 1

0006                code  ends
```

In Pass 2, the label *GO* is known to be a label of BYTE type, which is an illegal type for the JMP instruction. As a result, MASM produces only two bytes of code in Pass 2, one less than in Pass 1. The result is a phase error.

```
0000                code  segment
0003 R                    jmp   GO
file(line) : error 57:Illegal size for item
0003          GO    label byte
file(line) : error 6: Phase error between passes
0003 B8 0001        mov   ax, 1
0006                code  ends
```

Appendix A
Error Messages

A.1 Introduction

This appendix lists and explains the error messages that can be generated by the Macro Assembler, MASM, and the Linker ld.

A.2 Macro Assembler Messages

This section lists and explains the messages displayed by the Macro Assembler,MASM. MASM displays a message whenever it encounters an error during processing. It displays a warning message whenever it encounters questionable statement syntax.

An end-of-assembly message is displayed at the end of processing, even if no errors occurred. The message contains a count of errors and warning messages it displayed during the assembly. The message has the form:

n Bytes of symbol space free
n Warning Errors
n Severe Errors

This message is also copied to the source listing.

Assembler Errors

0: Block nesting error

Nested procedures, segments, structures, macros, IRC, IRP, or REPT are not properly terminated. An example of this error is closing an outer level of nesting with inner level(s) still open.

1: Extra characters on line

This occurs when sufficient information to define the instruction directive has been received on a line and superfluous characters beyond are received.

2: Register already defined

This will only occur if the assembler has internal logic errors.

3: Unknown symbol type

Symbol statement has something in the type field that is unrecognizable.

4: Redefinition of symbol

This error occurs on Pass 2 and succeeding definitions of a symbol.

5: Symbol is multi-defined

This error occurs on a symbol that is later redefined.

6: Phase error between passes

The program has ambiguous instruction directives such that the location of a label in the program changed in value between Pass 1 and Pass 2 of the assembler. An example of this is a forward reference coded without a segment override where one is required. There would be an additional byte (the code segment override) generated in Pass 2 causing the next label to change. You can use the **-D** option to produce a listing to aid in resolving phase errors between passes. See Chapter 2.

7: Already had ELSE clause

Attempt to define an ELSE clause within an existing ELSE clause (you cannot nest ELSE without nesting IF...ENDIF).

8: Not in conditional block

An ENDIF or ELSE is specified without a previous conditional assembly directive active.

9: Symbol not defined

A symbol is used that has no definition.

10: Syntax error

The syntax of the statement does not match any recognizable syntax.

11: Type illegal in context

The type specified is of an unacceptable size.

12: Should have been group name

Expecting a group name but something other than this was given.

13: Must be declared in Pass 1

An item was referenced before it was defined in Pass 1. For example, "IF DEBUG" is illegal if DEBUG is not previously defined.

14: Symbol type usage illegal

Illegal use of a PUBLIC symbol.

15: Symbol already different kind

Attempt to define a symbol differently from a previous definition.

16: Symbol is reserved word

Attempt to use an assembler reserved word illegally. For example, to declare MOV as a variable.

17: Forward reference is illegal

Attempt to reference something before it is defined in Pass 1.

18: Must be register

Register expected as operand but you furnished a symbol that was not a register.

19: Wrong type of register

Directive or instruction expected one type of register, but another was specified. For example, INC CS.

20: Must be segment or group

Expecting segment or group and something else was specified.

21: Symbol has no segment

Trying to use a variable with SEG, and the variable has no known segment.

22: Must be symbol type

Must be WORD, DW, QW, BYTE, or TB but received something else.

23: Already defined locally

Tried to define a symbol as EXTERNAL that had already been defined locally.

24: Segment parameters are changed

List of arguments to SEGMENT were not identical to the first time this segment was used.

25: Not proper align/combine type

SEGMENT parameters are incorrect.

26: Reference to mult defined

The instruction references something that has been multi-defined.

27: Operand was expected

Assembler is expecting an operand but an operator was received.

28: Operator was expected

Assembler was expecting an operator but an operand was received.

29: Division by 0 or overflow

An expression is given that results in a divide by 0 or a number larger then can be represented.

30: Shift count is negative

A shift expression is generated that results in a negative shift count.

31: Operand types must match

Assembler gets different kinds or sizes of arguments in a case where they must match. For example, MOV.

32: Illegal use of external

Use of an external in some illegal manner. For example, DB M DUP(?) where M is declared external.

33: Must be record field name

Expecting a record field name but received something else.

error 34: Must be record field name

Expecting a record name or field name and received something else.

error 35: Operand must have size

Expected operand to have a size, but it did not.

error 36: Must be var, label or constant

Expecting a variable, label, or constant but received something else.

error 37: Must be structure field name

Expecting a structure field name but received something else.

error 38: Left operand must have segment

Used something in right operand that required a segment in the left operand. (For example, ":.")

error 39: One operand must be const

This is an illegal use of the addition operator.

error 40: Operands must be same or 1 abs

Illegal use of the subtraction operator.

error 41: Normal type operand expected

Received STRUC, FIELDS, NAMES, BYTE, WORD, or DW when expecting a variable label.

error 42: Constant was expected

Expecting a constant and received an item that does not evaluate to a constant. For example, a variable name or external.

error 43: Operand must have segment

Illegal use of SEG directive.

error 44: Must be associated with data

Use of code related item where data related item was expected. For example, MOV AX,<code-label>.

error 45: Must be associated with code

Use of data related item where code item was expected.

error 46: Already have base register

Trying to double base register.

error 47: Already have index register

Trying to double index address.

error 48: Must be index or base register

Instruction requires a base or index register and some other register was specified in square brackets, [ ].

error 49: Illegal use of register

Use of a register with an instruction where there is no 8086 or 8088 instruction possible.

error 50: Value is out of range

Value is too large for expected use. For example, MOV AL,5000.

error 51: Operand not in IP segment

Access of operand is impossible because it is not in the current IP segment.

error 52: Improper operand type

Use of an operand such that the opcode cannot be generated.

error 53: Relative jump out of range

Relative jumps must be within the range -128 to +127 of the current instruction, and the specific jump is beyond this range.

error 54: Index displacement must be constant

Illegal use of index display.

error 55: Illegal register value

The register value specified does not fit into the "reg" field (the value is greater than 7).

error 56: No immediate mode

Immediate mode specified or an opcode that cannot accept the immediate. For example, PUSH.

error 57: Illegal size for item

Size of referenced item is illegal. For example, shift of a double word.

error 58: Byte register is illegal

Use of one of the byte registers in context where it is illegal. For example, "PUSH AL," is illegal.

error 59: CS register illegal usage

Trying to use the CS register illegally. For example, "XCHG CS,AX," is illegal.

error 60: Must be AX or AL

Specification of some register other than AX or AL where only these are acceptable. For example, the IN instruction.

error 61: Improper use of segment register

Specification of a segment register where this is illegal. For example, an immediate move to a segment register.

error 62: No or unreachable CS

Trying to jump to a label that is unreachable.

error 63: Operand combination illegal

Specification of a two-operand instruction where the combination specified is illegal.

error 64: Near JMP/CALL to different CS

Attempt to do a NEAR jump or call to a location in a different CS ASSUME.

error 65: Label can't have segment override

Illegal use of segment override.

error 66: Must have opcode after prefix

Use of a REPE, REPNE, REPZ, or REPNZ instructions without specifying any opcode after it.

error 67: Can't override ES segment

Trying to override the ES segment in an instruction where this override is not legal. For example, "STOS DS:TARGET" is illegal.

error 68: Can't reach with segment register

There is no ASSUME that makes the variable reachable.

error 69: Must be in segment block

Attempt to generate code when not in a segment.

error 70: Can't use EVEN on BYTE segment

Segment was declared to be byte segment and attempt to use EVEN was made.

error 72: Illegal value for DUP count

DUP counts must be a constant that is not 0 or negative.

error 73: Symbol already external

Attempt to define a symbol as local that is already external.

error 74: DUP is too large for linker

Nesting of DUPs was such that too large a record was created for the linker.

error 75: Usage of ? (indeterminate) bad

Improper use of the "?". For example, ?+5.

error 76: More values than defined with

Too many initial values given when defining a variable using a REC or STRUC type.

error 77: Only initialize list legal

Attempt to use STRUC name without angle brackets, < >.

error 78: Directive illegal in STRUC

All statements within STRUC blocks must either be comments preceded by a semicolon (;), or one of the Define directives.

error 79: Override with DUP is illegal

In a STRUC initialization statement, you tried to use DUP in an override.

error 80: Field cannot be overridden

In a STRUC initialization statement, you tried to give a value to a field that cannot be overridden.

error 81: Override is of wrong type

In a STRUC initialization statement, you tried to use the wrong size on override. For example, 'HELLO' for DW field.

error 82: Register can't be forward reference

error 83: Circular chain of EQU aliases

An alias EQU eventually points to itself.

error 84: 8087 opcode can't be emulated

Either the 8087 opcode or the operands you used with it produce an instruction that the emulator cannot support.

error 85: End of file, no END directive

You forgot an end statement or there is a nesting error.

error 98: Override value is wrong length

There is an improper sized value in a RECORD or STRUC field.

error 99: Line too long expanding <symbol>

The line became too long for one of the assembler's internal buffers.

error 100: Impure memory reference

You attempted to explicitly store into memory via the CS register.

error 101: Missing data; zero assumed

An instruction that expects an operand has failed to set one, and immediate zero has been used instead.

error 102: Segment at or near 64K limit

Too much code or data. (This is a 8086 or 80286 instruction set message only.)

error 103: Can't change processor after first segment

An instruction set directive, such as .286 , has been given when a conflicting directive, such as .386 , had previously been given.

error 104: Operand size does not match word size

An attempt has been made to push or pop an item different in size from the current wordsize.

error 105: Address size does not match word size

A 16-bit addressing mode has been used in a 32-bit segment or a 32-bit addressing mode has been used in a 16-bit segment.

error 106: Jump shortened, No-op instruction inserted

A **jmp** or *jcc* has been used where a **jmp** or *jcc* short would have sufficed.

A.3 Linker Error Messages

The error messages produced by the linker fall into three categories:

- Fatal error messages
- Linker error messages
- Warning messages

Fatal error messages indicate a severe problem, one that prevents the linker from processing the object code. After printing out a message about the fatal error, the linker terminates linking without producing an executable object file or checking for further errors. Fatal error messages have the following form:

<location> : fatal error L1xxx: <messagetext>

Linker error messages indicate a problem in the executable object file. After printing out a message, the linker produces the executable file and sets the error bit in the header. Linker error messages have the following form:

<location> : error L2xxx: <messagetext>

Warning messages are informational only; they do not prevent the linker from processing the relocatable object code into executable object code. Rather, warning messages just indicate possible problems in the executable object file. Warning messages have the following form:

<location> : warning L4xxx: <messagetext>

In the messages, *<location>* represents the input file, or pathname of the linker if an input file is not present. The *xxx* represents the message number, and *<messagetext>* defines the message.

A.3.1 Fatal Error Messages

The following messages identify fatal errors. The linker can not recover from a fatal error, instead the linker terminates linking after printing the fatal error message.

fatal error L1002: unrecognized option name

An unrecognized character was give following '–' on the command line.

fatal error L1004: badly formed number

An invalid numeric value was given for an option.

fatal error L1008: segment limit set too high

The number following the −S option is larger than 1024, which is the largest number allowed.

fatal error L1011: badly formed hex number

An invalid hexadecimal numeric value was given with an option.

fatal error L1012: number too large

A number was appended to an option greater than 2^32-1.

fatal error L1013: version number missing

The −v switch was given without a version number.

fatal error L1014: unrecognized Xenix version number

The number following the −v switch must be either 2, 3, or 5.

fatal error L1015: address missing

The −A switch requires an appended number.

fatal error L1016: −A and −F are mutually exclusive

The −A and −F switches are mutually exclusive.

fatal error L1018: Pagesize value missing

The −N option was given without a following pagesize number.

fatal error L1019: pagesize larger than 0xfe00

The −N option is given with a pagesize value larger than 0xfe00, which is the largest allowed.

fatal error L1020: no object modules specified

No object modules are specified on the command line. At least one must be specified for the linker to produce an output file.

fatal error L1023: terminated by user

An interrupt was issued while the linker was operating.

fatal error L1045: too many TYPDEF records

An object module contains more than 255 TYPDEF records.

fatal error L1046: too many external symbols in one module

An object module specifies more than the maximum limit of 1023 external symbols. Break the module into smaller parts or reduce the number of external references.

fatal error L1047: too many group, segment, and class names in one module

An object module contains too many group, segment, and class names. Reduce the number of groups, segments, or classes in the module.

fatal error L1048: too many segments in one module

An object module has more than the maximum limit of 255 segments. Split the module or combine segments.

fatal error L1049: too many segments

The program contains more than the default maximum limit of 128 segments. Relink the program using the -S option assigning an appropriate number of segments.

fatal error L1050: too many groups in one module

The module contains more than the maximum limit of 21 group definitions (GRPDEF records). Split the module or redefine group definitions.

fatal error L1051: too many groups

The program contains more than the maximum limit of 20 groups, not counting DGROUP. Reduce the number of group definitions.

fatal error L1053: symbol table overflow

The program contains more than the maximum limit of 256K symbols, such as public, external, segment, group, class, and file names. Reduce the amount of symbols.

fatal error L1054: requested segment limit too high

The linker does not have have sufficient memory to describe the number of segments requested by the -S option. Reduce the segment argument to a number below 1024.

fatal error L1057: data record too large

An LEDATA record contains more than the maximum limit of 1024 bytes of data. Note which translator, compiler or assembler, produced the incorrect object module.

fatal error L1070: segment size exceeds 64K

A single 16-bit segment contains more than 64K bytes of code or data. Reduce the size of the segment less than 64K.

fatal error L1072: common area longer than 65536 bytes

The program has more than the maximum limit of 64K of communal variables allowed for 8086 and 80286 executables. Note that this error is not generated by the macro assembler, but only by compilers supporting communal variables.

fatal error L1075: segment size exceeds *<number>*

A 32-bit segment exceeds the maximum limit of code or data imposed by the linker, which is indicated by *number*.
Reduce the size of the segment.

fatal error L1076: common area longer than 4G-1 bytes

The program has more than the maximum limit of 4G-1 bytes of communal variables allowed for 80386 executables. Note that this error is not generated by the macro assembler, but only by compilers supporting communal variables.

fatal error L1080: cannot open list file

The linker can not create the list (map) file.

fatal error L1081: out of space for run file

The disk on which the executable output file is being written to is full. Free more space on the disk and restart the linker.

fatal error L1083: cannot open run file

The disk on which the executable output file is being written to is full or the file already exists with read-only permissions. Free more space on the disk or change permissions.

fatal error L1085: cannot open temporary file

The disk on which the temporary file is being written to is full. Free more space on the disk and restart the linker.

fatal error L1086: scratch file missing

The linker is unable to open a temporary file recently created. Restart the linker.

fatal error L1087: unexpected end-of-file on scratch file

A temporary file recently created by the linker was unexpectedly reduced in size. Restart the linker.

fatal error L1088: out of space for list file

The disk on which the list file is being written to is full. Free more space on the disk and restart the linker.

fatal error L1091: unexpected end-of-file on library

All required data in the library file was not read before encountering the end-of-file. Replace the library file and restart the linker.

fatal error L1093: object not found

The object module specified on the command line does not exist. Restart the linker verifying the correct object module pathname.

fatal error L1101: invalid object module

One of the object modules specified on the command line is invalid. Restart the linker. If the fatal error persists, contact the XENIX system administrator.

fatal error L1103: attempt to access data outside segment bounds

A data record in an object module specifies data extending beyond the end of the segment. Note which translator, assembler or compiler, produced the incorrect object module and notify the XENIX system administrator.

Fatal error L1113: unresolved COMDEF, internal error

An internal error has occurred, notify the XENIX The program uses more than one segment and it is being linked impure. Impure executables can have only one segment.

fatal error L1121: *<name>: group larger than 4G-1 bytes*

The *name* 32-bit group contains segments larger than 4G-1 bytes.

fatal error L1122: *<name>: group larger than 64K bytes*

The *name* 16-bit group contains segments larger than 64K bytes.

fatal error L1123: *<name>: both 16-bit and 32-bit segments in group*

The *name* group contains both 16-bit and 32-bit segments.

fatal error L1124: relocation value missing

The **−Rt** or **−Rd** option was given without an argument.

fatal error L1125: stack size missing

The **−F** option was given without an argument.

A.3.2 Linking Error Messages

The following messages identify errors in the executable object file. The linker continues processing when encountering these errors.

error L2001: fixup(s) without data

A FIXUPP record occurs without a data record immediately preceding. Note which translator, compiler or assembler, produced the incorrect object file and notify the XENIX system administrator.

error L2002: fixup overflow near *num* in frame segment *name* target segment *segment name* target offset *number*

Some possible causes are: (1) A group is larger than 64K bytes; (2) the user's program contains an inter-segment short jump or inter-segment short call; (3) the user has a data item whose name conflicts with that of a subroutine in a library included in the link; and (4) the user has an EXTRN declaration inside the body of a segment. For example:

```
CODE    segment public 'code'
extern  main:far
start   proc          far
        call          main
        ret
start   endp
CODE    ends
```

The following construction is preferred:

```
extern   main:far
CODE     segment public 'code'
start    proc            far
         call            main
         ret
start    endp
CODE     ends
```

Revise the source and re-create the object file.

error L2011: *name : NEAR/HUGE conflict*

Both the NEAR and HUGE attributes are given for the *name* communal variable. This error only occurs in programs produced by compilers supporting communal variables.

error L2012: *name : array-element size mismatch*

The FAR *name* communal array is declared with two or more different array-element sizes. This error only occurs in programs produced by compilers supporting communal variables.

error L2025: *name : symbol defined more than once*

The public *name* symbol is defined more than once. Use only one declaration.

error L2029: unresolved externals

One or more symbols are declared external, but they are not declared in any other module or library. A list of unresolved external references appears after the error messages in the following form:

unresolved_external_symbol in file(s)

file ...

The *unresolved_external_symbol* is the symbol that is not resolved and *file* is the file(s) that the symbol is referenced.

A.3.3 Warning Error Messages

The following messages identify errors in the relocatable object file to be processed, or the pathname of the linker if the object file is not given.

warning L4020: *<name> : code segment size exceeds 65500*

The 16-bit code segment *name* of length 65,501 to 65,536 bytes is unreliable on the 80286.

warning L4031: *<name> : segment declared in more than one group*

The *name* segment is declared in more than one group.

warning L4032: *<name> : segment defined both 16-, 32-bit, assuming 32*

The *name* segment is defined both as a 16-bit and 32-bit segment and is marked as a 32-bit segment in the segment table.

warning L4050: too many public symbols

The maximum limit of 3072 public symbols has been defined, causing the **−m** option not to sort the symbols in the map file.

warning L4060: code group longer than 65530

A group containing 16-bit code segments with a total length of 65,501 to 65,536 bytes is unreliable on the 80286.

warning L4061: multiple code segments--should be medium model

The program defines more than one code segment and the **−Mm**, **−Ml**, **−Mh**, or **−Me** option was not given. Verify that all modules have the same memory model or link with the **−Me** option.

warning L4062: multiple data segments--should be large model

The program defines more than one data segment and the **-Ml**, **-Mh**, or **-Me** option was not given. Verify that all modules have the same memory model or link with the **-Me** option.

warning L4063: stack option ignored for 80386 executable

A **-F** option was given while linking an 80386 library, linker ignored the **-F** option.

warning L4064: page-alignment option ignored for 80286 executable

A **-N** option was given while linking an 80286 library, linker ignored the **-N** option.

Index

XENIX® System V

Development System

Macro Assembler Reference Manual

This document was typeset with an IMAGEN® 8/300 Laser Printer.

SCO Document Number: XG-5-1-87-4.0

Contents

Chapter 1 Introduction

1.1 Introduction

This guide describes the usage and input syntax of the Macro Assembler, MASM (MASM(CP)). The assembler produces relocatable object modules from 8086, 8088, 80186, 80286, and 80386 assembly language source files. The relocatable object modules can be linked to create executable programs for the XENIX operating system.

MASM is an assembler for the Intel® 8086, 80186, 80286, and 80386 families of microprocessors. It can assemble the instructions of the 8086, 8088, 80186, 80286, and 80386 microprocessors and of the 8087 and 80287 floating-point coprocessors. It has a powerful set of assembly language directives that give the programmer complete control of the segmented architecture of the 8086, 80186, 80286, and 80386 microprocessors. MASM instruction syntax allows a wide variety of operand data types, including integers, strings, packed decimals, floating point numbers, structures, and records.

MASM is a macro assembler. It has a full set of macro directives that let a programmer create and use macros in a source file. The directives instruct MASM to repeat common blocks of statements or replace macro names with the block of statements which they represent. MASM also has conditional directives that let the programmer exclude portions of a source file from assembly or include additional program statements by simply defining a symbol.

MASM carries out strict syntax checking of all instruction statements, including strong typing for memory operands. Unlike other assemblers, MASM detects questionable operand usage that can lead to errors or unwanted results.

MASM produces object modules that are compatible with those created by high-level language compilers. Thus, you can make complete programs by combining MASM object modules with object modules created by C, Pascal, FORTRAN, or other high-level-language compilers.

This guide does not teach assembly language programming, or give a detailed description of 8086, 80186, 80286, or 80386 instructions. For information on these topics you will need other references.

1.2 About This Guide

This guide is organized as follows:

Chapter 1, "Introduction," explains what steps you need to take to create these programs, and summarizes the organization of this guide and the conventions used.

Chapter 2, "Elements of the Assembler," describes the characters which can be used in a program and how to form numbers, names, statements, and comments.

Chapter 3, "Program Structure," describes program structures and instruction set directives.

Chapter 4, "Types and Declarations," explains how to generate data for a program, and how to declare labels, variables, and other symbols that refer to instruction and data locations. It also explains how to define types that can be used to generate data blocks that contain multiple fields, such as structures and records.

Chapter 5, "Operands and Expressions," describes the syntax and meaning of operands and expressions used in assembly language statements and directives.

Chapter 6, "Global Declarations," describes global declaration directives in detail.

Chapter 7, "Conditional Assembly," describes the directives that provide conditional assembly of blocks of statements within a source file.

Chapter 8, "Macro Directives," explain how to create and use macros in your source files.

Chapter 9, "File Control," describes directives that provide control of the source, object, and listing files read and created by MASM during an assembly.

Appendix A, "Instruction Summary," gives a complete list of the instruction names and syntax for all processors.

Appendix B, "Directive Summary," gives a complete list of the directives used by MASM with their syntax and function.

Appendix C, "Segment Names For High-Level Languages," describes the naming conventions used to form assembly language source files that are compatible with object modules produced by the Microsoft C, Pascal, and Fortran language compilers (version 3.0 or later).

1.3 What You Need

This guide is intended to be used with the *Macro Assembler User's Guide*, which explains the steps required to create executable programs from source files.

You also need to know the function and operation of the instruction sets for the 8086, 80186, 80286, and 80386 families of microprocessors. For an explanation of these instruction sets, refer to one of the many books that define these instructions. Refer to Appendix A, "Instruction Summary," for a complete list of the instruction names and syntax for all processors.

1.4 Notational Conventions

This manual uses the following notational conventions to define the assembly-language syntax:

Roman	Indicates command, keyword, directive, or parameter names that must be typed as shown. In most cases, uppercase roman represents keywords and directives, and lowercase roman represents commands and parameters. Uppercase and lowercase letters can be freely mixed in some cases.
Bold	Indicates command-line options and arguments used to call the assembler and assembler options.
Italics	Indicates placeholders or parameters; i.e., a name that you must replace with the value or file name required by the program. Also indicates file names and path names in text.
Ellipsis dots...	Indicate that you can repeat the preceding item any number of times.
Commas ,,,	Indicate that you can repeat the preceding item any number of times, as long as you separate the items with a comma.

[Brackets]	Indicate that the enclosed item is optional. If you do not use the optional item, the program selects the default action.
Vertical bar |	Indicates that only one of the separated items can be used. You must make a choice between the items.
"Quotation marks"	Indicate text from a source-code example.
I/O	Indicates source code created with a text editor.

Chapter 2
Elements of the Assembler

2.1 Introduction

All assembly-language programs consist of one or more statements and comments. A statement or comment is a combination of characters, numbers, and names. Names and numbers are used to identify values in instruction statements. Characters are used to form names, numbers, and character constants.

The next section lists the characters that can be used in a program and the following sections describe how to form numbers, names, statements, and comments.

2.2 Characters

MASM recognizes the following character set:

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

0 1 2 3 4 5 6 7 8 9

? @ _ $: Microsoft . [] () <> {}

+ - / * & % ! ' " |\ = # ^ ; , `

2.3 Integers

Syntax

digits

*digits*B

*digits*Q

*digits*O

*digits*D

*digits*H

An integer is an integer number: a combination of binary, octal, decimal, or hexadecimal *digits* plus an optional radix. The *digits* are combinations of one or more digits of the specified radix: B, Q, O, D, or H. The real-number designator R can also be used. If no radix is given, MASM uses the current default radix (decimal). The following table lists the digits that can be used with each radix.

Digits Used with Each Radix

Radix	Type	Digits
B	Binary	0 1
Q	Octal	0 1 2 3 4 5 6 7
O	Octal	0 1 2 3 4 5 6 7
D	Decimal	0 1 2 3 4 5 6 7 8 9
H	Hexadecimal	0 1 2 3 4 5 6 7 8 9 A B C D E F

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

The real number-designator can be used only with hexadecimal numbers consisting of 8, 16, or 20 significant digits (a leading 0 can be added).

The maximum number of digits in an integer depends on the instruction or directive in which the integer is used. The default radix can be specified by using the .RADIX directive (see the section on the .RADIX directive in the "File Control," cbapter.

Examples

```
01010b       132q  5ah   90d   90
01111b       17O   0fh   15d   15
```

2.4 Real Numbers

Syntax

[+ | -] *digits.digits* E [+ | -] *digits*

A real number is a number consisting of an integer, a fraction, and an exponent. The *digits* can be any combination of decimal digits. Digits to tbe left of the decimal point (.) represent the integer. Those to the right of the point represent the fraction. The digits following the exponent mark (E) represent the exponent. If an exponent is given, the plus (+) and minus (–) signs can be used to indieate its sign.

Real numbers can be used only with the DD, DQ, and DT directives. The maximum number of digits in the number and the maximum range of exponent values depends on the directive.

Examples

```
DD   25.23
DQ   2.523E1
DT   2523.0E-2
```

2.5 Encoded Real Numbers

Syntax

*digits*R

An encoded real number is an 8-, 16-, or 20-digit hexadecimal number that represents a real number in encoded format. An encoded real number has a sign, a biased exponent, and a mantissa. These values are encoded as bit fields within the number. The exact size and meaning of each bit field depends on the number of bits in the number. The *digits* must be hexadecimal digits. The number must begin with a decimal digit (0-9) and must be followed by the real number-designator (R).

Encoded real numbers can be declared only with the DD, DQ, and DT directives. The number of digits for the encoded numbers used with DD, DQ, and DT must be 8, 16, and 20 digits, respectively. (If a leading 0 is supplied, the number must be 9, 17, or 21 digits.) See the "Data Declarations" section in the "Types and Declarations" chapter.

Examples

```
DD   3f800000           ; 1.0 for DD
DQ   3ff0000000000000   ; 1.0 for DQ
```

2.6 Packed Decimal Numbers

Syntax

[+ | –] *digits*

A packed decimal number represents a decimal integer that is to be stored in packed decimal format. Packed decimal storage has a leading sign byte and 9 value bytes. Each value byte contains two decimal digits. The high-order bit of the sign byte is 0 for positive values, and 1 for negative values.

Packed decimals have the same format as other decimal integers, except that they can take an optional plus sign (+) or minus sign (–) and can be defined only with the DT directive. A packed decimal must not have more than 18 digits.

Examples

```
        DT      1234567890   ; encoded as 00000000001234567890
        DT      -1234567890  ; encoded as 80000000001234567890
```

2.7 Character and String Constants

Syntax

' *characters*'

" *characters* "

A character constant consists of a single ASCII character. A string constant is composed of two or more ASCII characters. Constants must be enclosed in single quotation marks (') or double quotation marks ("). String constants are case sensitive.

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

Examples

```
'a'
'ab'
"a"
"This is a message."
'Can't find the file.'
"Specified ""value"" not found."
```

2.8 Names

Syntax

characters...

A name is a combination of letters, digits, and special characters used as a label, variable, or symbol in an assembly-language statement. Names have the following formatting rules:

- A name must begin with a letter, an underscore (_), a question mark (?), a dollar sign ($), or an at sign (@).
- A name can include any combination of upper- and lowercase letters. All uppercase and lowercase letters remain the same during assembly unless either the **-Ml** or **-Mx** option is used.
- A name can have any number of characters, but only the first 31 characters are used. All other characters are ignored.

Examples

```
subrout3
array
_main
```

2.9 Reserved Names

A reserved name is any name with a special, predefined meaning to the assembler. Reserved names include instruction and directive mnemonics, register names, and operator names. These names can be used only as defined and cannot be redefined.

All upper- and lowercase combinations of these names are treated as the same name. For example, the names "Length" and "LENGTH" represent the same operator, "LENGTH" operator.

The following table lists all reserved names except instruction mnemonics. For a complete list of instruction mnemonics, see Appendix A, "Instruction Summary."

Reserved Names

%OUT	CS	ESP	INCLUDE	QWORD
.186	CX	ELSE	IRP	.RADIX
.286	DB	END	IRPC	RECORD
.286C	DD	ENDIF	LABEL	.SALL
.286P	DF	ENDM	.LALL	SEG
.287	DH	ENDP	LE	SEGMENT
.386	DI	ENDS	LENGTH	.SFCOND
.386C	DL	EQ	.LFCOND	SHL
.386P	DQ	EQU	.LIST	SHORT
.8086	DR0	ES	LOCAL	SHR
.8087	DR1	EVEN	LOW	SI
=	DR2	EXITM	LT	SIZE
AH	DR3	EXTRN	MACRO	SP
AL	DR4	FAR	MASK	SS
AND	DR5	GE	MOD	STRUC
ASSUME	DR6	GROUP	NAME	SUBTTL
AX	DR7	GT	NE	TBYTE
BH	DS	HIGH	NEAR	.TFCOND
BL	DT	IF	NOT	THIS
BP	DW	IF1	OFFSET	TITLE
BX	DWORD	IF2	OR	TYPE
BYTE	DX	IFB	ORG	.TYPE
CH	EAX	IFDEF	PAGE	WIDTH
CL	EBP	IFDIF	PROC	WORD
COMMENT	EBX	IFE	.PROT	.XALL
CR0	ECX	IFIDN	PTR	.XCREF
CR2	EDI	IFNB	PUBLIC	.XLIST
CR3	EDX	IFNDEF	PURGE	XOR
.CREF	ESI	USE16	USE32	

2.10 Statements

Syntax

[*name*] *mnemonic* [*operands*] [*;comment*]

A statement is a combination of an optional *name,* mandatory instruction or directive *mnemonic,* one or more optional *operands,* and an optional *comment*. A statement represents an action to be taken by the assembler, such as generating a machine instruction or 1 or more bytes of data.

The following formatting rules apply to statements:

- A statement can begin in any column.
- A statement must not consist of more than 128 characters and must not contain an embedded carriage-return–line-feed character. In other words, continuing a statement on multiple lines is not allowed.
- All statements except the last one in the file must be terminated by a carriage-return–line-feed character.

Examples

```
count   db      0
        mov     eax, ebx    ; a comment
        assume  cs:_text, ds:dgroup
_main   proc    far
```

2.11 Comments

Syntax

;text

COMMENT *delimiter*
text
delimiter[*text*]

Comments describe the action of a program at a given point, but are ignored by the assembler. Comments using the semicolon (;) can be placed anywhere in a program, even on the same line as a statement. Comments using the COMMENT directive can be placed only on lines that do not contain other statements.

If the semicolon comment shares the line with a statement, it must be to the right of all names, mnemonics and operands. A semicolon comment is any combination of characters preceded by a semicolon (;) and terminated by a carriage-return–line-feed character.

A semicolon comment must not continue past the end of the line on which it begins; that is, it must not contain any embedded carriage-return–line-feed characters. For long comments, the COMMENT directive can be used.

The COMMENT directive causes MASM to treat all *text* between the two *delimiters* as a comment. The *delimiter* must be the first nonblank character following the COMMENT keyword. The *text* is all remaining characters up to the next occurrence of the *delimiter*. The *text* must not contain the delimiter.

The COMMENT directive is typically used for multiple-line comments. Note that text can appear on the same line as the last *delimiter* and is ignored by the assembler.

The first three examples use the standard semicolon (;) comment and the last two examples use the COMMENT directive.

Examples

```
        ; This comment is alone on a line.
mov     eax, ebx ; This comment follows a statement.
        ; Comments can contain reserved words like PUBLIC.

COMMENT *
This comment continues until the
next asterisk.
*

COMMENT +
The assembler ignores the
following MOV statement
+ moveax, 1
```

Chapter 3
Program Structure

3.1 Introduction

The following program structure directives let you define the organization that a program's code and data will have when loaded into memory:

SEGMENT	Segment definition
ENDS	Segment end
END	Source-file end
GROUP	Segment groups
ASSUME	Segment registers
ORG	Segment origin
EVEN	Segment alignment
PROC	Procedure definition
ENDP	Procedure end

3.2 Source Files

Every assembly language program is created from one or more source files that contain statements defining the program's data and instructions. MASM reads source files and assembles the statements to create object modules that can be prepared for execution by the system linker.

All source files have the same form – zero or more program "segments" followed by an END directive. The END directive, required in every source file, signals the end of the source file. The END directive also provides a way to define the program entry point or starting address. All other statements in a source file are optional.

The two program examples in this section illustrate the source-file format for assembly language programs. The examples are complete assembly language modules that use system calls to print the message "Hello." on the user terminal. Linking the modules with the standard C run-time library will produce a complete executable program. Note that the first example contains 286 instructions and the last example contains 386 instructions.

The following list summarizes the main features of each assembly language module:

- The END statements, which signify the end of the source file and define the program entry point or starting address.
- The SEGMENT and ENDS statements, which define segments named "_DATA" and "_TEXT"
- The GROUP statement defining a group, DGROUP, which contains the data segment "_DATA"
- The variables "HELLO" and "TTY" in the "_DATA" segment, defining the string to be displayed and the name of the file that is opened to do this. Note that all strings are terminated with a zero
- The instruction label "_MAIN" in the "_TEXT" segment and its PUBLIC declaration, which provides the necessary entry point for the run-time library to call.
- The ASSUME statements in the "_DATA" and "_TEXT" segments, defining which segment registers will be associated with the labels, variables, and symbols defined within the segments.

Examples

```
                .286c                                   ;Enable 286 instructions

_DATA           SEGMENT WORD PUBLIC 'DATA'              ;Program data segment
HELLO           db    "Hello.",0
TTY             db    "/dev/tty",0
fd              dw    0
_DATA           ENDS

DGROUP          GROUP _DATA

extrn           _open:near                              ;External entry points
extrn           _close:near
extrn           _write:near
extrn           _exit:near

_TEXT           SEGMENT BYTE PUBLIC 'CODE'              ;Program code segment

                ASSUME cs:_TEXT, ds:DGROUP,
                ss:DGROUP, es:DGROUP

PUBLIC          _MAIN

_MAIN:          ;Program entry point

                push  2                                 ;Fd = open ("/dev/tty", 2)
                push  offset DGROUP:TTY
                call  _open
                add   sp, 4
                mov   fd, ax

                push  7                                 ;Write (fd, &hello, 7)
                push  offset DGROUP:HELLO
                push  fd
                call  _write
                add   sp, 6

                push  fd                                ;Close (fd)
                call  _close
                add   sp, 2

                push  0                                 ;Exit (O)
                call  _exit

_TEXT           ENDS
                END
                .386                                    ; Enable 386 instructions

_DATA           SEGMENT DWORD PUBLIC USE32 'DATA'       ; Program Data Segment
HELLO           db "Hello.", 0
TTY             db "/dev/tty", 0
fd              dw 0
_DATA           ENDS

DGROUP          GROUP _DATA

extrn           _open:near
extrn           _close:near
extrn           _write:near
extrn           _exit:near

_TEXT           SEGMENT DWORD PUBLIC USE32 'CODE'       ; Program Code Segment

                ASSUME cs:_TEXT, ds:DGROUP,
                ss:DGROUP, es:DGROUP

PUBLIC          _MAIN

_MAIN:                                                  ; Program Entry Point

                push  2                                 ; fd = open("/dev/tty", 2)
                push  offset DGROUP:TTY
                call  _open
                add   esp, 8
                mov   fd, eax

                push  7                                 ; write(fd, &hello, 7)
                push  offset DGROUP:HELLO
                push  fd
                call  _write
                add   esp, 12

                push  fd                                ; close(fd)
                call  _close
                add   esp, 4

                push  0                                 ; exit(0)
                call  _exit

_TEXT           ENDS
                END
```

3.3 Instruction Set Directives

Syntax

.186
.286
.286C
.286P
.287
.386
.386C
.386P
.8086
.8087
.PRIV

The instruction set directives enable instruction sets for given microprocessors. When a directive is given, MASM will recognize and assemble any subsequent instructions belonging to that microprocessor. The instruction set directives, if used, must be placed at the beginning of the program source file.

The following list summarizes the 8086, 286, and 386 directives that enable each instruction set.

.8086	Enables the 8086 instruction set (default)
.8087	Enables 8087 floating-point instruction set
.186	Enables 186 instruction set
.286	Enables 286 instruction set, except for protected mode
.286C	Enables 286 instruction, except for protected mode
.286P	Enables 286 protected mode instruction set. It is equivalent to the following directive sequence: .286 .PRIV

.287	Enables 287 instruction set, but does not turn off 386 instruction set
.386	Enables 386 instruction set and defines the default word size to 4 bytes
.386C	Enables 386 instruction set and defines the default word size to 4 bytes
.386P	Enables 386 protected-mode instruction set and defines the default word size to 4 bytes. This directive is the equivalent to the following sequence: .386 .PROT
.PRIV	Enables protected mode instruction set

Even though a source file may contain the .8087 or .287 directive, MASM also requires the −r or −e option in the MASM command line to define how to assemble floating point instructions. The −r option directs the assembler to generate the actual instruction code for the floating point instruction. The −e option, enables MASM to generate the instruction codes which changed into software interrupts at program link time.

3.4 SEGMENT and ENDS Directives

Syntax

name SEGMENT *align* *[use]* *combine* *'class'*
name ENDS

The SEGMENT and ENDS directives mark the beginning and end of a program segment. A program segment is a collection of instructions or data whose addresses are all relative to the same segment register.

The *name* defines the name of the segment. This name can be unique or be the same name given to other segments in the program. Segments with identical names are treated as the same segment.

The *align, use, combine,* and *class* options give the linker instruction on how to setup segments. These options are described in the following sections.

The following 286 and 386 program examples each contain two segments: "SAMPLE_TEXT" and "CONST" . The "CONST" segment is nested within the "SAMPLE_TEXT" segment. Although a given segment name can be used more than once in a source file, each segment definition using that name must have either exactly the same attributes, or attributes that do not conflict.

Examples

```
                .286                            ; enable 286 instructions
SAMPLE_TEXT     SEGMENT word public 'code'
_main           proc far
                .
                .
                .

CONST           SEGMENT word public 'CONST'  ; nested segment
seg1            dw          array_data
CONST           ENDS                            ; end nesting

                mov         es, seg1
                push        es
                mov         ax, es:pointer
                push        ax
                call        _printf
                add         sp, 4
                .
                .
                .
                ret
_main           endp
SAMPLE_TEXT     ENDS

                .386                            ; Enable 386 instructions
SAMPLE_TEXT     SEGMENT word public 'code'
_main           proc       far
                .
                .
                .

CONST           SEGMENT word public 'CONST'
seg1            dw         array_data
CONST           ENDS                            ; End nesting

                mov        es, seg1
                push       es
                mov        eax, es:pointer
                push       eax
                call       _printf
                add        esp, 8
                .
                .
                .
                ret
_main           endp
SAMPLE_TEXT     ENDS
```

3.4.1 Align Type

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

BYTE	Use any byte address
WORD	Use any word address (2 bytes/word)
PARA	Use paragraph addresses (16 bytes/paragraph)
PAGE	Use page addresses (1024 bytes/page)
DWORD	Use any double word address (4 byte/dword)

If no *align* is given, PARA is used by default. The actual start address is computed when the program is loaded. The linker guarantees that the address will be on the given boundary.

3.4.2 Use Type

The *use* type defines the segment wordsize. The wordsize can be defined to be either 16 bits or 32 bits. To define a segment wordsize of 16 bits, use the USE16 directive. Similarly, to define a segment wordsize of 32 bits, use the USE32 directive. The default segment size is 32 bits.

Note

The USE16 and USE32 directives can only be used in 386 programs.

The following example defines two segments, "TEXT" and "DATA" , that have 16-bit and 32-bit segment wordsizes, respectively:

```
        .386            ; Enable 386 instruction set
        assume cs:_TEXT, ds:_DATA
_TEXT   segment word use16 public 'code'
                .
                .
                .
_TEXT   ends

_DATA   segment byte use32 public 'DATA'
                .
                .
                .
_DATA   ends
```

3.4.3 Combine Type

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

PUBLIC

Concatenates all segments having the same name to form a single, contiguous segment. All instruction and data addresses in the new segment are relative to a single segment register, and all offsets are adjusted to represent the distance from the beginning of the new segment.

STACK

Concatenates all segments having the same name to form a single, contiguous segment. This combine type is the same as the PUBLIC combine type, except that all addresses in the new segment are relative to the SS segment register. The stack pointer SP register is initialized to the last address of the segment. Stack segments should normally use the STACK type, since this automatically initializes the SS register. If you create a stack segment and do not use the STACK type, you must give instructions to load the segment address into the SS register.

COMMON

Creates overlapping segments by placing the start of all segments having the same name at the same address. The length of the resulting area is the length of the longest segment. All addresses in the segments are relative to the same base address.

MEMORY

Places all segments having the same name in the highest physical segment in memory. If more than one MEMORY segment is given, the segments are overlapped as with COMMON segments.

AT *address*

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

If no *combine* is given, the segment is not combined. Instead, it receives its own physical segment when loaded into memory.

3.4.4 Class Type

The *class* type defines which segments are to be loaded in contiguous memory. Segments having the same class name are loaded into memory one after another. All segments of a given class are loaded before segments of any other class. The *class* name must be enclosed in single quotation marks (').

The following example illustrates the general form of a text segment for a small module program. The segment name is "_TEXT" . The segment alignment, use, and combine type are "WORD" , "USE16" , and "PUBLIC" , respectively. The class is "CODE" .

Example

```
                .386
                assume    cs:_TEXT
_TEXT           segment   WORD USE16 PUBLIC 'CODE'
                          .
                          .
                          .
_TEXT   ends
```

3.4.5 Segment Nesting

Segments can be nested. When MASM encounters a nested segment, it temporarily suspends assembly of the enclosing segment and begins assembly of the nested segment. When the nested segment has been assembled, MASM continues assembly of the enclosing segment. Overlapping segments are not permitted.

The following example contains two segments: a code segment called "SAMPLE" and a data segment called "CONST" . The "CONST" segment is nested within the "SAMPLE" segment.

Example

```
SAMPLE    segment   word public 'code'      ; Outside segment
main      proc      far
          .
          .
          .
CONST     segment   word public 'CONST'     ; Inside segment
array     dw        array_data
CONST     ends
          .
          .
          .
          ret
main      endp
SAMPLE    ends
```

3.5 END Directive

Syntax

END [*expression*]

The END directive marks the end of the module. The assembler ignores any statements following this directive.

The optional *expression* defines the program entry point, the address at which program execution is to start. If the program has more than one module, only one of these modules can define an entry point. The module with the entry point is called the "main module." If no entry point is given, none is assumed.

Note

> If you fail to define an entry point for the main module, your program may not be able to initialize correctly. The program will assemble and link without error messages, but it may crash when you attempt to run it. Remember, one (and only one) module must define an entry point.

Examples

```
    END                 ; Does not define an entry
    END    _START       ; Defines _START as entry
```

3.6 GROUP Directive

Syntax

name GROUP *segment-name*,,,

The GROUP directive associates a group *name* with one or more segments. It also causes all defined labels and variables to have addresses relative to the group beginning address rather than to the segment beginning address. The *segment-name* must be the name of a segment defined using the SEGMENT directive, or a SEG expression The *name* must be unique.

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

Segments in a group do not have to be contiguous. Segments that do not belong to the group can be loaded between segments that do. The only restriction is that the distance (in bytes) between the first byte in the first segment of the group and the last byte in the last segment must not exceed 64K bytes in 8086 and 286 programs and 4G bytes in 386 programs.

Group names can be used with the ASSUME directive and as an operand prefix with the segment override operator (:) .

Note

> A group name must not be used in more than one GROUP directive in any source file. If several segments within the source file belong to the same group, all segment names must be given in the same GROUP directive.

Example

```
dgroup    GROUP      _data, _bss
          ASSUME     ds:dgroup

_data     segment word public 'data'
          .
          .
          .
_data     ends
_bss      segment word public 'bss'
          .
          .
          .
_bss      ends
          end
```

3.7 ASSUME Directive

Syntax

ASSUME *segment-register* : *segment-name* ,,,
ASSUME NOTHING

The ASSUME directive specifies *segment-register* as the default segment register for all labels and variables defined in the segment or group given by *segment-name*. Subsequent references to the label or variable will automatically assume the selected register when the effective address is computed.

The *segment-register* must be one of the following: CS, SS, DS, ES, FS, or GS.

Note

> The FS and GS segment registers are accessible only in 386 programs.

The *segment-name* must be one of the following:

- The name of a segment previously defined with the SEGMENT directive.
- The name of a group previously defined with the GROUP directive.
- The keyword NOTHING.

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

Note

The segment override operator (:) can be used to override the current segment register selected by the ASSUME directive.

Examples

```
ASSUME CS:code
ASSUME CS:dgroup,SS:dgroup,DS:dgroup
ASSUME ES:dgroup,FS:dgroup,GS:NOTHING
ASSUME NOTHING
```

3.8 ORG Directive

Syntax

ORG *expression*

The ORG directive sets the location counter to *expression*. Subsequent instruction and data addresses begin at the new value.

The *expression* must resolve to an absolute number. In other words, all symbols used in the expression must be known on the first pass of the assembler. The location counter symbol ($) can also be used.

In the example below, the statement "MOV EAX, EAX" begins at byte 120h in the current segment. Similarly, the variable "ARRAY" is declared to start at the address 2 bytes beyond the current address. See the section on the Location Counter Operand in the "Operands and Expressions" chapter for more information on the location-counter symbol ($).

Examples

```
        ORG     120h
        MOV     EAX, EDX
        ORG     $+2
ARRAY   dw      100 dup (0)
```

3.9 EVEN Directive

Syntax

EVEN

The EVEN directive aligns the next data or instruction byte on a word boundary. If the current value of the location counter is odd, the directive increments the location counter to an even value and generates one NOP (no operation) instruction. If the location counter is already even, the directive does nothing.

Note

The EVEN directive must not be used in byte-aligned segments.

In the following example, the EVEN directive tells MASM to increment the location counter, and generates a single NOP instruction (90h). This means the offset of "TEST2" is 2, not 1, as it would be without the EVEN directive.

Example

```
        org     0
test1   db      1
        EVEN
TEST2   dw      513
```

3.10 ALIGN Directive

Syntax

ALIGN [*size*]

The ALIGN directive aligns the next data or instruction byte on the current position in a segment. The *size* specifies the number of NOP instructions (90h) inserted.

The ALIGN directive must be given an argument that is a power of 2.

Note

Note that the EVEN directive is equivalent to ALIGN 2.

Examples

```
ALIGN 2
ALIGN 4
ALIGN 8
ALIGN 16
```

3.11 PROC and ENDP Directives

Syntax

name PROC *type*
    statements
name ENDP

The PROC and ENDP directives mark the beginning and end of a procedure. A procedure is a block of instructions that forms a program subroutine. Every procedure must have a *name*.

The *name* must be a unique name, not previously defined in the program. The optional *type* can be either NEAR or FAR. NEAR is assumed if no *type* is given. The *name* has the same attributes as a label, and can be used as an operand in a jump, call, or loop instruction.

Any number of *statements* can appear between the PROC and ENDP statements. The procedure should contain at least one RET directive to return control to the point of call. Nested procedures are allowed.

Examples

```
        .286                                ; Enable 286 instructions
_main   PROC    NEAR
        push    bp
        mov     bp, sp
        mov     ax, offset dgroup:string
        push    ax
        call    _printf
        add     sp, 2
        mov     sp, bp
        pop     bp
        RET
_main   ENDP

        .386                                ; Enable 386 instructions
_main   PROC    NEAR
        push    ebp
        mov     ebp, esp
        push    offset dgroup:string
        call    _printf
        add     esp, 4
        mov     esp, ebp
        pop     ebp
        RET
_main   ENDP
```

Chapter 4
Types and Declarations

4.1 Introduction

This chapter explains how to generate data for a program, how to declare labels, variables, and other symbols that refer to instruction and data locations, and how to define types that can be used to generate data blocks that contain multiple fields, such as structures and records.

4.2 Label Declarations

Label declarations create "labels" . A label is a name that represents the address of a instruction. Labels can be used in jump, call, and loop instructions to direct program execution to the instruction at the address of the label.

4.2.1 Near Label Declarations

Syntax

name:

A near label declaration creates an instruction label that has NEAR type. The label can be used in subsequent instructions in the same segment to pass execution control to the corresponding instruction.

The *name* must be unique, not previously defined, and it must be followed by a colon (:). Furthermore, the segment containing the declaration must be associated with the CS segment register. The assembler sets the name to the current value of the location counter.

A near label declaration can appear on a line by itself or on a line with an instruction. Labels must be declared with the PUBLIC or EXTRN directive if they are located in one module but called from another module.

Examples

```
start:
loop:  inc    4[bp]
for:   inc    8[ebp]
```

4.2.2 Procedure Labels

Syntax

name PROC [NEAR|FAR]

The PROC directive creates a label *name* and assigns its type to NEAR or FAR. The label then represents the address of the following instruction and can be used in a jump, call, or loop instruction to direct execution control to the first instruction of the procedure. If you do not specify the NEAR or FAR type, MASM assumes NEAR as the default.

When the PROC label definition is encountered, the assembler sets the label's value to the current value of the location counter and sets its type to NEAR or FAR. If the label has FAR type, the assembler also sets its segment value to that of the enclosing segment.

NEAR labels can be used with jump, call, and loop instructions to transfer program control to any address in the current segment. FAR labels can be used to transfer program control to an address in any segment outside the current segment.

Labels must be declared with the PUBLIC and EXTRN directive if they are located in one module but called from another module.

4.3 Data Declarations

The data declaration directives let you generate data for a program. The directives translate numbers, strings, and expressions into individual bytes, words, or other units of data. The encoded data is copied to the program object file.

The data declaration directives are listed below:

DB	Data byte
DW	Data word
DD	Data doubleword
DF	Data farword (6 bytes)
DQ	Data quadword
DT	Data ten-byte word

4.3.1 DB Directive

Syntax

[*name*] DB *initial-value* ,,,

The DB directive allocates and initializes a byte (8 bits) of storage for each *initial-value*. The *initial-value* can be an integer, a character string constant, a DUP operator, a constant expression, or question mark (?). The question mark (?) represents an undefined initial value. If two or more initial values are given, they must be separated by commas (,).

The *name* is optional. If a *name* is given, the directive creates a variable of the type BYTE, whose offset value is the current location counter value.

A string constant can have any number of characters, as long as it fits on a single line. When the string is encoded, the characters are stored in the order given, with the first character in the constant at the lowest address and the last at the highest.

Examples

```
integer      DB   16
string       DB   'ab'
message      DB   "Enter your name: "
constantexp  DB   4 * 3
empty        DB   ?
multiple     DB   1, 2, 3, '$'
duplicate    DB   10 DUP(?)
high_byte    DB   255
```

4.3.2 DW Directive

Syntax

[*name*] DW *initial-value* ,,,

The DW directive allocates and initializes a word (2 bytes) of storage for each given *initial-value*. An *initial-value* can be an integer, a one or two character string constant, a DUP operator, a constant expression, an address expression, or a question mark (?). The question mark (?) represents an undefined initial value. If two or more expressions are given, they must be separated by commas (,).

The *name* is optional. If a *name* is given, the directive creates a variable of the type WORD, whose offset value is the current location counter value.

String constants must not exceed two characters in length. The last (or only) character in the string is placed in the low-order byte, and either 0 or the first character is placed in the high-order byte.

Examples

```
integer      DW  16728
character    DW  'a'
string       DW  'bc'
constantexp  DW  4 * 3
addressexp   DW  string
empty        DW  ?
multiple     DW  1, 2, 3, '$'
duplicate    DW  10 DUP(?)
high_word    DW  65535
arrayptr     DW  array
arrayptr2    DW  offset dgroup:array
```

4.3.3 DD Directive

Syntax

[*name*] DD *initial-value* ,,,

The DD directive allocates and initializes a doubleword (4 bytes) of storage for each given *initial-value*. An *initial-value* can be an integer, a real number, a one or two character string constant, an encoded real number, a DUP operation, a constant expression, an address expression, or a question mark (?). The question mark (?) represents an undefined initial value. If two or more expressions are given, they must be separated by commas (,).

The *name* is optional. If a *name* is given, the directive creates a variable of the type DWORD, whose offset value is the current location counter value.

String constants must not exceed two characters in length. The last (or only) character in the string is placed in the low-order byte, and the first character (if there are two in the string) is placed in the next byte. Zeroes are placed in all remaining bytes.

If the DD directive is used in a USE16 segment to declare a pointer (8086 or 286 program defualt) with an address as an argument, then the DD directive declares a near pointer. If the DD directive is used in a USE32 segment (386 program default), then the DD directive declares a far pointer.

Examples

```
integer      DD   16728
character    DD   'a'
string       DD   'bc'
real         DD   1.5
encodedreal  DD   3f000000r
constantexp  DD   4 * 3
addsegexp    DD   real
empty        DD   ?
multiple     DD   1, 2, 3, '$'
duplicate    DD   10 DUP(?)
high_double  DD   4294967295
```

4.3.4 DF Directive

Syntax

[*name*] DF *initial-value* ,,,

The DF directive allocates and initializes a farword (6 bytes) of storage for each given *initial-value*. An *initial-value* can be an address or integer. If two or more initial values are given, they must be separated by commas (,).

The *name* is optional. If a *name* is given, the directive creates a variable of the type FWORD, whose offset value is the current location counter value.

Examples

```
multiple      DF    1, 2, 3, 4
address       DF    2120h, 4222h
integer       DF    140737500000000
empty         DF    ?
addsegexp     DF    segment
constantexp   DF    22*12316[illegible]
duplicate     DF    10 DUP (?)
```

4.3.5 DQ Directive

Syntax

[*name*] DQ *initial-value* ,,,

The DQ directive allocates and initializes a quadword (8 bytes) of storage for each *initial-value*. An *initial-value* can be an integer, a real number, a one or two character string constant, an encoded real number, a DUP operator, a constant expression, or a question mark (?). The question mark (?) represents an undefined initial value. If two or more initial values are given, they must be separated by commas (,).

The *name* is optional. If a *name* is given, the directive creates a variable of the type QWORD, whose offset value is the current location counter value.

String constants must not consist of more than two characters. The last (or only) character in the string is placed in the low-order byte, and the first character (if there are two in the string) is placed in the next byte. Zeroes are placed in all remaining bytes.

Examples

```
integer      DQ   16728
character    DQ   'a'
string       DQ   'bc'
real         DQ   1.5
encodedreal  DQ   3f00000000000000r
constantexp  DQ   4 * 3
empty        DQ   ?
multiple     DQ   1, 2, 3, '$'
duplicate    DQ   10 DUP(?)
high_quad    DQ   18446744073709551615
```

4.3.6 DT Directive

Syntax

[*name*] DT *initial-value* ,,,

The DT directive allocates and initializes 10 bytes of storage for each *initial-value.* An *initial-value* can be an integer expression, a packed decimal, a one or two character string constant, an encoded real number, a DUP operator, or a question mark (?). The question mark (?) represents an undefined initial value. If two or more expressions are given, they must be separated by commas (,).

The *name* is optional. If a *name* is given, the directive creates a variable of the type TBYTE, whose offset value is the current location counter value.

String constants must not consist of more than two characters. The last (or only) character in the string is placed in the low-order byte, and the first character (if there are two in the string) is placed in the next byte. Zeroes are placed in all remaining bytes.

Note

> The DT directive assumes that constants with decimal digits are packed decimals, not integers. If you want to specify a 10-byte integer, you must follow the integer with the appropriate letter specifying the numbering system you are using (for example, "D" or "d" for decimal or "H" or "h" for hexadecimal).

Examples

```
packeddecimal   DT   1234567890
integer         DT   16728D
character       DT   'a'
string          DT   'bc'
real            DT   1.5
encodedreal     DT   3f000000000000000000r
empty           DT   ?
multiple        DT   1, 2, 3, '$'
duplicate       DT   10 DUP(?)
high_tbyte      DT   1208925819614629174706175d
```

4.3.7 DUP Operator

Syntax

count DUP (*initial-value,,,*)

The DUP operator is a special operator that can be used with the data declaration directives and other directives to specify multiple occurrences of one or more initial values. The *count* sets the number of times to define the *initial-value*. An initial value can be any expression that evalutes to an integer value, a character constant, or another DUP operator. If more than one initial value is given, the values must be separated by commas (,). DUP operators can be nested up to 17 levels.

In the following examples, the first example generates 100 bytes with the value 1. The second example generates 80 words of data. The first four words have the values 1, 2, 3, and 4, respectively. This pattern is duplicated for the remaining words. The third example generates 125 bytes of data, each byte having the value 1. The final example generates 14 doublewords of uninitialized data.

Examples

```
data1  db   100   DUP (1)
data2  dw   20    DUP ( 1,2,3,4 )
data3  db   5     DUP ( 5 DUP( 5 DUP (1)))
data4  dd   14    DUP (?)
```

4.4 Symbol Declarations

The symbol declaration directives let you create and use symbols. A symbol is a descriptive name representing a number, text, an instruction, or an address. Symbols make programs easier to read and maintain by using descriptive names to represent values. A symbol can be used anywhere its corresponding value is allowed.

The symbol declaration directives are listed below:

=	Assign absolutes
EQU	Equate absolutes, aliases, or text symbols
LABEL	Instruction or data labels

4.4.1 Equal-Sign (=) Directive

Syntax

name = *expression*

The equal-sign (=) directive creates an absolute symbol by assigning the numeric value of *expression* to *name*. An absolute symbol is simply a name that represents a 16-bit value. No storage is allocated for the number. Instead, the assembler replaces each subsequent occurrence of the *name* with the value of the given *expression*. The value is a variable during assembly, but it is a constant at runtime.

The *expression* can be an integer, a one or two character string constant, a constant expression, or an address expression. The expression's value must not exceed 64K in 8086 and 286 programs or 4G in 386 programs. The *name* must be either a unique name, or a name that was previously defined using the equal-sign (=) directive.

Absolute symbols can be redefined at any time.

Examples

```
integer        =    16728
string         =    'ab'
constantexp    =    3 * 4
addressexp     =    string
```

4.4.2 EQU Directive

Syntax

name EQU *expression*

The EQU directive creates absolute symbols, aliases, or text symbols by assigning the *expression* to *name*. An absolute symbol is a *name* that represents 16-bit values in 8086 and 286 programs, and 32-bit values in 386 programs. An alias is a *name* that represents another symbol; and a text symbol is a *name* that represents a character string or other combination of characters. The assembler replaces each subsequent occurrence of *name* with either the text or the value of the *expression*, depending on the type of expression given.

The *name* must be a unique name, not previously defined. The *expression* can be an integer, a string constant, a real number, an encoded real number, an instruction mnemonic, a constant expression, or an address expression. Expressions that evaluate to integer values, in the range 0 to 64K in 8086 and 286 programs or 0 to 4G in 386 programs, create absolute symbols and cause MASM to replace *name* with a value. All other expressions cause the assembler to replace *name* with text.

The EQU directive is sometimes used to create simple macros. Note that the assembler replaces a name with text or a value before attempting to assemble the statement containing the name.

Symbols defined using EQU directive cannot be redefined.

Examples

```
integer      EQU  16728        ; replaced with value
real         EQU  3.14159      ; replaced with text
constantexp  EQU  3 * 4        ; replaced with value
memoryop     EQU  [bp]         ; replaced with text
memoryop     EQU  [ebp]        ; replaced with text
mnemonic     EQU  mov          ; replaced with text
addressexp   EQU  real         ; replaced with text
string       EQU  'Type Enter' ; replaced with text
```

4.4.3 LABEL Directive

Syntax

name LABEL *type*

The LABEL directive creates a new variable or label by assigning the current location counter value and the given *type* to *name*.

Name must be unique and not previously defined. *Type* can be any one of the following:

BYTE

WORD

DWORD

FWORD

QWORD

TBYTE

NEAR

FAR

Type can also be the name of a valid structure type.

Examples

```
subroutine   LABEL    FAR
barray       LABEL    BYTE
```

4.5 Structure and Record Declarations

The STRUC, ENDS, and RECORD directives define data types that can be used to create variables that consist of multiple elements. The directives data types associate one or more elements to a given structure or record name.

The structure and record declaration directives are listed below:

STRUC and ENDS	Structure declarations
RECORD	Record types

4.5.1 Defining Structures

Syntax

structuretype STRUC
    fieldindentifiers
structuretype ENDS

The STRUC and ENDS directives declare the beginning and end of a structure. The STRUC and ENDS directives define the name and type of a structure and the default values of the fields contained in the structure.

The *structuretype* defines the type and name of the structure. The *name* must be unique. The *fieldindentifiers* define the name and type of each field. Each *fieldindentifier* can be one of the following:

[*fieldname*] DB *initialvalue,,,*
[*fieldname*] DW *initialvalue,,,*
[*fieldna e*] DD *initialvalue,,,*
[*fieldna e*] DF *initialvalue,,,*
[*fieldna e*] DQ *initialvalue,,,*
[*fieldna e*] DT *initialvalue,,,*

The optional *fieldna e* defines the name of each field; the DB, DW, DD, DF, DQ, or DT directive defines the type of each field, and *initialvalue* defines the value given to each element when the variable is declared. Each *fieldna e* must be unique, and once defined, represents the offset from the beginning of the structure to the corresponding field. The *initialvalue* can define a number, character or string constant, or symbol. It may also contain the DUP operator to define multiple values. If the *initialvalue* is a string constant, the field contains the same number of bytes as characters in the string. If multiple *initialvalues* are given, each *initialvalue* must be separated by a comma (,).

A structure can contain *fieldindentifiers* and comments only. It can not contain any other statements. Therefore, structures cannot be nested.

In the following example, the structure type and name is "TABLE" and the *fieldindentifiers* are "COUNT" , "VALUE" , and "NAME" . The "COUNT" field is a single byte value initialized to 10. The field "VALUE" is an array of 10 uninitialized word values. The field "NAME" is a character array of 5 bytes initialized to "FONT3" . The *fieldindentifiers* "COUNT" , "VALUE" , and "NAME" have the offset values 0, 1 and 21, respectively.

Example

```
TABLE     STRUC
COUNT     DB   10
VALUE     DW   10 DUP (?)
NAME      DB   'FONT3'
TABLE     ENDS
```

4.5.2 Declaring Structures

Syntax

[*variablename*] *structurename* < [*initial-value*],,, >

A variable can contain a structure of one or more fields of different sizes. The *variablename* is the name of the symbol, *structurename* is the type of structure created by the STRUCT and ENDS directives, and *initial-value* is one or more values defining the initial value of the structure's field(s). One *initial-value* can be given for each field in the structure.

The *variablename* is optional. If *variablename* is not given, MASM allocates space for the structure, but does not create a name that you can use to access the structure.

The *initial-value* can be an integer, string constant, or expression that evaluates to a value having the same type as the corresponding field. The angle brackets (< >) are required even if no *initial-value* is given. If more than one *initial-value* is given, the values must be separated with commas (,). If the DUP operator is used, only the values within the parentheses need to be enclosed in angle brackets.

You do not have to initialize all fields in a structure. If an initial value is left blank, MASM automatically uses the default initial value of the field, which was originally determined by the structure type. If there is no default value, the field is uninitialized. The "Structure Operands" section in the "Operands and Expressions" chapter illustrates several ways to use structure data after they have been declared.

Note

You cannot initialize any structure field that has multiple values if the field was given a default initial value when the structure was defined. For example, assume the following structure definition:

```
strings   STRUC
          BUFFER    db 100 dup (?)      ; Can't override
          CRLF      db 13, 10           ; Can't override
          QUERY     db 'Filename: '     ; String <= can override
          ENDMARK db 36
strings   ENDS
```

The "BUFFER" and "CRLF" variables cannot be overridden because they have multiple values. The "QUERY" variable can be overridden as long as the overriding data are no longer than "QUERY" (10 bytes). Similarly, the "ENDMARK" filed can be overridden by any byte value.

The first example creates a variable named "STRUCT1" . All fields of the structure are intialized with "TABLE" default values. The second example creates a variable named "STRUCT2" that contains one structure of type "TABLE" . The first field of "STRUCT2" is initialized to zero, the remaining fields are intialized to their corresponding default values. The final example creates a variable named "STRUCT3" that contains 10 structures of the type "TABLE" . The first field in each structure is set to the initial value 0. The remaining fields are set to the default values.

Examples

```
STRUCT1   TABLE     <>
STRUCT2   TABLE     <0,,>
STRUCT3   TABLE     10 DUP(<0,,>)
```

4.5.3 Defining Records

Syntax

recordtype RECORD *fieldname:width* [=*expression*],,,

The RECORD directive defines an 8- or 16-bit record that contains one or more fields. The *recordtype* is the type and name of the record, *fieldname* is the name of a field in the record, *width* is the number of bits in the field, and *expression* is the initial (or default) value for the field.

Any number of *field:width=expression* combinations can be given for a record, as long as each is separated from the preceding with a comma (,). The sum of the widths for all fields must not exceed 16 .

The *width* must be an integer value in the range 1 to 16. If the total width of all declared fields is larger than 8 hits, then the assembler allocates 2 bytes to the field. Otherwise, only 1 byte is allocated.

If =*expression* is given, it defines the initial value for the field. If the field is at least 7 bits wide, you can use an ASCII character for *expression*. The *expression* must not contain a forward reference to any symbol. If the =*expression* is not given, the RECORD directive does not allocate any space.

In all cases, the first field you declare goes into the most significant bits of the record. Successively declared fields are placed in the succeeding bits to the right. If the fields you declared do not total exactly 8 bits, or exactly 16 bits, the entire record is shifted right so that the last bit of the last field is the lowest bit of the record. Unused bits will be initialized to 0 in the high end of the record.

The first example creates a record of type "ENCODE" that contains three fields: "HIGH" , "MID" , and "LOW" . Each variable declared with the record of type "ENCODE" will occupy 16 bits of memory. The field "HIGH" is in bits 6 to 9, field "MID" in bits 3 to 5, and field "LOW" in bits 0 to 2. The remaining high-order bits are unused. The bit diagram below shows the record of type "ENCODE" .

```
        hi    mid  lo
000000  0000  000  000
765432  1076  543  210
```

The second example creates a record of type "ITEM" that contains two fields: "CHAR" and "WEIGHT" . The two fields are initialized with the default values of the letter Q and the number 2, respectively. Unused bits are set to 0. The bit diagram below shows the record of type "ITEM" .

```
        char     wt.
00000   1010001  0010
76543   2107654  3210
```

Examples

```
ENCODE    RECORD    HIGH:4, MID:3, LOW:3
ITEM      RECORD    CHAR:7='Q', WEIGHT:4=2
```

4.5.4 Declaring Records

Syntax

[*variablename*] *recordtype* < [*initial-value*],,, >

A variable can contain a 8- or 16-bit record whose bits are divided into one or more fields. The *variablename* is the name of the variable, *recordtype* is the type and name of the record that has been created using the RECORD directive, and *initial-value* is one or more values defining the initial value of the record. One *initial-value* can be given for each field in the record.

The *variablename* is optional. If no *variablename* is given, MASM allocates space for the record, but does not create a variable that can access the record.

The *initial-value* can be an integer, string constant, or any expression that evaluates to a value no larger than can be represented in the specified field width when the record was defined. Angle brackets (< >) are required even if no initial value is given. If more than one initial value is given, the values must be separated with commas (,). If the DUP operator is used, only the values within the parentheses need to be enclosed in angle brackets. You do not have to initialize all fields in a record. If an initial value is left blank, MASM automatically uses the default initial value of the field. If there is no default value, then the field is uninitialized.

The first example creates the variable "REC1" containing the record of type "ENCODE" . The initial values of the fields in the record are initialized to the default values defined in the record. The second example creates the variable "TABLE" that contains 10 records of type "ITEM" . The fields in these records are all set to the initial values A and 2. The bit diagram shows one of the 10 bytes created:

```
        char     wt.
00000   1000001  0010
76543   2107654  3210
```

The final example creates the variable "PASSKEY" that contains the record of type "ENCODE" . The first two fields of the variable are initialized with the default values defined for the record. The last field is initialized with the initial value of "7" . The following bit diagram shows the variable's contents:

```
        hi     mid   lo
00000   00000  000   111
76543   21076  543   210
```

Examples

```
REC1      ENCODE   <>
TABLE     ITEM     10 DUP(<'A',2>)
PASSKEY   ENCODE   <,,7>
```

Chapter 5
Operands and Expressions

5.1 Introduction

This chapter describes the syntax and meaning of operands and expressions used in assembly language statements and directives. Operands represent values, registers, or memory locations to be acted on by instructions or directives. Expressions are combinations of operands and arithmetic, logical, bitwise, and attribute operators to calculate a value or memory location that can be acted on by an instruction or directive. Operators indicate what operations will be performed on one or more values in an expression to calculate the value of the expression.

5.2 Operands

An operand is a constant, label, variable, or other symbol that is used in an instruction or directive to represent a value, register, or memory location to be acted on.

The operand types are listed below.

Constant

Direct Memory

Relocatable

Location Counter

Addressing

Register

Structure

Record

5.2.1 Constant Operands

Syntax

number | *string* | *expression*

A constant operand is a number, string constant, symbol, or expression that evaluates to a fixed value. Constant operands, unlike other operands, represent values to be acted on, rather than memory addresses.

In the following examples, note that "COUNT" is a constant only if it was defined with the EQU or equal-sign (=) operator. If "COUNT" is a symbol representing a relocatable value or address, it is not a constant but a direct memory operand.

Examples

```
        mov   bx, 65535/3
        mov   ax, 9
        mov   dx, 987/4
        mov   al, 'c'
        mov   eax, 1048576
        mov   ebx, 65535 * 20
        mov   ecx, COUNT
```

5.2.2 Direct Memory Operands

Syntax

segment : *offset*

A direct memory operand is a pair of segment and offset values that represents the absolute memory address of 1 or more bytes of memory. The *segment* can be a segment register name (CS, SS, DS, ES, FS, or GS), a segment name, or a group name. The *offset* must be an integer, absolute symbol, or expression that evaluates to a value within the range 0 to 64K in 8086 or 286 programs and 0 to 4G in 386 programs.

Examples

```
        mov   dx, SS:0031H
        mov   bx, data:0
        mov   cx, dgroup:block
```

5.2.3 Relocatable Operands

Syntax

symbol

A relocatable operand is any symbol that represents the memory address (segment and offset) of an instruction or data to be acted on. Relocatable operands, unlike direct memory operands, are relative to the start of the segment or group in which the symbol is defined and have no explicit value until the program has been linked.

In the following examples, note that "COUNT" is a relocatable operand only if it was defined with the DW directive. If "COUNT" was defined with the EQU or equal-sign (=) operator, it is a constant.

The size of a *symbol* (16-, 32-, or 48-bits) is derived from the addressing mode. If 16-bit addressing is used, *symbols* are 16-bits. If 32-bit addressing is used, *symbols* are 32-bits.

Examples

```
call    main
mov     bx, loop
mov     cx, offset dgroup:list
mov     ebx, local
mov     ebx, offset dgroup:table
mov     ecx, COUNT
```

5.2.4 Location Counter Operand

Syntax

$

The location counter is a special operand that, during assembly, represents the current location within the current segment. The location counter has the same attributes as a near label. It represents an instruction address that is relative to the current segment. Its offset is equal to the number of bytes generated for that segment to that point. After each statement in the segment has been assembled, the assembler increments the location counter by the number of bytes generated.

In the following example, the location counter forces the assembler to count the total length of a group of declared strings, saving the programmer the trouble of counting each byte.

Example

```
help    db     'Program options:',13,10
f1      db     ' f1   This help screen',13,10
f2      db     ' f2   Save file',13,10
        .
        .
        .
f10     db     ' f10  Exit program',13,10,'$'
distance =     $-help
```

5.2.5 Register Operands

Syntax

register-name

A register operand is the name of a register. Register operands direct instructions to carry out actions on the contents of the given registers. The *register-name* can be any one of the following operands.

Register Operands

EAX	AX	AH	AL	EBX	BX	BH	BL
ECX	CX	CH	CL	EDX	DX	DH	DL
ESI	SI	EDI	DI	EBP	BP	ESP	SP
CS	SS	DS	ES	FS	GS	CR0	CR2
CR3	DR0	DR1	DR2	DR3	DR4	DR5	DR6
DR7	TR6	TR7					

Any combination of upper and lowercase letters is allowed.

The EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP registers are 32-bit general purpose registers. These registers can be used for data or numeric manipulation only with the 386 instruction set. The AX, BX, CX, DX, SI, DI, BP, and SP registers are 16-bit general purpose registers. These registers can be used for data or numeric manipulation with the 8086, 286, or 386 instruction set.

The AH, AL, BH, BL, CH, CL, DH, and DL registers represent the higher 8 bits and lower 8 bits of the AX, BX, CX, and DX registers. These registers offer additional flexibility for data or numeric manipulation. Similarly, they can be used with any instruction set.

The CS, SS, DS, ES, FS, and GS registers are 16-bit segment selector registers that contain the current code, stack, and data segments. The CS register contains the current code segment; The SS register contains the current stack segment; and, the DS, ES, FS, and GS registers contain data segments.

Note

The FS and GS segment registers are accessible only in 386 programs.

The CR0, CR2, CR3, DR0, DR1, DR2, DR3, DR4, DR5, DR6, DR7, TR6, and TR7 registers are 32-bit control, debug, and test registers. These registers can only be used with the 386 instruction set.

5.2.6 Memory Operands

Syntax

base + [*index*] + [*displacement*]

base + [(*index* * *scale*)] + [*displacement*]

Memory operands calculate the effective address. When the 8086 or 286 instruction set is enabled, the effective address is calculated by summing the *base*, *index*, and the *displacement*. When using the 386 instruction set, the effective address is calculated by summing the *base*, *index* multiplied by the *scale*, and the *displacement*.

The *base* can be a register, symbol, or constant. If the 8086 or 286 instruction set is enabled and only one register operand is used, then the *base* register can be any one of the BP, BX, DI, or SI registers. If two register operands are used, then the *base* register can only be the BP or BX register. Although if the 386 instruction set is enabled, then the *base* register can be any 16- or 32-bit general purpose register.

The *index* can be a register or constant. If two-register addressing modes used and the 8086 or 286 instruction set is enabled, then the *index* can be either the DI or SI register. If the 386 instruction set is enabled, then the *index* can be any 16- or 32-bit general purpose register, except the ESP register.

The *scale* is an integer multiplied to the *index*. In 8086 and 286 addressing modes, the *scale* cannot be multiplied to the *index* to find the effect ve address. Although in 386 addressing modes, the *index* can be multiplied by one of the following integer values: 1, 2, 4, or 8.

The *displacement* can be added to both 8086, 286, and 386 addressing mode instructions. In 8086 and 286 addressing modes, the *displacement* can be an 8- or 16-bit integer value. In 386 addressing modes, the *displacement* can be an 8- or 32-bit integer value.

The following examples show addressing modes for the 8086, 286, and 386 instruction sets. The first set is limited to 8086 and 286 addressing modes and the last set is limited to 386 addressing modes.

Examples

```
mov   ax, const              ; 8086 and 286 addressing modes
mov   ax, mem
mov   ax, bx
mov   ax, [ bx ]
mov   ax, [ di ]
mov   ax, [ bx - 32 ]
mov   ax, 32 [ di ]
mov   ax, mem [ si ]
mov   ax, [ si + mem ]
mov   ax, [ bp ] [ di ]
mov   ax, mem [ bx + 5 ] [ si - 2 ]
mov   ax, [ bx + di - 32 ]

mov   eax, const             ; 386 addressing modes
mov   eax, mem
mov   eax, [ eax ]
mov   eax, mem [ eax ]
mov   eax, [ ebp + eax - 64 ]
mov   eax, [ ebp + eax * 4 - 64 ]
mov   eax, mem [ eax * 4 ]
```

5.2.7 Structure Operands

Syntax

variable.field

A structure operand represents the memory address of one member of a structure. The *variable* must be the name of a structure or must be a memory operand that resolves to the address of a structure, and *field* must be the name of a field within that structure. The *variable* is separated from *field* by the structure field-name operator (.), which is described in a following section.

The effective address of a structure operand is the sum of the offsets of *variable* and *field*. The address is relative to the segment or group in which the *variable* is defined.

The first example creates the structure and variable "DATE" and "CURRENT_DATE" , and then puts "31" (the value of "CURRENT_DATE.DAY") in the AX register. The next instruction puts the value "86" in the symbol "CURRENT_DATE.YEAR" .

In the second and third examples, structure operands are used to access values on the stack, first with the 286 instruction set and then with the 386 instruction set.

Note

The following procedures do not conform to the method of passing parameters used in Microsoft high-level languages.

Examples

```
date    struc                       ; First example
        month   dw  ?
        DAY     dw  ?
        YEAR    dw  ?
date    ends

CURRENT_DATE  date <'de','31','63'>
        MOV     AX, CURRENT_DATE.DAY
        mov     CURRENT_DATE.YEAR, '86'
```

```
                                            ; Second example
                .286                        ; Enable 286 instructions
stframe         struc                       ; Stack frame
 retadr         dw     ?                    ; From lowest...
 dest           dw     ?
 source         dw     ?
 nbytes         dw     ?                    ; ...to highest address
stframe ends

copy            proc   near                 ; Push nbytes, source, dest before call
                mov    bx,sp                ; Load stack into base register
                mov    ax,ds
                mov    es,ax                ; (es) = data segment
                mov    di,ss:[bx].dest      ; (di)= destination
                mov    si,ss:[bx].source    ; (si)= source
                mov    cx,ss:[bx].nbytes    ; (cx)= nbytes
                rep    movsb                ; Move bytes from ds:esi to es:di
                ret
copy            endp
```

```
                                            ; Third example
                .386                        ; Enable 386 instructions
s frame struc                               ; Stack frame
 retadr         dd     ?                    ; From lowest
 dest           dd     ?
 source         dd     ?
 nbytes         dd     ?                    ; ...to highest address
stframe ends

copy            proc   near                 ; Push nbytes, source, dest before call
                mov    ebx, csp             ; Load stack into base register
                mov    eax, ds
                mov    es, eax              ; (es)  = data segment
                mov    edi, ss:[ebx].dest   ; (edi) = destination
                mov    esi, ss:[ebx].source ; (esi) = source
                mov    ecx, ss:[ebx].nbytes ; (ecx) = nbytes
                rep    movsb                ; Move bytes from ds:esi to es:edi
                ret
```

5.2.8 Record Operands

Syntax

*variablename*recordtype < *[* initial-value *]*,,, >

A record operand refers to the field values in a record. The *variablename* is the name of the variable containing the record. The *recordtype* is the type and name of the record defined in the source file. The optional *initial-value* is the value of the field in the record. If more than one *initial-value* is given, the values must be separated by commas (,). The *initial-value* is an expression or variahle that evaluates to a constant. The enclosing angle brackets (<>) are required, even if no *initial-value* is given. If no *initial-value* for a field is given, the default value for that field is used. For the example, assume the following record definition:

```
encode        recordhi:4, mid:3, lo:3
```

The examples shows a constant with the value 209 (0D1h) move into the AX register. The following bit diagram illustrates the value.

```
            h i  m i d  l o
 0 0 0 0 0 0 0 0|1 1 0 1 0 0 0 1
 7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0
```

Using record operands is similar to declaring a record and then using the declared data except that, in using record operands, you are using constant data. See the chapter "Types and Declarations" for information on declaring record data.

Example

```
rec1  encode <3,2,1>
      mov   AX, rec1
```

5.2.9 Record Field Operands

Syntax

record-fieldname

The record field operand represents the location of a field in its corresponding record. The operand evaluates to the bit position of the low-order bit in the field and can be used as a constant operand.

The *record-fieldname* must be the name of a previously defined record field. For the example, assume the following record definition and declaration:

```
encode        recordhi:4, mid:3, lo:3
REC1 encode        <9, 7, 4>
```

At this point "REC1" has a value of 636 (27Ch), shown in the bit diagram.

```
               h i  m i d  l o
value 0 0 0 0 0 0 1 0|0 1 1 1 1 1 0 0
bit    7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0
```

The example then copies "6" , the shift count for "HI" , to register CL. The contents of "REC1" are then copied to DX. The shift count of filed three ("HI") is next used to rotate the value of "REC1" so that the value of "HI" is now at the lowest bit. The new value is then put back into "REC1" . At this point "REC1" has a value of 61449 (0F009b), as shown in the bit diagram below.

```
               h i  m i d  l o
value 1 1 1 1 0 0 0 0|0 0 0 0 1 0 0 1
bit    7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0
```

Example

```
        mov    CL, HI
        mov    DX, REC1
        ror    DX, CL
        mov    REC1, DX
```

5.3 Operators and Expressions

An expression is a combination of operands and operators that evaluates to a single value. Operands in expressions can include any of the operands described in this section. The result of an expression can be a value or a memory location, depending on the types of operands and operators used.

MASM provides a variety of operators. Arithmetic, shift, relational, and bitwise operators manipulate and compare the values of operands. Attribute operators manipulate the attributes of operands, such as their type, address, and size.

The first few sections in this chapter describe the arithmetic, relational, and logical operators in detail. Following sections describe attribute operators. In addition to the operators described here, you can use the DUP operator (see the "Types and Declarations" chapter) and the special macro operators (see the "Macros" chapter).

5.3.1 Arithmetic Operators

Syntax

expression1 * *expression2*
expression1 / *expression2*
expression1 MOD *expression2*
expression1 + *expression2*
expression1 – *expression2*
+ *expression*
– *expression*

Arithmetic operators perform common mathematical operations. The following table lists the operators and their function.

Arithmetic Operators

Operator	Function
*	Multiplication.
/	Integer division.
MOD	Remainder after division (modulus).
+	Addition.
–	Subtraction.
+	Positive (unary).
–	Negative (unary).

For all arithmetic operators except + and –, the expressions *expression1* and *expression2* must be integer values. The + operator can be used to add an integer value to a relocatable memory operand. The – operator can be used to subtract an integer number from a relocatable memory operand. The – operator can also be used to subtract one relocatable operand from another, but only if the operands refer to locations within the same segment. The result is an absolute value.

Note

The unary plus and minus (used to designate positive or negative numbers) are not the same as the binary plus and minus (used to designate addition or subtraction). The unary plus and minus have a higher level or precedence, as shown in the section on "Expression Evaluation and Precedence" in this chapter.

Examples

```
14 * 4       ; Equals 56
14 / 4       ; Equals 3
14 MOD 4     ; Equals 2
14 + 4       ; Equals 18
14 - 4       ; Equals 10
14 - +4      ; Equals 10
14 - -4      ; Equals 18
alpha + 5    ; Add 5 to alpha's memory location
alpha - 5    ; Subtract 5 from alpha's memory location
alpha - beta ; Subtract beta's memory location from alpha's
```

5.3.2 SHR and SHL Operators

Syntax

expression SHR *count*
expression SHL *count*

The SHR and SHL operators shift the given *expression* right or left by *count* bits. Depending on the size of the register containing the *expression*, bits can shift off the end of the register. For example, if *count* is greater than or equal to 16 when shifting a 16-bit register, then the result is 0.

Examples

```
mov   AX, 01110111b SHL 3     ; equals 0000001110111000b
mov   AH, 01110111b SHR 3     ; equals 00001110b
```

5.3.3 Relational Operators

Syntax

expression1 EQ *expression2*
expression1 NE *expression2*
expression1 LT *expression2*
expression1 LE *expression2*
expression1 GT *expression2*
expression1 GE *expression2*

The relational operators compare *expression1* and *expression2* and return true (0FFFFH) if the given condition is satisfied, or false (0000H) if it is not. The following table lists the operators and the condition to be satisfied.

Relational Operators

Operator	Condition is satisfied when:
EQ	Operands are equal.
NE	Operands are not equal.
LT	Left operand is less than right.
LE	Left operand is less than or equal to right.
GT	Left operand is greater than right.
GE	Left operand is greater than or equal to right.

Relational operators are typically used with conditional directives and conditional instructions to direct program control.

Examples

```
        1  EQ  0            ; False
        1  NE  0            ; True
        1  LT  0            ; False
        1  LE  0            ; False
        1  GT  0            ; True
        1  GE  0            ; True
```

5.3.4 Bitwise Operators

Syntax

NOT *expression*
expression1 AND *expression2*
expression1 OR *expression2*
expression1 XOR *expression2*

The logical operators perform bitwise operations on expressions. In a bitwise operation, the operation is performed on each bit in an expression rather than on the expression as a whole.

The following table lists the logical operators and their meanings:

Logical Operators

Operator	Meaning
NOT	Inverse.
AND	Boolean AND.
OR	Boolean OR.
XOR	Boolean exclusive OR.

Examples

```
        NOT  11110000b                  ; equals 00001111b
        01010101B  AND  11110000b       ; equals 01010000b
        01010101B  OR   11110000b       ; equals 11110101b
        01010101B  XOR  11110000b       ; equals 10100101b
```

5.3.5 Index Operator

Syntax

[*expression1*] [*expression2*]

The index operator, [ ], adds the value of *expression1* to *expression2*. This operator is identical to the addition (+) operator, except that *expression1* is optional.

If *expression1* is given, the expression must appear to the left of the operator. It can be any integer value, absolute symbol, or relocatable operand. If no *expression1* is given, tbe integer value, 0, is assumed. If *expression1* is a relocatable operand, *expression2* must be an integer value or absolute symbol. Otherwise, *expression2* can be any integer value, absolute symbol, or relocatable operand.

The index operator is typically used to index elements of an array, such as individual characters in a character string.

Examples

```
        mov    al, string[3]         ; Move 4th element of string
        mov    eax, array[4]         ; Move 5th element of array
        mov    string[last], al      ; Move into last element
        mov    cl, dgroup:[1]        ; Move 2nd byte of dgroup
```

5.3.6 PTR Operator

Syntax

type PTR *expression*

The PTR operator forces the variable or label given by the *expression* to be treated as a variable or label having the type given by *type*. The *type* must be one of the following names or values:

BYTE	1
WORD	2
DWORD	4
FWORD	6
QWORD	8
TBYTE	10
NEAR	0FFFFh
FAR	0FFFEh

The *expression* can be any operand. The BYTE, WORD, DWORD, FWORD types can be used with memory operands only. The NEAR and FAR types can be used with labels only.

The PTR operator is typically used with forward references to explicitly define what size or distance a reference has. If it is not used, MASM assumes a default size or distance for the reference. The PTR operator is also used to give instructions access to variables in ways that would otherwise generate errors. For example, you could use the PTR operator to access the high-order byte of a WORD size variable.

Note

The FWORD PTR conversion can be used to do 386 far-indirect calls. For example, the first call is a near call, but the second call is a far call.

```
call array[ebx] callfword ptr [ebx]
```

In the following examples the PTR operator overrides a previous data declaration. The procedure "SUBROUT3" might have been declared NEAR, while "ARRAY" and "FULL_WORD" could have been declared with the DW directive.

Examples

```
call   far PTR SUBROUT3
mov    byte PTR [ARRAY], 1
add    al, byte PTR [FULL_WORD]
```

5.3.7 Segment Override (:) Operator

Syntax

segment-register : *expression*
segment-name : *expression*
group-name : *expression*

The segment override operator (:) forces the address of a given variable or label to be computed using the beginning of the given *segment-register*, *segment-name*, or *group-name*. If a *segment-name* or *group-name* is given, the name must have been assigned to a segment register with a previous ASSUME directive and defined using a SEGMENT or

GROUP directive. The *expression* can be an absolute symbol or relocatable operand. The *segment-register* must be one of CS, SS, DS, ES, FS, or GS.

By default, the effective address of a memory operand is computed relative to the DS, SS, or CS register, depending on the instruction and operand type. Similarly, all labels are assumed to be NEAR. These default types can be overridden using the segment override operator.

Examples

```
mov   _text:far_label, eax
mov   eax, dgroup:variable
mov   al, CS:0001H
```

5.3.8 Structure Field-Name Operator

Syntax

symbol.field

The structure field-name operator (.) is used to designate a field within a structure. The *symbol* is an symbol operand of the structure type and *field* is the name of a field within the structure. This operator is equivalent to the addition operator (+).

Example

```
inc   month.day
mov   time.min, 0
mov   [bx].dest, 0
mov   [eax].dest, 0
```

5.3.9 SHORT Operator

Syntax

SHORT *label*

The SHORT operator sets the type of the given *label* to SHORT. Short labels can be used in jump instructions whenever the distance from the label to the instruction is not more than 127 bytes.

Instructions using short labels are 1 byte smaller than identical instructions using near labels.

Note

In 386 programs, use the SHORT operator on conditional jumps less than 128 bytes.

Example

```
    jmp    SHORT do_again   ; Jump less than 128 bytes
```

5.3.10 THIS Operator

Syntax

THIS *type*

The THIS operator creates an operand whose offset and segment value are equal to the current location counter value and whose type is given by *type*. The *type* can be any one of the following:

NEAR

BYTE

DWORD

QWORD

FAR

WORD

FWORD

TBYTE

The THIS operator is typically used with the EQU or equal-sign (=) directive to create labels and symbols. This is similar to using the LABEL directive to create labels and symbols.

In the following, the first and second examples are equivalent in operation. Similarly, the third and fourth examples are also equivalent.

Examples

```
tag   EQU  THIS BYTE
tag   LABEL     BYTE
check =    this near
check label near
```

5.3.11 HIGH and LOW Operators

Syntax

HIGH *expression*
LOW *expression*

The HIGH and LOW operators return the high and low 8 bits of the given *expression*. The HIGH operator returns the high 8 bits of the *expression*; the LOW operator returns the low-order 8 bits. The *expression* can be any value.

Examples

```
mov   ah, HIGH word_value
mov   al, LOW 0ffffh
```

5.3.12 SEG Operator

Syntax

SEG *name*

The SEG operator returns the segment value of the given *name*. The *name* can be any label, segment name, group name, or symbol.

Example

```
mov   eax, SEG variable_name
mov   eax, SEG label_name
```

5.3.13 OFFSET Operator

Syntax

OFFSET *name*

The OFFSET operator returns the offset of the given *name*. The *name* can be any label, segment name, or symbol. The returned value is tbe number of bytes between the item and the beginning of the segment in which it is defined. For a segment name, the return value is the offset from the start of the segment to the most recent byte generated for that segment.

The segment override operator (:) can be used to force OFFSET to return the number of bytes between the item in the *name* and the beginning of a named segment or group. This is the method used to generate valid offsets for items in a group. The returned value is always a relative value that is subject to change by the linker when the program is actually linked.

Examples

```
mov   bx, OFFSET subrout3
mov   ebx, OFFSET subrout3
mov   ebx, OFFSET dgroup:array
```

5.3.14 TYPE Operator

Syntax

TYPE *name*

The TYPE operator returns a number representing the type of the given *name*. If the *name* is a variable, the operator returns the size of the operand in bytes. If the *name* is a label, the operator returns 0FFFFH if the label is NEAR, and 0FFFEH if the label is FAR.

As in the second example, note that the return value can be used to specify the type for a PTR operator.

Examples

```
mov   eax, TYPE array
jmp   (TYPE get_loc) PTR destiny
```

5.3.15 .TYPE Operator

Syntax

.TYPE *name*

The .TYPE operator returns a byte that defines the mode and scope of the given *name*. If the *name* is not valid, .TYPE returns 0.

The next table lists the variable's attributes as returned in bits 0, 1, 5, and 7.

.TYPE Operator and Variable Attributes

Bit Position	If Bit=0	If Bit=1
0	Absolute	Program related
1	Not Data related	Data related
5	Not defined	Defined
7	Local scope	External scope

If both the scope bit and defined bit are zero, the *name* is not valid.

The .TYPE operator is typically used with conditional directives, where an argument may need to be tested to make a decision regarding program flow.

The following example sets "Z" to 22h (00100010b). Bit 0 is not set in "Z" because "X" is not program-related. Bit 1 is set because "X" is data-related. Bit 5 is set because "X" is defined. Bit 7 is not set because "X" is local. The remaining bits are never set.

Examples

```
X      db     12
Z      equ    .TYPE X
```

5.3.16 LENGTH Operator

Syntax

LENGTH *symbol*

The LENGTH operator returns the number of BYTE, WORD, DWORD, QWORD, or TBYTE elements in the given *symbol*. The size of each element depends on the *symbol*'s defined type.

Only symbols that have been defined using the DUP operator return values greater than 1. The return value is always the number that precedes the first DUP operator.

In the following examples, both LENGTH operators return 100 to the CX register. The return value does not depend on any nested DUP operators.

Examples

```
array  dw   100  DUP(1)
table  dw   100  DUP(1,10 DUP(?))
       mov  CX, LENGTH array
       mov  CX, LENGTH table
```

5.3.17 SIZE Operator

Syntax

SIZE *symbol*

The SIZE operator returns the total number of bytes allocated for the *symbol*. The returned value is equal to the value of LENGTH times the value of TYPE.

In the following example, SIZE returns 200 to the EBX register.

Example

```
array dw   100  dup(1)
      mov  EBX, SIZE array
```

5.3.18 WIDTH Operator

Syntax

WIDTH *record-fieldname* | *record*

The WIDTH operator returns the width (in bits) of the given record field or record. The *record-fieldname* must be the name of a record defined in a field. The *record* must be the name of a record.

The field name represents the bit count. The "FIELD1" field equals 13 (the width of "FIELD2" plus the width of "FIELD3") while the "WIDTH" of "FIELD1" equals 3.

Examples

```
rtype  record field1:3,field2:6,field3:7
rec1   rtype  <>
wid1   =      WIDTH FIELD1    ; Equals 3
wid2   =      WIDTH FIELD2    ; Equals 6
wid3   =      WIDTH FIELD3    ; Equals 7
wid4   =      WIDTH rtype     ; Equals 16
```

5.3.19 MASK Operator

Syntax

MASK *record-fieldname* | *record*

The MASK operator returns a bit mask for the bit positions in a record occupied by the given record field. A bit in the mask contains a 1 if that bit corresponds to a record bit. The *record-fieldname* must be the name of a record field. All other bits contain 0.

Examples

```
rtype  recordfield1:3,field2:6,field3:7
rec1   rtype  <>
m1     =      MASK field1; Equals E000H (1110000000000000b)
m2     =      MASK field2; Equals 1F80H (1111110000000b)
m3     =      MASK field3; Equals 003FH (1111111b)
m4     =      MASK rtype ; Equals 0FFFFH(1111111111111111b)
```

5.3.20 Expression Evaluation and Precedence

Expressions are evaluated according to the rules of operator precedence and order. Operations of highest precedence are performed first. Operations of equal precedence are performed from left to right. This default order of evaluation can be overridden using enclosing parentheses. Operations in parentheses are always performed before any adjacent operations. The following table lists the precedence of all operators. Operators on the same line have equal precedence.

Operator Precedence

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

Examples

```
8 / 4 * 2               ; Equals 4
8 / (4 * 2)             ; Equals 1
8 + 4 * 2               ; Equals 16
(8 + 4) * 2             ; Equals 24
8 EQ 4 AND 2 LT 3       ; Equals 0000H (false)
8 EQ 4 OR 2 LT 3        ; Equals 0FFFFH (true)
```

5.4 Forward References

Although MASM permits forward references to labels, segment names, and symbols, such references can lead to assembly errors if not used properly. A forward reference is any use of a name before it has been declared. For example, in the JMP instruction below, the label "TARGET" is a forward reference.

```
        JMP   TARGET
        mov   ax, 0
TARGET:
```

Whenever MASM encounters an undefined name in Pass 1, it assumes that the name is a forward reference. If only a name is given, MASM makes assumptions about that name's type and segment register, and uses these assumptions to generate code or data for the statement. For example, in the JMP instruction above, MASM assumes that "TARGET" is an instruction label having NEAR type. If the 8086 or 286 instruction set is enabled, MASM generates 3 bytes of instruction code for the instruction. Otherwise, MASM generates 5 bytes of code for the 386 instruction set.

MASM bases its assumptions on the statement containing the forward reference. Errors can occur when these assumptions are incorrect. For example, if "TARGET" were really a FAR label and not a NEAR label, the assumption made by MASM in Pass 1 would cause a phase error. In other words, MASM would generate 5 (7) bytes of instruction code for the JMP instruction in Pass 2 but only 3 (5) in Pass 1.

To avoid errors with forward references, the segment override (:), PTR, and SHORT operators should be used to override the assumptions made by MASM whenever necessary. The following guidelines list when these operators should be used.

- If a forward reference is a variable that is relative to the DS, SS, or CS register, then use the segment override operator (:) to specify the variable's segment register, segment, or group.

  Example

  ```
  mov   eax, SS:stacktop
  inc   data:time[1]
  add   eax, dgroup:_I
  ```

  If the segment override operator is not used, MASM assumes that the variable is DS relative.

- If a forward reference is an instruction label in a JMP instruction, then use the SHORT operator if the instruction is less than 128 bytes from the point of reference.

  Example

  ```
  jmp   SHORT target
  ```

  If SHORT is not used, MASM assumes that the instruction is greater than 128 bytes away. This does not cause an error, but it does cause MASM to generate an extra, and unnecessary, NOP instruction.

Note

In the event a NOP instruction is generated, MASM generates a warning message.

- If a forward reference is an instruction label in a CALL or JMP instruction, then use the PTR operator to specify the label's type.

Examples

```
                CALLFAR PTR print
                JMP   NEAR PTR exit
```

MASM assumes that the label has NEAR type, so PTR need not be used for NEAR labels. If the label has FAR type, however, and PTR is not used, a phase error will result.

- If the forward reference is a segment name with a segment override operator (:), use the GROUP statement to associate the segment name with a group name, then use the ASSUME statement to associate the group name with a segment register.

Example

```
dgroup          segment stack
                ASSUME    ss: dgroup

code            segment
                .
                .
                .
                mov   ax, stack:stacktop
                .
                .
                .
```

If you do not associate a group with the segment name, MASM may ignore the segment override and use the default segment register for the variable. This usually results in a phase error in Pass 2.

5.5 Strong Typing for Memory Operands

MASM carries out strict syntax checks for all instruction statements, including strong typing for operands that refer to memory locations. This means that any relocatable operand used in an instruction that operates on an implied data type must either have that type, or have an explicit type override (PTR operator).

For example, in the following program segment, the variable "STRING" is incorrectly used in an move instruction.

```
string db     "A message."
       mov   eax, STRING[1]
```

The preceding statement will create an "Operand types must match" error since "STRING" has BYTE type and the instruction expects a variable having WORD type.

To avoid this error, the PTR operator must be used to override the variable's type. The following statement will assemble correctly and execute as expected.

```
        mov   eax, dword PTR STRING[1]
```

Chapter 6
Global Declarations

6.1 Introduction

The global-declaration directives allow you to define labels, and symbols that can be accessed globally, that is, from all modules in a program. Global declarations transform "local" symbols (labels, variables, and other symbols that are specified to the source files in which they are defined) into "global" symbols that are available to all other modules of the program.

The two global-declaration directives are PUBLIC and EXTRN. The PUBLIC directive is used in public declarations, which transform a locally defined symbol into a global symbol, making it available to other modules. The EXTRN directive is used in external declarations, making a global symbol's name and type known in a source file so that the global symbol be used in that file. Every global symbol must have a public declaration in exactly one source file of the program. A global symbol can have external declarations in any number of other source files.

6.2 PUBLIC Directive

Syntax
PUBLIC *name*,,,

The PUBLIC directive makes the label or absolute symbol specified by *name* available to all other modules in the program. The *name* must be the name of a label or absolute symbol defined within the current source file. Absolute symbols, if given, can only represent 1- or 2-byte integer or string values.

MASM converts all lowercase letters in *name* to uppercase before copying the name to the object file. The **-Ml** and **-Mx** options can be used in MASM command lines to direct MASM to preserve lowercase letters when copying to the object file.

The values declared public in this example include an absolute symbol, a variable, a label, and a procedure.

Example

```
        PUBLIC     true, test, start
true    =      0ffffh
test    db     1
start   label  far
clear   proc   near
```

6.3 EXTRN Directive

Syntax

EXTRN *name:type* ,,,

The EXTRN directive defines an external variable, label, or symbol of the specified *name* and *type*. An external item is any variable, label, or symbol that has been declared with a PUBLIC directive in another module of the program.

The *type* must match the type given to the item in its actual definition. It can be any one of the following:

BYTE
DWORD
QWORD
NEAR
ABS
WORD
FWORD
TBYTE
FAR

The ABS type is reserved for symbols that represent absolute numbers.

Although the actual address is not determined until the object files are linked, the assembler may assume a default segment for the external item, based on where the EXTRN directive is placed in the module. If the directive is placed inside a segment, the external item is assumed to be relative to that segment. And, the item's public declaration (in some other module) must be in a segment having the same name and attributes. If the directive is outside all segments, no assumption is made about what segment the item is relative to, and the item's public declaration can be in any segment in any module. In either case, the segment override operator (:) can be used to override the defaults segment of an external variable or label.

Examples

```
EXTRN      tagn:NEAR            ; 8086 or 286 instruction
EXTRN      var1:WORD, var2:DWORD       ; 8086 or 286 instruction
EXTRN      tagn:NEAR                   ; 386 instruction
EXTRN      var1:DWORD, var2:FWORD  ; 386 instruction
```

6.4 Program Example

The following source files illustrate a program that uses public and external declarations to access instruction labels. Each program consists of two modules, named "STARTMOD" and "PRINTMOD" . The "STARTMOD" module is the program's main module. The first two modules are written with the 286 instruction set and the last two modules are written with the 386 instruction set.

Execution starts at the instruction labeled "_MAIN" in "STARTMOD" , and passes execution to the instruction labeled "_PRINT" in "PRINTMOD" where the "_PRINTF" routine is called to print the message "Hello." at the system console. Execution then returns to the instruction labeled "_FINISH" in "STARTMOD" .

The "STARTMOD" files publicly declare two symbols, "_MAIN" and "_FINISH" , making the symbols available to the other source file in the program. Both of these symbols are locally defined as instruction labels later in the source file, and therefore can be used as instruction labels in the other source file. The "STARTMOD" files also contain an external declaration of the symbol "_PRINT" . This declaration defines "_PRINT" to be a near label and is assumed to have been publicly declared in the other source file. The label is used in a JMP instruction given later in the file.

The "PRINTMOD" files contain a public declaration of the symbol "_PRINT" and an external declaration of the symbol "_FINISH" . In these cases, "_PRINT" is locally defined as a near label and matches the external declaration given to it in "STARTMOD" . The symbol "_FINISH" is declared to be a near label, matching its definition in "STARTMOD" .

Before the programs are executed, the source files must be assembled individually, then linked together using the system linker.

16-Bit Mode Startmod Module:

```
        .286                        ; Enable 286 instructions
        name        STARTMOD
        public      _MAIN, _FINISH
        extrn       _exit:near
        extrn       _PRINT:near

_data   segment word public 'data'
_data   ends
dgroup          group       _data

_text   segment byte public 'code'
        assume cs:_text,ds:dgroup

_MAIN:
        jmp _PRINT
_FINISH:
        push        0               ;exit(0)
        call _exit
_text   ends
        end
```

16-Bit Mode Printmod Module:

```
        .286                ; Enable 286 instructions
        name        PRINTMOD
        public      _PRINT
        extrn       _FINISH:near
        extrn       _PRINTF:near

_data   segment     word public 'data'
string  db  "Hello.", 10, 0
_data   ends
dgroup          group _data

_text   segment byte public 'code'
        assume      cs:_text, ds:dgroup
_PRINT:
        push        offset dgroup:string
        call _PRINTF
        add sp, 2
        jmp _FINISH
_text   ends
        end
```

32-Bit Mode Startmod Module:

```
        .386                    ; Enable 386 instructions
        name    STARTMOD
        public  _MAIN, _FINISH
        extrn   _exit:near
        extrn   _PRINT:near

_data   segment word public 'data'
_data   ends
dgroup      group       _data

_text   segment byte public 'code'
        assume cs:_text,ds:dgroup

_MAIN:
        jmp _PRINT
_FINISH:
        push    0               ;exit(0)
        call _exit
_text   ends
        end
```

32-Bit Mode Printmod Module:

```
        .386                    ; Enable 386 instructions
        name    PRINTMOD
        public  _PRINT
        extrn   _FINISH:near
        extrn   _PRINTF:near

_data   segment word public 'data'
string  db  "Hello.", 10, 0
_data   ends
dgroup      group       _data

_text   segment byte public 'code'
        assume  cs:_text, ds:dgroup
_PRINT:
        push    offset dgroup:string
        call _PRINTF
        add esp, 4
        jmp _FINISH
_text   ends
        end
```

Chapter 7
Conditional Assembly

7.1 Introduction

MASM provides conditional directives to test blocks of statements within source files for assembly-time conditions. The conditional directives include the following:

IF

IFE

IF1

IF2

IFDEF

IFNDEF

IFB

IFNB

IFIDN

IFDIF

ELSE

ENDIF

The IF directives and the ENDIF and ELSE directives can be used to enclose statements considered for conditional assembly. The conditional block takes the form:

```
IF
    statements
ELSE
    statements
ENDIF
```

The *statements* following IF can be any valid statements, including other conditional blocks. The ELSE directive and its *statements* are optional. ENDIF ends the block.

The statements in the conditional block are assembled only if the condition specified by the corresponding IF directive is satisfied. If the conditional block contains an ELSE directive, only the statements up to the ELSE directive will be assembled. The statements following the ELSE directive are assembled only if the IF condition is not met. An ENDIF directive must mark the end of the conditional block. No more than one ELSE for each IF directive is allowed.

IF directives can be nested up to 255 levels. To avoid ambiguity, a nested ELSE directive always belongs to the nearest preceding IF directive that does not have its own ELSE.

7.2 IF and IFE Directives

Syntax

IF *expression*
ENDIF

IFE *expression*
ENDIF

The IF and IFE directives test the value of *expression*. The IF directive grants assembly if *expression* is true (nonzero). The IFE directive grants assembly if *expression* is false (0). The *expression* must resolve to an absolute value and must not contain forward references.

In the following example, the symbols within the block will only be declared external if the symbol "DEBUG" evaluates to true (nonzero).

Example

```
IF DEBUG
        extrn dump:far
        extrn trace:far
        extrn breakpoint:far
ENDIF
```

7.3 IF1 and IF2 Directives

Syntax

```
IF1
    statements
ENDIF
```

```
IF2
    statements
ENDIF
```

The IF1 and IF2 directives test the current assembly pass of the assembler and grants assembly of *statements* only when the corresponding pass occurs. The IF1 directive grants assembly of *statements* on only Pass 1. The IF2 directive grants assembly of *statements* on only Pass 2. This directive does not require any arguments.

Example

```
IF1
        %out Pass 1 Starting
ELSE
        %out Pass 2 Starting
ENDIF
```

7.4 IFDEF and IFNDEF Directives

Syntax

```
IFDEF  symbol
    statements
ENDIF
```

```
IFNDEF  symbol
    statements
ENDIF
```

The IFDEF and IFNDEF directives test whether or not the given *symbol* has been defined. The IFDEF directive grants assembly if *symbol* has been defined. The IFNDEF directive grants assembly if *symbol* has not yet been defined.

The *symbol* can be any valid name. Note that if *symbol* is a forward reference, it is considered undefined on Pass 1, but defined on Pass 2.

In the following example, "BUF1" is allocated only if "BUFFER" has been defined.

Example

```
IFNDEF     BUFFER
BUF1 db    10 dup(?)
ENDIF
```

7.5 IFB and IFNB Directives

Syntax

```
IFB <argument>
     statements
ENDIF
```

```
IFNB <argument>
     statements
ENDIF
```

The IFB and IFNB directives test *argument*. The IFB directive grants assembly if *argument* is blank. The IFNB directive grants assembly if *argument* is not blank. The *argument* can be any expression. The angle brackets (< >) are required.

The IFB and IFNB directives are intended for use in macro definitions. They can control conditional assembly of statements in the macro, based on the parameters passed in the macro call. In such cases, *argument* should be one of the dummy parameters listed by the MACRO directive.

In the following example, "PUSHALL" is a recursive macro that continues to call itself until it encounters a blank argument. Any register or list of registers (consisting of up to six registers) can be passed to the macro.

Examples

```
PUSHALL  MACRO    reg1,reg2,reg3,reg4,reg5,reg6
         IFNB     <reg1>            ;; if parameter not blank
                  push   reg1   ;; push one register and repeat
                  PUSHALL   reg2,reg3,reg4,reg5,reg6
         ENDIF
         ENDM

PUSHALL  eax,ebx,esi,eds
PUSHALL  cs,es
```

7.6 IFIDN and IFDIF Directives

Syntax

IFIDN *<argument1>*, *<argument2>*
    statements
ENDIF

IFDIF *<argument1>*, *<argument2>*
    statements
ENDIF

The IFIDN and IFDIF directives compare *argument1* and *argument2*. The IFIDN directive grants assembly if the arguments are identical. The IFDIF directive grants assembly if the arguments are different. The arguments can be any expressions. To be identical, each character in *argument1* must match the corresponding character in *argument2*. The angle brackets (< >) are required. The arguments must be separated by a comma (,).

The IFIDN and IFDIF directives are intended to be used in macro definitions. They can be used to control conditional assembly of statements based on the parameters passed in the macro call. In such cases, the arguments should be dummy parameters listed by the MACRO directive.

In this example, a macro uses the IFDIF directive to check against dividing by a constant that evaluates to 0. The macro is then called, using a percent sign (%) on the second parameter so that the value of the parameter, rather than its name, will be evaluated.

Example

```
divide MACRO    numerator, denominator
       IFDIF    <denominator>,0       ;; if not dividing by zero
       mov      eax, numerator        ;;    divide eax by ebx
       mov      ebx, denominator
       div      ebx                   ;; result in accumulator
       ENDIF
       ENDM

divide 6,%test
```

Chapter 8
Macros

8.1 Introduction

This chapter explains how to create and use macros in your source files. It discusses the macro directives and the special macro operators. Since macros are closely related to conditional directives, you may need to review the previous chapter in order to completely understand examples in this chapter.

Macro directives enable you to write a named block of source statements, then use that name in you source file to represent the statements. During assembly, MASM automatically replaces each occurrence of the macro name with the statements in the macro definition. You can place a block of statements anywhere in your source file any number of times by simply defining a macro block once, then inserting the macro name at each location where you want the macro block to be assembled. You can also pass parameters to macros.

A macro can be defined any place in the source file as long as the definition precedes the first source line that calls that macro. Macros can be kept in a separate file and made available to the program through an INCLUDE directive.

Often a task can be done by either a macro or procedure. For example, the "ADDUP" procedure does the same thing as the "ADDUP" macro. Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if called repeatedly. Procedures take up less space, but the increased overhead of saving and restoring addresses and parameters can make programs execute slower.

8.2 Macro Directives

The MASM assembler accepts the following macro directives:

MACRO

LOCAL

PURGE

REPT

IRP

IRPC

EXITM

ENDM

The MACRO and ENDM directives designate the beginning and end of a macro. The LOCAL directive defines labels used only within a macro, and the PURGE directive deletes previously defined macros. The EXITM directive exits from a macro before all the statements in the macro are expanded.

The REPT, IRP, and IRPC directives create contiguous blocks of statements within a macro. To control the number of repetitions of code, specify a number; or allow each block of code to be repeated once for each parameter in a list; or by having the block repeated once for each character in a string.

8.3 MACRO and ENDM Directives

Syntax

name MACRO [*dummy-parameter*,,,]
  statements
  ENDM

The MACRO and ENDM directives create a macro called *name* that contains *statements*.

The *name* must be a valid symbol and must be unique. It is used in the source file to invoke the macro. The *dummy-parameter* is a name that acts as a placeholder for values to be passed to the macro when it is called. Any number of *dummy-parameters* can be given, but they must all fit on one line. If you give more than one, you must separate them with commas (,). The *statements* are any valid MASM statements, including other MACRO directives. Any number of statements can be used. The dummy parameter can be used any number of times in these statements.

A macro is called any time its name appears in a source file (macro names in comments are ignored). MASM copies the statements in the macro definition to the point of call, replacing any dummy parameters in these statements with actual parameters passed in the call.

Macro definitions can he nested. This means a macro can be defined within another macro. MASM does not process nested definitions until the outer macro has been called. Therefore, nested macros cannot be called until the outer macro has be called at least once. Macro definitions can be nested to any depth. Nesting is limited only by the amount of memory available when the source file is assembled.

Macro definitions can contain calls to other macros. These nested macro calls are expanded like any other macro call, but only when the outer macro is called. Macros can also be recursive, they can call themselves.

Note

You must be careful when using the word MACRO after the TITLE, SUBTTL, and NAME directives. Since tbe MACRO directive overrides these directives, placing the word immediately after these directives causes MASM to begin to create macros named TITLE, SUBTTL, and NAME. To avoid this problem, you should alter the word MACRO in some way when using it in a title or name. For example, add a hyphen to the word, "- MACRO." The following example defines the macro "ADDUP" , which uses three *dummy-parameters* to add three values and leave their sum in the EAX register. The three *dummy-parameters* will be replaced with actual values when the macro is called.

MASM assembles the statements in the macro only if the macro is called, and only at the point in the source file from which it is called. Thus, all addresses in the assembled code are relative to the macro call, not the macro definition. The macro definition itself is never assembled.

Example

```
ADDUP   MACRO   ad1, ad2, ad3
        mov     EAX, ad1
        add     EAX, ad2
        add     EAX, ad3
        ENDM
```

8.4 Macro Calls

Syntax

name [*actual-parameter,,,*]

A macro call directs MASM to copy the statements of the macro *name* to the point of call and to replace any dummy parameters in these statements with the corresponding *actual-parameters*. The *name* must be the name of a macro defined earlier in the source file. The *actual-parameter* can be any expression. Any number of actual parameters can be given, but they must all fit on one line. Multiple parameters must be separated with commas, spaces, or tabs.

MASM replaces the first dummy parameter with the first actual parameter, the second with the second, and so on. If a macro call has more actual parameters than dummy parameters, the extra actual parameters are ignored. If a call has fewer actual parameters, any remaining dummy parameters are replaced with a null (blank) string.

If you wish to pass a list of values as a single actual parameter, you must place angle brackets (< >) around the list. The items in the list must be separated by commas (,).

In the following examples, the first macro call passes five numeric parameters to the macro "ALLOCBLOCK" . The second macro call passes one parameter to "ALLOCBLOCK" . This parameter is a list of five numbers. The last macro call passes three parameters to the macro "ADDUP" . MASM replaces the corresponding dummy parameters with exactly what is typed in the macro call parameters. Assuming that "ADDUP" is the same macro defined in the "Macros" chapter, the assembler would expand the macro to the following code:

```
        mov   eax, ebx,
        add   eax, 2
        add   eax, count
```

Examples

```
ALLOCBLOCK 1, 2, 3, 4, 5
ALLOCBLOCK <1, 2, 3, 4, 5>
ADDUP      ebx, 2, count
```

8.5 LOCAL Directive

Syntax

LOCAL *dummy-name*,,,

The LOCAL directive creates unique names for use in macros. The *dummy-name* is a name for a placeholder that is to be replaced by an unique name when the macro is expanded. At least one *dummy-name* is required. If you give more than one, you must separate the names with commas (,). A dummy name can be used in any statement within the macro.

MASM creates a new name for a dummy each time the macro is expanded. The actual name has the form:

??number

The *number* is a hexadecimal number in the range 0000 to FFFF. Do not give other symbol names in this format, since doing so will produce a label or symbol with multiple definitions. In listings, the *dummyname* is shown in the macro definition, but the actual names are shown for each expansion of the macro.

The LOCAL directive is typically used to create an unique label that will be only used in a macro. Normally, if a macro containing a label is used more than once, MASM will display an error message indicating the file contains a labeled or symbol with multiple definitions, since the same label will appear in both expansions. To avoid this problem, all labels in macros should be *dummy-names* declared with the LOCAL directive .

In the following example, the LOCAL directive defines the *dummy-name*'s "AGAIN" and "GOTZERO" . These names will be replaced with unique names each time the macro is expanded. For example, the first time the macro is called, "AGAIN" will be assigned the name "??0000" and "GOTZERO" will be assigned "??0001" . The second time through, "AGAIN" will be assigned "??0002" and "GOTZERO" will be assigned "??0003" , and so on.

Example

```
power           macro     factor, exponent
                LOCAL     AGAIN, GOTZERO
                mov       cx, exponent
                mov       ax, 1
                jcxz      GOTZERO
                mov       bx, factor
AGAIN:          mul       bx
                loop      bx
GOTZERO:
                endm
```

8.6 PURGE Directive

Syntax
PURGE *macro-name* ,,,

The PURGE directive deletes the current definition of the macro called *macro-name*. Any subsequent call to that macro causes MASM to generate an error.

The PURGE directive is intended to clear memory space no longer needed by a macro. If the *macro-name* is an instruction or directive mnemonic, the directive restores its previous meaning.

The PURGE directive is often used with a macro library to let you choose those macros from the library you really need in your source file. A macro library is simply a file containing macro definitions. You add this library to your source file using the INCLUDE directive, then remove unwanted definitions using the PURGE directive.

It is not necessary to PURGE a macro before redefining it. Any redefinition of a macro automatically purges the previous definition. Also, any macro can purge itself as long as the PURGE directive is on the last line of the macro.

In the following examples, the first PURGE directive deletes the macro named "ADDUP", and the second PURGE directive deletes the macros named "MAC1", "MAC2", and "MAC9".

Examples

```
        PURGE     ADDUP
        PURGE     MAC1, MAC2, MAC9
```

8.7 REPT and ENDM Directives

Syntax

REPT *expression*
    statements
ENDM

The REPT and ENDM directives define a block of *statements* that are to be repeated *expression* number of times. The *expression* must evaluate to a 16-bit unsigned number, and must not contain external or undefined symbols. The *statements* can be any valid statements.

The following example repeats the equal-sign (=) and DB directives 10 times. The resulting statements create 10 bytes of data whose values range from 1 to 10.

Example

```
x       =       0
        rept    10
x       =       x+1
        db      x
        endm
```

8.8 IRP and ENDM Directives

Syntax

IRP *dummy-name*, <*parameter*,,, >
    statements
ENDM

The IRP and ENDM directives designate a block of *statements* that are to be repeated once for each *parameter* in the list enclosed by angle brackets (< >). The *dummy-name* is a name for a placeholder to be replaced by the current *parameter*. The *parameter* can be any legal symbol. Any number of parameters can be given. If you give more than one parameter, you must separate them with commas (,). The *statements* can be any valid assembler statements. The *dummy-name* can be used any number of times in these statements.

When MASM encounters an IRP directive, it makes one copy of the statements for each parameter in the enclosed list. While copying the statements, it replaces all occurrences of the *dummy-name* in these statements. If a null parameter (<>) is found in the list, the *dummy-name* is replaced with a null value. If the parameter list is empty, the IRP directive is ignored and no statements are copied.

Assume an IRP directive is used inside a macro definition and the parameter list of the IRP directive is also a dummy parameter of the macro. In this case, you must enclose that dummy parameter within angle brackets. For example, in the following macro definition, the dummy parameter "X" is used as the parameter list for the IRP directive:

```
alloc   macro   X
        IRP     y, <X>
        db      y
        ENDM
        endm
```

If this macro is called with the following line of code, then the macro expansion becomes the following second, third and fourth lines of code.

```
alloc       <0, 1, 2, 3, 4, 5, 6, 7, 8, 9>

    IRP         y, <0, 1, 2, 3, 4, 5, 6, 7, 8, 9>
    db    y
    ENDM
```

The macro removes the brackets from the actual parameter before replacing the dummy parameter. You must provide the angle brackets for the parameter list yourself.

In the following example, the DB directive is repeated 10 times, duplicating the numbers in the list once for each repetition. The resulting statements create 100 bytes of data with the values 0 through 9 duplicated 10 times.

Example

```
    IRP         x, <0, 1, 2, 3, 4, 5, 6, 7, 8, 9,>
    db          10 dup (x)
    ENDM
```

8.9 IRPC and ENDM Directives

Syntax

```
IRPC  dummy-name,string
      statements
ENDM
```

The IRPC and ENDM directives enclose a block of *statements* that are repeated once for each character in the *string*. The *dummy-name* is a name for a placeholder to be replaced by the current character in the *string*. The *string* can be any combination of characters in the MASM character set. The string should be enclosed with angle brackets (< >) if it contains spaces, commas, or other separating characters. The *statements* can be any valid assembler statements.

When MASM encounters an IRPC directive, it makes one copy of the statements for each character in the string. While copying the statements, it substitutes the current character for all occurrences of the *dummy-name* in these statements.

In the following example, the DB directive is repeated 10 times, once for each character in the string "0123456789" . The resulting statements create 10 bytes of data having the values 1 through 10.

Example

```
IRPC    cx, 0123456789
db      x+1
ENDM
```

8.10 EXITM Directive

Syntax

```
EXITM
```

The EXITM directive directs MASM to terminate macro or repeat block expansion and continue assembly with the next statement after the macro call or repeat block. The directive is typically used with IF directives to allow conditional expansion of the last statements in a macro or repeat block.

When an EXITM is encountered, MASM exits the macro or repeat block immediately. Any remaining statements in the macro or repeat block are not processed. If EXITM is encountered in a macro or repeat block nested in another macro or repeat block, MASM returns to expanding the outer level block.

The following example defines a macro that creates no more than 255 bytes of data. The macro contains an IFE directive that checks the expression "X-0FFH." When this expression is 0 (x equal to 255), the EXITM directive is processed and expansion of the macro stops.

Example

```
alloc   macro    times
x        =       0
        rept     times          ;; Repeat up to 256 times
        IFE      X-0FFH         ;; Does x = 255 yet?
                 EXITM          ;; If so, quit
        ELSE
                 db      x      ;; Else allocate x
        ENDIF
x =     x+1                     ;; Increment x
        endm
        endm
```

8.11 Macro Operators

The macro and conditional directives use the following special set of macro operators:

&	Substitute operator
<>	Literal-text operator
!	Literal-character operator
%	Expression operator
;;	Macro comment

When used in a macro definition or a conditional-assembly directive, these operators carry out special control operations, such as text substitution.

8.11.1 Substitute Operator

Syntax

&dummy-parameter
or
dummy-parameter&

The substitute operator (&) forces MASM to replace the given *dummy-parameter* with its corresponding actual parameter value. The operator is used anywhere a *dummy-parameter* immediately precedes or follows other characters, or whenever the parameter appears in a quoted string.

The following example replaces "&X" with the value of the actual parameter passed to the macro "ERRGEN" . If the macro is called with the statement

```
errgen 1, wait
```

the macro is expanded to

```
errorwait    db     'Error 1 - wait'
```

Example

```
ERRGEN   macro   y,X
error&X  db      'Error &y - &X'
         endm
```

Note

For complex, nested macros, you can use extra ampersands (&) to delay the actual replacement of a dummy parameter. In general, you need to supply as many ampersands as there are levels of nesting.

For example, in the following macro definition, the substitute operator is used twice with "Z" to make sure its replacement occurs while the IRP directive is being processed.

```
alloc   macro   X
        IRP     Z,<1,2,3>
        X&&Z    db
        ENDM
        endm
```

In this example, the dummy parameter "X" is replaced immediately when the macro is called. The dummy parameter "Z" , however, is not replaced until the IRP directive is processed. This means the parameter is replaced once for each number in the IRP parameter list. If the macro is called with

```
alloc var
```

the expanded macro will be

```
var1  db   1
var2  db   2
var3  db   3
```

8.11.2 Literal Text Operator

Syntax

<*text*>

The literal text operator directs MASM to treat *text* as a single literal regardless of whether it contains commas, spaces, or other separators. The operator is most often used with macro calls and the IRP directive to ensure that values in a parameter list are treated as a single parameter.

The literal text operator can also be used to force MASM to treat special characters such as the semicolon (;) or ampersand (&) literally. For example, the semicolon inside angle brackets <;> becomes a semicolon, not a comment indicator.

MASM removes one set of angle brackets each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets as there are levels of nesting.

8.11.3 Literal Character Operator

Syntax

!*character*

The literal character operator forces MASM to treat *character* as a literal. For example, you can force MASM to treat special characters such as semicolon (;) or ampersand (&) literally. Therefore, !; is equivalent to <;>.

8.11.4 Expression Operator

Syntax

%*text*

The expression operator (%) causes MASM to treat *text* as an expression. MASM computes the expression's value, using numbers of the current radix, and replaces *text* with this new value. The *text* must represent a valid assembler expression.

The expression operator is typically used in macro calls where the programmer needs to pass the result of an expression to the macro instead of to the actual expression.

In the following example, the macro call

printe <SYM1 + SYM2 = >,%(SYM1 + SYM2)

passes the text literal "SYM1 + SYM2 =" to the dummy parameter "MSG" . It then passes the value 300 (the result of the expression "sym + sym2" to the dummyparameter "N").

Example

```
printe  macro   MSG,N
        %out    * MSG,N *
        endm
SYM1    equ     100
SYM2    equ     200
        printe  <SYM1 + SYM2 = >,%(SYM1 + SYM2)
```

8.11.5 Macro Comment

Syntax

;;text

The macro comment is any text in a macro definition that does not need to be copied in the macro expansion. All *text* following the double semicolon (;;) is ignored by the assembler and will appear only in the macro definition when the source listing is created.

The regular comment operator (;) can also be used in macros. However, regular comments may appear in listings when the macro is expanded. Macro comments will appear in the macro definition, but not in macro expansions. Whether or not regular comments are listed in macro expansions depends on the use of the .LALL, .XALL, and .SALL directives described in the "File control" chapter.

Chapter 9
File Control

9.1 Introduction

This chapter describes file control directives, which provide control of the source, object, and listing files read and created by MASM during an assembly.

The file control directives include the following:

INCLUDE	Include a Source File
.RADIX	Alter Default Input Radix
%OUT	Display Message on Console
NAME	Copy Name to Object File
TITLE	Set Program Listing Title
SUBTTL	Set Program Listing Subtitle
PAGE	Set Program Listing Page Size
.LIST	List Statements in Program Listing
.XLIST	Suppress Listing Statements
.LFCOND	List False Conditional in Program Listing
.SFCOND	Suppress False Conditional Listing
.TFCOND	Toggle False Conditional Listing
.LALL	List Macro Expansions in Program Listing
.SALL	Suppress Listing Macro Expansion
.XALL	Exclude Comments from Macro Listing
.CREF	List Symbols in Cross Reference File
.XCREF	Suppress Symbol Listing

9.2 INCLUDE Directive

Syntax
INCLUDE *filename*

The INCLUDE directive inserts source code from the source file given by *filename* into the current source file during assembly. The *filename* must name an existing file. A pathname must be given if the file is not in the current working directory. If the named file is not found, MASM displays an error message and stops.

When MASM encounters an INCLUDE directive, it opens the specified file and begins to assemble its statements. When all statements have been read, MASM continues assembly with the next statement following the directive.

Nested INCLUDE directives are allowed. MASM marks included statements with the letter C in listings. This means a file named by an INCLUDE directive can contain its own INCLUDE directives.

Examples

```
        INCLUDE entry                    ; file name
        INCLUDE include/record           ; path name
        INCLUDE /usr/include/as/stdio    ; path name
```

9.3 .RADIX Directive

Syntax
.RADIX *expression*

The .RADIX directive sets the input radix for numbers in the source file. The *expression* defines whether the numbers are binary, octal, decimal, hexadecimal, or numbers of some other base in the range 2 to 16. The following table lists some common values.

Number	
2	**Binary**
8	Octal
10	Decimal
16	Hexadecimal

The *expression* is always considered a decimal number regardless of the current default radix. The default input radix is decimal.

In the following examples, the first .RADIX directive sets the input radix to hexadecimal, while the second sets the input radix to binary. The .RADIX directive does not affect the DD, DQ, or DT directives. Numbers entered in the expression of these directives are always evaluated as decimal unless a numeric suffix is appended to the value.

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

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

Examples

```
.RADIX 16
.RADIX 2
```

9.4 %OUT Directive

Syntax

%OUT *text*

The %OUT directive instructs MASM to display the *text* at the user's terminal when it reaches the line containing the specified *text* during assembly. The directive is useful for displaying messages during specific points of a long assembly.

The %OUT directive generates output for both assembly passes. The IF1 and IF2 directives can be used to control when the directive is processed.

In the following example, the %OUT directive displays the message "First Pass -- Okay" at the end of the assembler's first pass, but ignores the %OUT directive and message after the second pass.

Example

```
IF1
        %OUT First Pass -- Okay
ENDIF
```

9.5 NAME Directive

Syntax

NAME *module-name*

The NAME directive sets the name of the current module to *module-name*. A module name is used by the linker when displaying error messages.

The *module-name* can be any combination of letters and digits. Although the name can be any length, only the first six characters are used. The name must be unique and not be a reserved word.

If the NAME directive is not used, MASM creates a default module name using the first six characters of a TITLE directive. If no TITLE directive is found, the default name "A" is used.

The following examples sets the module name to "MAIN" .

Example

```
        NAME       MAIN
```

9.6 TITLE Directive

Syntax

TITLE *text*

The TITLE directive defines the program listing title. It directs MASM to copy *text* to the first line of each new page in the program listing. The *text* can be any combination of characters up to 60 characters in length.

No more than one TITLE directive per module is allowed.

Note

The first six non-blank characters of the title will be used as the module name if the module does not contain a NAME directive.

Example

```
TITLE PROG1 -- 1st Program
```

9.7 SUBTTL Directive

Syntax

SUBTTL *text*

The SUBTTL directive specifies the listing subtitle. It directs MASM to copy *text* to the line immediately after the title on each new page in the program listing. The *text* can be any combination of characters. Only the first 60 characters are used. If no *text* is given, the subtitle line is left blank.

Any number of SUBTTL directives can be given in a program. Each new directive replaces the current subtitle with the new *text*.

In the following examples, the first SUBTTL directive creates the subtitle "SPECIAL I/O ROUTINE" , while the second creates a blank subtitle.

Examples

```
SUBTTL   SPECIAL I/O ROUTINE
SUBTTL
```

9.8 PAGE Directive

Syntax

PAGE *length, width*
PAGE +
PAGE

The PAGE directive can be used to designate the line length and width for the program listing, to increment the section and adjust the section number, or to generate a page break in the listing.

If a *length* and *width* are specified, the PAGE directive sets the maximum number of lines per page to *length*, and the maximum number of characters per line to *width*. The *length* must be in the range 10 to 255. The default is 50. The *width* must be in the range 60 to 132. The default is 80. If *width* is specified, but *length* is not, a comma (,) must precede *width*.

If a plus sign (+) follows PAGE, the section number is incremented and the page number is reset to 1. Program listing page numbers have the form where *section* is the section number withing the module, and *page* is the page number within the section:

section-minor

By default, section and page numbers start at 1-1.

If no argument is given, PAGE starts a new output page in the program listing. It copies a form-feed character to the file and generates a title and subtitle line.

In the following examples, the first PAGE directive creates a page break. The second PAGE directive sets the maximum page length to 58 lines, and the maximum widtb to 60 characters. The third PAGE directive sets the maximum width to 132 characters. The current page length remains unchanged. And, the last PAGE directive increments the current section number and sets the page number to 1.

Examples

```
        PAGE
        PAGE 58,60
        PAGE ,132
        PAGE +
```

9.9 .LIST and .XLIST Directives

Syntax

```
.LIST
.XLIST
```

The .LIST and .XLIST directives control which source program lines are copied to the program listing. The .XLIST directive suppresses copying of subsequent source lines to the program listing. The .LIST directive restores copying. The directives are typically used in pairs to prevent a section of a given source file from being copied to the program listing.

The .XLIST directive overrides all other listing directives.

Example

```
        .XLIST
                ; Listing suspended here
        .LIST
                ; Listing resumes here
```

9.10 .SFCOND, .LFCOND, and .TFCOND Directives

Syntax

.SFCOND
.LFCOND
.TFCOND

The .SFCOND and .LFCOND directives determine whether false conditional blocks should be listed. The .SFCOND directive suppresses the listing of any subsequent conditional blocks whose IF condition is false. The .LFCOND directive restores the listing of these blocks. Like .LIST and .XLIST, false conditional listing directives can be used to suppress listing of the conditional blocks in sections of a program.

The .TFCOND directive sets the default mode for listing of conditional blocks. This directive works in conjunction with the **-X** option of the assembler. If **-X** option is not given in the MASM command line, .TFCOND causes false conditional blocks to be suppressed by default. If **-X** option is given, .TFCOND causes false conditional blocks to be suppressed. Everytime a new .TFCOND is inserted in the source code, listing of false-conditionals is turned off if it was on, or on if it was off.

Examples

```
.SFCOND
IF 0
        ; This block will not be listed.
ENDIF
.LFCOND
IF 0
        ; This block will be listed.
ENDIF
```

9.11 .LALL, .XALL, and .SALL Directives

Syntax

.LALL
.XALL
.SALL

The .LALL, .XALL, and .SALL directives control the listing of the statements in macros that have been expanded in the source file. MASM lists the full macro definition, but lists macro expansions only if the appropriate directive is set.

The .LALL directive causes MASM to list all the source statements in a macro, including comments preceded by a single semicolon (;) but not those preceded by a double semicolon (;;). The .XALL directive lists only those source statements that generate code or data. Comments are ignored.

The .SALL directive suppresses listing of all macro expansions. That is, MASM copies the macro call to the source listing, but does not copy the source lines generated by the call.

The .XALL directive is in effect when MASM first begins execution.

Examples

```
        .SALL
                ; No macros listed here.
        .LALL
                ; Macros listed in full.
        .XALL
                ; Macros listed by generated code or data only.
```

9.12 .CREF and .XCREF Directives

Syntax

.CREF
.XCREF *name*,,,

The .CREF and .XCREF directives control the generation of cross references for the macro assembler's cross-reference file. The .XCREF directive suppresses the generation of label, variable, and symbol cross references. The .CREF function restores this generation.

If a *name* is given with .XCREF, only that label, variable, or symbol will be suppressed. All other names will be cross referenced. The named label, variable, or symbol will also be omitted from the symbol table of the program listing. If two or more names are be given, they must be separated with commas (,).

Example

```
.XCREF               ; Suppress cross-referencing
.                    ;  of symbols in this block
.
.
.CREF                       ; Restore cross-referencing
                     ;  of symbols in this block
.XCREF  test1, test2        ; Don't cross reference test1 or test2
.                    ;  in this block
.
.
```

Appendix A
Instruction Summary

A.1 Introduction

The Macro Assembler, MASM, is an assembler for the Intel 8086, 80186, 80286, and 80386 family of microprocessors. It is capable of assembling instructions for the 8086, 80186, 80286, and 80386 microprocessors and the 8087 and 80287 floating point coprocessors. MASM will assemble any program written for an 8086, 80186, 80286, or 80386 microprocessor environment as long as the program uses the instruction syntax described in this chapter.

By default, MASM recognizes 8086 and 8087 instructions only. If a source program contains 80186, 80286, or 80287 instructions, one or more instruction set directives must be used in the source file to enable assembly of the instructions. The following sections list the syntax of all instructions recognized by MASM and the instruction set directives.

The following table explains the abbreviations used in the 8086, 8087, 80186, 80286, 80287, and 80386 syntax descriptions:

Syntax Desriptions

Symbol	Meaning
accum	accumulator: AX, or AL
reg	byte or word register byte: AL, AH, BL, BH, CL, CH, DL, DH word: AX, BX, CX, DX, SI, DI, BP, SP dword: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
segreg	segment register: CS, DS, SS, ES, FS, GS
r/m	general operand: register, memory address, indexed operand, based operand, or based indexed operand
immed	8- or 16-bit immediate value: constant or symbol
mem	memory operand: label, variable, or symbol
label	instruction label

A.2 8086 Instruction Mnemonics

The 8086 instructions are listed below. MASM assembles all 8086 instructions by default.

Syntax	Action
AAA	ASCII adjust for addition
AAD	ASCII adjust for division
AAM	ASCII adjust for multiplication

AAS	ASCII adjust for subtraction
ADC *accum, immed*	Add immediate with carry to accumulator
ADC *r/m, immed*	Add immediate with carry to operand
ADC *r/m, reg*	Add register with carry to operand
ADC *reg, r/m*	Add operand with carry to register
ADD *accum, immed*	Add immediate to accumulator
ADD *r/m, immed*	Add immediate to operand
ADD *r/m, reg*	Add register to operand
ADD *reg, r/m*	Add operand to register
AND *accum, immed*	Bitwise AND immediate with accumulator
AND *r/m, immed*	Bitwise AND immediate with operand
AND *r/m, reg*	Bitwise AND register with operand
AND *reg, r/m*	Bitwise AND operand with register
CALL *label*	Call instruction at label
CALL *r/m*	Call instruction indirect
CBW	Convert byte to word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CMC	Complement carry flag
CMP *accum, immed*	Compare immediate with accumulator
CMP *r/m, immed*	Compare immediate with operand
CMP *r/m, reg*	Compare register with operand
CMP *reg, r/m*	Compare operand with register
CMPS *src, dest*	Compare strings
CMPSB	Compare strings byte for byte
CMPSW	Compare strings word for word
CWD	Convert word to double word
DAA	Decimal adjust for addition
DAS	Decimal adjust for subtraction
DEC *r/m*	Decrement operand
DEC *reg*	Decrement 16-bit register
DIV *r/m*	Divide accumulator by operand
ESC *immed, r/m*	Escape with 6-bit immediate and operand
HLT	Halt processor
IDIV *r/m*	Integer divide accumulator by operand
IMUL *r/m*	Integer multiply accumulator by operand
IN *accum, immed*	Input from port (8-bit immediate)
IN *accum,* **DX**	Input from port given by DX
INC *r/m*	Increment operand
INC *reg*	Increment 16-bit register
INT 3	Software interrupt 3 (encoded as one byte)
INT *immed*	Software interrupt 0 through 255
INTO	Interrupt on overflow
IRET	Return from interrupt
JA *label*	Jump on above
JAE *label*	Jump on above or equal
JB *label*	Jump on below
JBE *label*	Jump on below or equal
JC *label*	Jump on carry

JCXZ *label*	Jump on CX zero
JE *label*	Jump on equal
JG *label*	Jump on greater
JGE *label*	Jump on greater or equal
JL *label*	Jump on less than
JLE *label*	Jump on less than or equal
JMP *label*	Jump to instruction at label
JMP *r/m*	Jump to instruction indirect
JNA *label*	Jump on not above
JNAE *label*	Jump on not above or equal
JNB *label*	Jump on not below
JNBE *label*	Jump on not below or equal
JNC *label*	Jump on no carry
JNE *label*	Jump on not equal
JNG *label*	Jump on not greater
JNGE *label*	Jump on not greater or equal
JNL *label*	Jump on not less than
JNLE *label*	Jump on not less than or equal
JNO *label*	Jump on not overflow
JNP *label*	Jump on not parity
JNS *label*	Jump on not sign
JNZ *label*	Jump on not zero
JO *label*	Jump on overflow
JP *label*	Jump on parity
JPE *label*	Jump on parity even
JPO *label*	Jump on parity odd
JS *label*	Jump on sign
JZ *label*	Jump on zero
LAHF	Load AH with flags
LDS *r/m*	Load operand into DS
LEA *r/m*	Load effective address of operand
LES *r/m*	Load operand into ES
LOCK	Lock bus
LODS *src*	Load string
LODSB	Load byte from string into AL
LODSW	Load word from string into AX
LOOP *label*	Loop
LOOPE *label*	Loop while equal
LOOPNE *label*	Loop while not equal
LOOPNZ *label*	Loop while not zero
LOOPZ *label*	Loop while zero
MOV *accum, mem*	Move memory to accumulator
MOV *mem, accum*	Move accumulator to memory
MOV *r/m, immed*	Move immediate to operand
MOV *r/m, reg*	Move register to operand
MOV *r/m, segreg*	Move segment register to operand
MOV *reg, immed*	Move immediate to register
MOV *reg, r/m*	Move operand to register
MOV *segreg, r/m*	Move operand to segment register
MOVS *dest, src*	Move string

MOVSB	Move string byte by byte
MOVSW	Move string word by word
MUL *r/m*	Multiply accumulator by operand
NEG *r/m*	Negate operand
NOP	No operation
NOT *r/m*	Invert operand bits
OR *accum, immed*	Bitwise OR immediate with accumulator
OR *r/m, immed*	Bitwise OR immediate with operand
OR *r/m, reg*	Bitwise OR register with operand
OR *reg, r/m*	Bitwise OR operand with register
OUT DX, *accum*	Output to port given by DX
OUT *immed, accum*	Output to port (8-bit immediate)
POP *r/m*	Pop 16-bit operand
POP *reg*	Pop 16-bit register from stack
POP *segreg*	Pop segment register
POPF	Pop flags
PUSH *r/m*	Push 16-bit operand
PUSH *reg*	Push 16-bit register onto stack
PUSH *segreg*	Push segment register
PUSHF	Push flags
RCL *r/m*, 1	Rotate left through carry by 1 bit
RCL *r/m*, CL	Rotate left through carry by CL
RCR *r/m*, 1	Rotate right through carry by 1 bit
RCR *r/m*, CL	Rotate right through carry by CL
REPE	Repeat if equal
REPNE	Repeat if not equal
REPNZ	Repeat if not zero
REPZ	Repeat if zero
RET [*immed*]	Return after popping bytes from stack
ROL *r/m*, 1	Rotate left by 1 bit
ROL *r/m*, CL	Rotate left by CL
ROR *r/m*, 1	Rotate right by 1 bit
ROR *r/m*, CL	Rotate right by CL
SAHF	Store AH into flags
SAL *r/m*, 1	Shift arithmetic left by 1 bit
SAL *r/m*, CL	Shift arithemetic left by CL
SAR *r/m*, 1	Shift arithmetic right by 1 bit
SAR *r/m*, CL	Shift arithmetic right by CL
SBB *accum, immed*	Subtract immediate and carry flag
SBB *r/m, immed*	Subtract immediate and carry flag
SBB *r/m, reg*	Subtract register and carry flag
SBB *reg, r/m*	Subtract operand and carry flag
SCAS *dest*	Scan string
SCASB	Scan string for byte in AL
SCASW	Scan string for word in AX
SHL *r/m*, 1	Shift left by 1 bit
SHL *r/m*, CL	Shift left by CL
SHR *r/m*, 1	Shift right by 1 bit
SHR *r/m*, CL	Shift right by CL
STC	Set carry flag

STD	Set direction flag
STI	Set interrupt flag
STOS *dest*	Store string
STOSB	Store byte in AL at string
STOSW	Store word in AX at string
SUB *accum, immed*	Subtract immediate from accumulator
SUB *r/m, immed*	Subtract immediate from operand
SUB *r/m, reg*	Subtract register from operand
SUB *reg, r/m*	Subtract operand from register
TEST *accum, immed*	Compare immediate bits with accumulator
TEST *r/m, immed*	Compare immediate bits with operand
TEST *r/m, reg*	Compare register bits with operand
TEST *reg, r/m*	Compare operand bits with register
WAIT	Wait
XCHG *accum, reg*	Exchange accumulator with register
XCHG *r/m, reg*	Exchange operand with register
XCHG *reg, accum*	Exchange register with accumulator
XCHG *reg, r/m*	Exchange register with operand
XLAT *mem*	Translate
XOR *accum, immed*	Bitwise XOR immediate with accumulator
XOR *r/m, immed*	Bitwise XOR immediate with operand
XOR *r/m, reg*	Bitwise XOR register with operand
XOR *reg, r/m*	Bitwise XOR operand with register

The string instructions (CMPS, LODS, MOVS, SCAS, and STOS) use the DS, SI, ES, and DI registers to compute operand locations. Source operands are assumed to be at DS:[SI]; destination operands at ES:[DI]. The operand type (BYTE or WORD) is defined by the instruction mnemonic. For example, CMPSB specifies BYTE operands and CMPSW specifies WORD operands. For the CMPS, LODS, MOVS, SCAS, and STOS instructions, the *src* and *dest* operands are dummy operands that define the operand type only. The offsets associated with these operands are not used. The *src* operand can also be used to specify a segment override. The ES register for the destination operand cannot be overridden.

Examples

```
CMPS  WORD ptr string, WORD ptr ES:0
LODS  BYTE ptr string
mov   BYTE ptr ES:0, BYTE ptr string
```

The REP, REPE, REPNE, REPNZ, or REPZ instructions provide a way to repeatedly execute a string instruction for a given count or while a given condition is true. If a repeat instruction immediately precedes a string instruction (both instructions must be on the same line), the instructions are repeated until the specified repeat condition is false, or the CX register is equal to zero. The repeat instruction decrements CX by one for each execution.

Example

```
        mov    CX, 10
        REP    SCASB
```

A.3 8087 Instruction Mnemonics

The 8087 instructions are listed below. MASM assembles all 8087 instructions by default.

Syntax	Action
F2XM1	Calculate 2^x-1
FABS	Take absolute value of top of stack
FADD	Add real
FADD *mem*	Add real from memory
FADD ST, ST(*i*)	Add real from stack
FADD ST(*i*), ST	Add real to stack
FADDP ST(*i*), ST	Add real and pop stack
FBLD *mem*	Load 10-byte packed decimal on stack
FBSTP *mem*	Store 10-byte packed decimal and pop
FCHS	Change sign on the top stack element
FCLEX	Clear exceptions after WAIT
FCOM	Compare real
FCOM ST	Compare real with top of stack
FCOM ST(*i*)	Compare real with stack
FCOMP	Compare real and pop stack
FCOMP ST	Compare real with top of stack and pop
FCOMP ST(*i*	Compare real with stack and pop stack
FCOMPP	Compare real and pop stack twice
FDECSTP	Decrement stack pointer
FDISI	Disable interrupts after WAIT
FDIV	Divide real
FDIV *mem*	Divide real from memory
FDIV ST, ST(*i*)	Divide real from stack
FDIV ST(*i*), ST	Divide real in stack
FDIVP ST(*i*), ST	Divide real and pop stack
FDIVR	Reversed real divide
FDIVR *mem*	Reverse real divide from memory
FDIVR ST, ST(*i*)	Reverse real divide from stack
FDIVR ST(*i*), ST	Reverse real divide in stack
FDIVRP ST(*i*), ST	Reversed real divide and pop stack twice
FENI	Enable interrupts after WAIT
FFREE	Free stack element
FFREE ST	Free top of stack element
FFREE ST(*i*)	Free *i*th stack element
FIADD *mem*	Add 2 or 4-byte integer
FICOM *mem*	2 or 4-byte integer compare

FICOMP *mem*	2 or 4-byte integer compare and pop stack
FIDIV *mem*	2 or 4-byte integer divide
FIDIVR *mem*	Reversed 2 or 4-byte integer divide
FILD *mem*	Load 2, 4, or 8-byte integer on stack
FIMUL *mem*	2 or 4-byte integer multiply
FINCSTP	Increment stack pointer
FINIT	Initialize processor after WAIT
FIST *mem*	Store 2 or 4-byte integer
FISTP *mem*	Store 2, 4, or 8-byte integer and pop stack
FISUB *mem*	2 or 4-byte integer subtract
FISUBR *mem*	Reversed 2 or 4-byte integer subtract
FLD *mem*	Load 4, 8, or 10-byte real on stack
FLD1	Load +1.0 onto top of stack
FLDCW *mem*	Load control word
FLDENV *mem*	Load 8087 environment (14-bytes)
FLDL2E	Load log$_2$e onto top of stack
FLDL2T	Load log$_2$10 onto top of stack
FLDLG2	Load log$_{10}$2 onto top of stack
FLDLN2	Load log$_e$2 onto top of stack
FLDPI	Load pi onto top of stack
FLDZ	Load +0.0 onto top of stack
FMUL	Multiply real
MUL *mem*	Multiply real from memory
FMUL ST, ST(*i*)	Multiply real from stack
FMUL ST(*i*), ST	Multiply real to stack
FMULP ST(*i*), ST	Multiply real and pop stack
FNCLEX	Clear exceptions with no WAIT
FNDISI	Disable interrupts with no WAIT
FNENI	Enable interrupts with no WAIT
FNINIT	Initialize processor, with no WAIT
FNOP	No operation
FNSAVE *mem*	Save 8087 state (94 bytes) with no WAIT
FNSTCW *mem*	Store control word with no WAIT
FNSTENV *mem*	Store 8087 environment with no WAIT
FNSTSW *mem*	Store 8087 status word with no WAIT
FPATAN	Partial arctangent function
FPREM	Partial remainder
FPTAN	Partial tangent function
FRNDINT	Round to integer
FRSTOR *mem*	Restore 8087 state (94 bytes)
FSAVE *mem*	Save 8087 state (94 bytes) after WAIT
FSCALE	Scale
FSQRT	Square root
FST	Store real
FST ST	Store real from top of stack
FST ST(*i*)	Store real from stack
FSTCW *mem*	Store control word with WAIT
FSTENV *mem*	Store 8087 environment after WAIT
FSTP *mem*	Store 4, 8, or 10-byte real and pop stack
FSTSW *mem*	Store 8087 status word after WAIT
FSUB	Subtract real
FSUB *mem*	Subtract real from memory
FSUB ST, ST(*i*)	Subtract real from stack
FSUB ST(*i*), ST	Subtract real to stack

FSUBP ST(*i*), ST	Subtract real and pop stack
FSUBR	Reversed real subtract
FSUBR *mem*	Reversed real subtract from memory
FSUBR ST, ST(*i*)	Reversed real subtract from stack
FSUBR ST(*i*), ST	Reversed real subtract in stack
FSUBRP ST(*i*), ST	Reversed real subtract and pop stack
FTST	Test top of stack
FWAIT	Wait for last 8087 operation to complete
FXAM	Examine top of stack element
FXCH	Exchange contents of stack elements
FFREE ST	Exchange top of stack element
FFREE ST(*i*)	Exchange top of stack and *i*th element
FXTRACT	Extract exponent and significand
FYL2X	Calculate Y $\log_2 x$
FYL2PI	Calculate Y $\log_2(x+1)$

A.4 80186 Instruction Mnemonics

The 80186 instruction set consists of all 8086 instructions plus the following instructions. The .186 directive must be placed at the beginning of the source file to enable these instructions.

Syntax	Action
BOUND *reg*, *mem*	Detect value out of range
ENTER *immed16*, *immed8*	Enter procedure
IMUL *immed*, *reg*	Integer multiply immediate byte into word register
IMUL *r/m*, *immed*	Integer multiply operand by immediate word/byte
INS *mem*, DX	Input string from port DX
INSB *mem*, DX	Input byte string from port DX
INSW *mem*, DX	Input word string from port DX
LEAVE	Leave procedure
OUTS DX, *mem*	Output byte/word/string to port DX
OUTSB DX, *mem*	Output byte string to port DX
OUTSW DX, *mem*	Output word string to port DX
POPA	Pop all registers
PUSH *immed*	Push immediate word/byte
PUSHA	Push all registers
RCL *r/m*, *immed*	Rotate left through carry immediate
RCR *r/m*, *immed*	Rotate right through carry immediate
ROL *r/m*, *immed*	Rotate left immediate
ROL *r/m*, *immed*	Rotate right immediate
SAL *r/m*, *immed*	Shift arithmetic left immediate
SAR *r/m*, *immed*	Shift arithmetic right immediate
SHL *r/m*, *immed*	Shift left immediate
SHR *r/m*, *immed*	Shift right immediate

A.5 80286 Non-Protected Instruction Mnemonics

The 80286 non-protected instruction set consists of all 8086 instructions plus the following instructions. The .286C directive must be placed at the beginning of the source file to enable these instructions.

Syntax	Action
BOUND *reg, mem*	Detect value out of range
ENTER *immed16, immed8*	Enter procedure
IMUL *immed, reg*	Integer multiply immediate byte into word register
IMUL *r/m, immed*	Integer multiply operand by immediate word/byte
INS *mem*, **DX**	Input string from port DX
INSB *mem*, **DX**	Input byte string from port DX
INSW *mem*, **DX**	Input word string from port DX
LEAVE	Leave procedure
OUTS DX, *mem*	Output byte/word/string to port DX
OUTSB DX, *mem*	Output byte string to port DX
OUTSW DX, *mem*	Output word string to port DX
POPA	Pop all registers
PUSH *immed*	Push immediate word/byte
PUSHA	Push all registers
RCL *r/m, immed*	Rotate left through carry immediate
RCR *r/m, immed*	Rotate right through carry immediate
ROL *r/m, immed*	Rotate left immediate
ROL *r/m, immed*	Rotate right immediate
SAL *r/m, immed*	Shift arithmetic left immediate
SAR *r/m, immed*	Shift arithmetic right immediate
SHL *r/m, immed*	Shift left immediate
SHR *r/m, immed*	Shift right immediate

A.6 80286 Protected Instruction Mnemonics

The 80286 protected instruction set consists of all 8086 and 80286 non-protected instructions plus the following instructions. The .286P directive must be placed at the beginning of the source file to enable these instructions.

Syntax	Action
ARPL *mem, reg*	Adjust requested privilege level
LAR *reg, mem*	Load access rights
LSL *reg, mem*	Load segment limit
SGDT *mem*	Store global descriptor table (8 bytes)
SIDT *mem*	Store interrupt descriptor table (8 bytes)
SLDT *mem*	Store local descriptor table
SMSW *mem*	Store machine status word
STR *mem*	Store task register
VERR *mem*	Verify read access
VERW *mem*	Verify write access

A.7 80287 Instruction Mnemonics

The 80287 instruction set consists of all 8087 instructions plus the following additional instructions. The .287 directive must be used to enable these instructions.

Syntax	Action
FSETPM	Set Protected Mode
FSTSW AX	Store Status Word in AX (wait)
FNSTSW AX	Store Status Word in AX (no-wait)

A.8 80386 Non-Protected Instruction Mnemonics

The 80386 non-protected instruction set consists of all 8086 and 80286 non-protected instructions plus the following instructions. The .386 directive must be placed at the beginning of the source file to enable these instructions.

Syntax	Action
BT reg, reg	Bit test
BT mem, reg	Bit test
BT reg, immed	Bit test
BT mem, immed	Bit test
BT mem	Bit test
BTC reg, reg	Bit test and complement
BTC mem, reg	Bit test and complement
BTC reg, immed	Bit test and complement
BTC mem, immed	Bit test and complement
BTC mem	Bit test and complement
BTR reg, reg	Bit test and reset
BTR mem, reg	Bit test and reset
BTR reg, immed	Bit test and reset
BTR mem, immed	Bit test and reset
BTR mem	Bit test and reset
BTS reg, reg	Bit test and set
BTS mem, reg	Bit test and set
BTS reg, immed	Bit test and set
BTS mem, immed	Bit test and set
BTS mem	Bit test and set
CDQ	Convert doubleword in EAX to quadword in EAX:EDX
CMPSD	String compare double word
CWDE	Convert word in AX doubleword in EAX
IMUL	r/m
IMUL reg, r/m	Uncharacterized multiply
IMUL reg, r/m, immed	Uncharacterized multiply
IMUL reg, immed	Uncharacterized multiply
INSD	String input double word
IRETD	
CDQ	Return from a 386 32-bit mode far interrupt
JA	Jump on above
JAE	Jump on above or equal
JB	Jump on below
JBE	Jump on below or equal
JC	Jump on carry
JE	Jump on equal
JG	Jump on greater

JGE	Jump on greater or equal
JL	Jump on less than
JNA	Jump on not above
JNA	Jump on not above or equal
JNB	Jump on not below
JNBE	Jump on not below or equal
JNC	Jump on no carry
JNE	Jump on no equal
JNG	Jump on no greater
JNGE	Jump on not greater or equal
JNL	Jump on not less than
JNLE	Jump on not less than or equal
JNO	Jump on not overflow
JNP	Jump on not parity
JNS	Jump on not sign
LFS reg, mem	Load reg and FS with far pointer
LGS reg, mem	Load reg and GS with far pointer
LODSD mem	Load string double word
LSS	Load reg and SS with far pointer
MOVSD	String move double word
MOVSX reg, r/m	Sign extend
MOVZX reg, r/m	Zero extend
OUTSD	Output string double word
POP FS/GS	Pop 386 segment register
POPFD	Pop double word flags
POPAD	Pop all double word registers
PUSH FS/GS	Push 386 segment register
PUSHAD	Push all double word registers
PUSHFD	Push double word flags
SCASD	Scan string double word
SETA r/m	Set byte if above
SETAE r/m	Set byte if above or equal
SETB r/m	Set byte if below
SETBE r/m	Set byte if below or equal
SETC r/m	Set byte if carry
SETE r/m	Set byte if equal
SETG r/m	Set byte if greater
SETGE r/m	Set byte if greater or equal
SETL r/m	Set byte if less
SETLE r/m	Set byte if less or equal
SETNA r/m	Set byte if not above
SETNAE r/m	Set byte if not above or equal
SETNB r/m	Set byte if not below
SETNBE r/m	Set byte if not below or equal
SETNC r/m	Set byte if not carry
SETNE r/m	Set byte if not equal
SETNG r/m	Set byte if greater

SETNGE r/m	Set byte if not greater or equal
SETNL r/m	Set byte if not less
SETNLE r/m	Set byte if not less or equal
SETNO r/m	Set byte if not overflow
SETNP r/m	Set byte if not parity
SETNS r/m	Set byte if not sign
SETNZ r/m	Set byte if not zero
SETO r/m	Set byte if overflow
SETP r/m	Set byte if parity
SETPE r/m	Set byte if parity even
SETPO r/m	Set byte if parity odd
SETS r/m	Set byte if sign
SETZ r/m	Set byte if zero
SHLD reg/mem,reg,imm/cl	Shift double precision left
SHRD reg/mem,reg,imm/cl	Shift double precision right
STOSD mem	Store string double word

A.9 80386 Protected Instruction Mnemonics

The 386 protected instruction set consists of all 8086 instructions and 80286 protected instructions plus the following instructions. The .386P directive must be placed at the beginning of the source file to enable these instructions.

Syntax	Action
CLTS	Clear task switched flag
HLT	Halt processor
LGDT *mem*	Load global descriptor table (8 bytes)
LIDT *mem*	Load interrupt descriptor table (8 bytes)
LLDT *mem*	Load local descriptor table
LMSW *mem*	Load machine status word
LTR *mem*	Load task register
MOV reg,creg	Move to or from creg
MOV dreg,dreg	Move to or from dreg
MOV treg,treg	Move to or from treg
MOV creg,reg	Move to or from creg
MOV dreg,reg	Move to or from dreg
MOV treg,reg	Move to or from treg

Appendix B
Directive Summary

B.1 Directive Names

Directives give the assembler directions and information about input and output, memory organization, conditional assembly, listing and cross-reference control, and definitions. The following table illustrates each directive.

Directives

.186	ASSUME	ENDP	IFDIF	PAGE
.286	COMMENT	ENDS	IFIDN	PROC
.286C	.CREF	EQU	IFNB	PUBLIC
.286P	DB	EVEN	IFNDEF	.RADIX
.287	DD	EXTRN	INCLUDE	RECORD
.386	DF	GROUP	LABEL	.SALL
.386C	DQ	IF	.LALL	SEGMENT
.386P	DT	IF1	.LFCOND	.SFCOND
.8086	DW	IF2	.LIST	STRUC
.8087	ELSE	IFB	NAME	SUBTTL
.PRIV	END	IFE	ORG	.TFCOND
=	ENDIF	IFDEF	%OUT	TITLE

Any combination of upper and lowercase letters can be used when giving directive names in a source file.

B.2 Directive Syntax and Function

The following is a complete list of directive syntax and function.

.186	Enables assembly of 186 instruction set.
.286	Enables assembly of 286 non-protected instruction set.
.286C	Enables assembly of 286 non-protected instruction set.
.286P	Enables assembly of 286 protected instruction set and is equivalent to the following sequence: .286 .PRIV

.287	Enables assembly of 287 instruction set.
.386	Enables assembly of 386 non-protected instruction set and sets the default word size to 4 bytes.
.386C	Enables assembly of 386 non-protected instruction set and sets the default word size to 4 bytes.
.386P	Enables assembly of 386 protected instruction set and is equivalent to the following sequence: .286 .PRIV
.8086	Enables assembly of 8086 instruction set.
.8087	Enables assembly of 8087 instruction set.
.PRIV	Enables the protected mode instruction set. Use with either the .286 or .386 directives.
name = *expression*	Assigns the numeric value of *expression* to *name*.
ALIGN *size*	Aligns the segment word size to *size* bytes. The *size* argument must be a power of 2.
ASSUME *segment-register* : *segment-name* ,,,	Selects the given segment register *segment-register* to be the default segment register for all symbols in the named segment or group. If *segment-name* is NOTHING, no register is selected.
COMMENT *delimiter text delimiter*	Treats all *text* between the given pair of delimiters *delimiter* as a comment.
.CREF	Restores listing of symbols in the cross-reference listing file.
[*name*] DB *initial-value* ,,,	Allocates and initializes a byte (8 bits) of storage for each *initial-value*.

[*name*] DW *initial-value* ,,,	Allocates and initializes a word (2 bytes) of storage for each given *initial-value*.
[*name*] DD *initial-value* ,,,	Allocates and initializes a doubleword (4 bytes) of storage for each given *initial-value*.
[*name*] DF *initial-value* ,,,	Allocates and initializes 6 bytes of storage for each given *initial-value*.
[*name*] DQ *initial-value* ,,,	Allocates and initializes a quadword (8 bytes) of storage for each given *initial-value*.
[*name*] DT *initial-value* ,,,	Allocates and initializes 10 bytes of storage for each given *initial-value*.
ELSE	Marks the beginning of an alternate block within a conditional block.
END [*expression*]	Marks the end of the module and optionally sets the program entry point to *expression*.
ENDIF	Terminates a conditional block.
name EQU *expression*	Assigns the *expression* to the given *name*.
name ENDP	Marks the end of a procedure definition.
name ENDS	Marks the end of a segment or structure type definition.
EVEN	If necessary, increments the location counter to an even value and generates one NOP instruction (90h).
EXTRN *name* : *type* ,,,	Defines an external variable, label, or symbol named *name* and whose type is *type*.

name GROUP *segment-name* ,,,	Associates a group name *name* with one or more segments.
IF *expression*	Grants assembly if the *expression* is non-zero (true).
IF1	Grants assembly on Pass 1 only.
IF2	Grants assembly on Pass 2 only.
IFB < *argument* >	Grants assembly if the *argument* is blank.
IFDEF *name*	Grants assembly if *name* is a previously defined label, variable, or symbol.
IFDIF < *argument1* >, < *argument2* >	Grants assembly if the arguments are different.
IFE *expression*	Grants assembly if the *expression* is 0 (false).
IFIDN < *argument1* >, < *argument2* >	Grants assembly if the arguments are identical.
IFNB < *argument* >	Grants assembly if the *argument* is not blank.
IFNDEF *name*	Grants assembly if *name* has not yet been defined.
INCLUDE *filename*	Inserts source code from the source file given by *filename* into the current source file during assembly.
name LABEL *type*	Creates a new variable or label by assigning the current location counter value and the given *type* to *name*.
.LALL	Lists all statements in a macro.
.LFCOND	Restores the listing of conditional blocks.

.LIST	Restores listing of statements in the program listing.
NAME *module-name*	Sets the name of the current module to *module-name.*
ORG *expression*	Sets the location counter to *expression.*
%OUT *text*	Displays *text* at the user's terminal.
name PROC *type*	Marks the beginning of a procedure definition.
PUBLIC *name* ,,,	Makes the variable, label, or absolute symbol given by *name* available to all other modules in the program.
.RADIX *expression*	Sets the input radix for numbers in the source file to *expression.*
recordname RECORD *fieldname* : *width* [= *exp*] ,,,	Defines a record type for a 8- or 16-bit record that contains one or more fields.
.SALL	Suppresses listing of all macro expansions.
name SEGMENT *align combine class*	Marks the beginning of a program segment named *name* and having segment attributes *align, combine*, and *class.*
.SFCOND	Suppresses listing of any subsequent conditional blocks whose IF condition is false.
name STRUC	Marks the beginning of a type definition for a structure.
PAGE *length* , *width*	Sets the line length and character width of the program listing.

PAGE +	Increments section page numbering.
PAGE	Generates a page break in the listing.
SUBTTL *text*	Defines the listing subtitle.
.TFCOND	Sets the default mode for listing of conditional blocks.
TITLE *text*	Defines the program listing title.
.XALL	Lists only those macro statements that generate code or data.
.XCREF *name* ,,,	Suppresses the listing of symbols in the cross-reference listing file.
.XLIST	Suppresses listing of subsequent source lines to the program listing.

Appendix C
Segment Names For High-Level Languages

C.1 Introduction

This appendix describes the naming conventions used to form assembly language source files that are compatible with object modules produced by recent Microsoft language compilers. Compilers that use these conventions include the following:

Microsoft C Version 3.0 or later

Microsoft Pascal Version 3.3 or later

Microsoft Fortran Version 3.3 or later

High-level language modules have the following four predefined segment types:

_TEXT	For program code
_DATA	For program data
_BSS	For uninitialized space (blank static storage)
_CONST	For constant data

Any assembly language source file to be assembled and linked to a high-level language module must use these segments.

High-level language modules must be one of three different memory model types when integrated with 8086 or 286 code:

Small	For single code and data segments
Middle	For multiple code segments with a single data segment
Large	For multiple code and data segments

High-level language modules must be one of two different memory model types when integrated with 386 code:

Pure-Text Small	For text and data in separate segments
Mixed	For code located in one segment and procedures or data located in another segment

Assembly language source files to be assembled for a given memory model must use the naming conventions given in this appendix.

C.2 Text Segments

Syntax

*name*_TEXT SEGMENT BYTE PUBLIC 'CODE'
        statements
*name*_TEXT ENDS

A text segment defines a module's program code. It contains *statements* that define instructions and data within the segment. A text segment must have the name *name*_TEXT, where *name* can be any valid name.

A segment can contain any combination of instructions and data statements. These statements must appear in an order that creates a valid program. All instructions and data addresses in a text segment are relative to the CS segment register. Therefore, the the following statement must appear at the beginning of the segment:

assume cs: *name*_TEXT

This statement ensures that each label and variable declared in the segment will be associated with the CS segment register.

Text segments must have "BYTE" alignment and "PUBLIC" combination type, and must have the class name "CODE" . These define loading instructions that are passed to the linker. Although other segment attributes are available, they should not be used. For a complete description of the attributes, see the "Program Structure." chapter.

For small model programs, only one text segment is allowed. The segment must not exceed 64K bytes in 8086 or 286 code, or 4G bytes in 386 code. All procedure and statement labels must have the NEAR type.

Example

```
_TEXT   segment BYTE PUBLIC 'CODE'
        assume cs:_TEXT
_main   proc near
        .
        .
        .
_main   endp
_TEXT   ends
```

C.3 Data Segments – Near

Syntax

```
DGROUP    group_DATA
          assume
_DATA     SEGMENT WORD PUBLIC 'DATA'
  statements
_DATA
```

A near data segment defines initialized data that is in the segment pointed to by the DS segment register when the program starts execution. The segment is "near" because all data in the segment is accessible without giving an explicit segment value. All programs have exactly one near data segment.

A near data segment's name must be "_DATA" . The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64K bytes in 8086 or 286 code and 4G bytes in 386 code. All data addresses in the segment are relative to the predefined group "DGROUP" . Therefore, the following statements must appear at the beginning of the segment:

```
DGROUP    group _DATA
          assume ds: DGROUP
```

These statements ensure that each variable declared in the data segment will be associated with the DS segment register and DGROUP.

Near data segments must be "WORD" aligned in 8086 or 286 code and "DWORD" aligned in 386 code, must have "PUBLIC" combination type, and must have the class name "DATA" . These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used.

Example

```
DGROUP   group     _DATA
         assume ds:DGROUP

_DATA    segment   word public 'DATA'
count    dw        0
array    dw        10 dup(1)
string   db        "Type CANCEL then press RETURN", 0ah, 0
_DATA    ends
```

C.4 Data Segments – Far

Syntax

*name*_DATA SEGMENT WORD PUBLIC 'FAR_DATA'
statements
*name*_DATA ENDS

A far data segment defines data or data space that can be accessed only by specifying an explicit segment value.

A far data segment's name must be *name*_DATA, where *name* can be any valid name. The name of the first variable declared in the segment is recommended. The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64K bytes in 8086 or 286 code and 4G bytes in 386 code. All data addresses in the segment are relative to the ES segment register. When accessing a variable in a far data segment, the ES register must be set to the appropriate segment value. Also, the segment override operator must be used with the variable's name (see the "Operands and Expressions" chapter).

Far data segments must be "WORD" aligned, must have "PUBLIC" combination type, and must have the class name "FAR_DATA" . These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used.

Example

```
array_DATA    segment   word public 'far_DATA'
array         dw        0
              dw        1
              dw        2
              dw        4
table         dw        1600 dup(?)
array_DATA    ends
```

C.5 Bss Segments

Syntax

```
DGROUP     group_BSS
           assume ds:DGROUP
_BSS       SEGMENT WORD PUBLIC 'BSS'
   statements
_BSS       ENDS
```

A bss segment defines uninitialized data space. A bss segment's name must be "_BSS" . The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64K bytes in 8086 or 286 code and 4G in 386 code. All data addresses in the segment are relative to the predefined group "DGROUP" . Therefore, the following statements must appear at the beginning of the segment:

```
DGROUP     group _BSS
        assume ds:DGROUP
```

These statements ensure that each variable declared in the bss segment will be associated with the DS segment register and DGROUP. The group name DGROUP must not be defined in more than one GROUP directive in a source file. If a source file contains both a DATA and BSS segment, the directive should be used:

```
DGROUP  group  _DATA, _BSS
```

A bss segment must be "WORD" aligned, must have "PUBLIC" combination type, and must have the class name "BSS" . These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used.

Example

```
DGROUP    group     _BSS
assume    ds:DGROUP

_BSS      segment   word public 'BSS'
count     dw        ?
array     dw        10 dup(?)
string    db        30 dup(?)
_BSS      ends
```

C.6 Constant Segments

Syntax

```
DGROUP    group   CONST
          assume ds:DGROUP
CONST     SEGMENT WORD PUBLIC 'CONST'
    statements
CONST  ENDS
```

A constant segment defines constant data that will not change during program execution.

The constant segment's name must be "CONST" . The segment can contain any combination of data *statements* defining constants to be used by the program. The segment must not exceed 64K bytes in 8086 or 286 code and 4G bytes in 386 code. All data addresses in the segment are relative to the predefined group "DGROUP" . Therefore, the following statements must appear at the beginning of the segment:

```
DGROUP    group CONST
          assume ds:DGROUP
```

These statements ensure that each variable declared in the constant segment will be associated with the DS segment register and DGROUP. The group name DGROUP must not be defined in more than one GROUP directive in a source file. If a source file eontains a DATA, BSS, and CONST segment, the directive should be used:

```
DGROUP    group    _DATA, _BSS, CONST
```

A constant segment should be "WORD" aligned, must have "PUBLIC" combination type, and must have the class name "CONST". These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used.

In the following example, the constant segment receives the segment values of two far data segments: "ARRAY_DATA" and "MESSAGE_DATA". These data segments must be defined elsewhere in the module.

Example

```
DGROUP    group      CONST
          assume ds:DGROUP

CONST     segment    word public 'CONST'
seg1      dw         ARRAY_DATA
seg2      dw         MESSAGE_DATA
CONST     ends
```

Index