

Application Development Guide

IBM Personal Computer XENIX™ Software Development System Version 2.00

Programming Family



**Personal
Computer
Software**

XENIX is a trademark of
Microsoft Corporation.

59X8634

Application Development Guide

IBM Personal Computer XENIX™ Software Development System Version 2.00

Programming Family



**Personal
Computer
Software**

XENIX is a trademark of
Microsoft Corporation.

Second Edition (April 1986)

Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication. This edition applies to Version 2.00 of the XENIX Operating System, and to all subsequent releases until otherwise indicated in new editions or technical newsletters.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Products are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer dealer.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Portions of the code and documentation described in this book were developed at the Electrical Engineering and Computer Sciences Department at the Berkeley Campus of the University of California under the auspices of the Regents of the University of California. Portions of the software and documentation are based on the Fourth Berkeley Distribution licensed from the Regents of the University of California.

- © Copyright International Business Machines Corporation 1985, 1986
- © Portions copyright by Microsoft Corporation 1983, 1984, 1986, all rights reserved
- © Portions copyright by AT&T Bell Laboratories 1983, 1984, 1986, all rights reserved

About This Book

This guide is written for the application development programmer. It explains how to use the utility programs and routines available in the PC XENIX Software Development System.

The programmer using this manual should be experienced with:

- The IBM Personal Computer AT
- The IBM Personal Computer XENIX
- C language or another high-level programming language.

For detailed formatting of command and option syntax, refer to *IBM Personal Computer XENIX Commands Reference*.

Each chapter of this guide may be studied independently. After reading Chapter 1, use the table of contents to select the topic that interests you. The book is organized as follows:

Chapter 1. Introduction

Provides an overview of the PC XENIX Software Development System.

Chapter 2. PC XENIX to DOS: A Cross Development System

Provides information on creating programs that run under DOS. You can create, compile, and link DOS programs on PC XENIX and transfer them to a DOS system.

Chapter 3. The lint Program: a C Language Program Checker

Explains how to check C language programs for correct syntax and semantics.

Chapter 4. A Program Maintainer: make

Explains how to automate the development of a program or other project.

Chapter 5. SCCS: A Source Code Control System

Explains how to control and maintain all versions of a project's source files.

Chapter 6. The adb Program Debugger

Explains how to debug C and assembler language programs using the PC XENIX debugger **adb**.

Chapter 7. The lex Program: A Lexical Analyzer

Explains how to create lexical analyzers using the program generator `lex`.

Chapter 8. The yacc Program Generator: A Compiler – Compiler

Explains how to create parsers using the program generator `yacc`.

Chapter 9. M4: A Macro Processor

Explains how to create and process macros.

Chapter 10. Writing Device Drivers

Explains how to write device drivers for PC XENIX systems.

Chapter 11. Sample Device Drivers

Shows examples of device drivers with comments.

Other PC XENIX Publications

- *IBM Personal Computer XENIX C Library Guide and Compiler Reference*
- *IBM Personal Computer XENIX Commands Reference*
- *IBM Personal Computer XENIX System Reference*
- *IBM Personal Computer XENIX Macro Assembler Reference*

Contents

Chapter 1. Introduction	1-1
Notational Conventions	1-1
Overview	1-2
Creating C Language Programs	1-3
Creating Other Programs	1-4
Creating and Maintaining Libraries	1-5
Maintaining Program Source Files	1-5
Creating Programs with Shell Commands	1-7
Chapter 2. PC XENIX to DOS: A	
Cross-Development System	2-1
Introduction	2-1
Creating DOS Source Files	2-2
Compiling a DOS Source File	2-3
Using Assembler Language Source Files	2-4
Linking DOS Object Files	2-5
Running and Debugging a DOS Program	2-5
Transferring Programs between Systems	2-6
Creating DOS Libraries	2-7
Chapter 3. The lint Program - a C Program	
Checker	3-1
Introduction	3-1
Invoking lint	3-2
Checking for Unused Variables and Functions	3-3
Checking Local Variables	3-4
Checking for Unreachable Statements	3-5
Checking Function Return Values	3-6
Checking for Unused Return Values	3-7
Checking Types	3-7
Checking Type Casts	3-9
Checking for Nonportable Characters	3-10
Checking for Assignment of longs to ints	3-11
Checking for Strange Constructions	3-12
Checking Pointer Alignment	3-14

Checking for Older C Syntax	3-14
Checking Expression Evaluation Order	3-16
Embedding Directives	3-17
Checking for Library Compatibility	3-18

Chapter 4. A Program Maintainer: make .. 4-1

Introduction	4-1
Creating a Makefile	4-2
Invoking make	4-4
Using Pseudo-Target Names	4-6
Using Macros	4-7
Using Shell Environment Variables	4-11
Using Built-In Rules	4-13
Changing Built-In Rules	4-15
Using Libraries	4-18
Troubleshooting	4-20
An Example: Using make	4-21

Chapter 5. SCCS: A Source Code Control

System	5-1
Introduction	5-1
Basic Information	5-2
Creating and Using s-files	5-7
Using Identification Keywords	5-18
Using s-file Flags	5-20
Modifying s-file Information	5-23
Printing from an s-file	5-28
Editing by Several Users	5-30
Protecting s-files	5-32
Repairing SCCS Files	5-35
Using other Command Options	5-36

Chapter 6. The adb Program Debugger 6-1

Introduction	6-1
Starting and Stopping adb	6-1
Displaying Instructions and Data	6-6
Debugging Program Execution	6-20
Using the adb Memory Maps	6-34
Validating Addresses	6-37
Miscellaneous Features	6-38
An Example: Directory and Inode Dumps	6-44
Patching Binary Files	6-46

Chapter 7. The lex Program: A Lexical Analyzer	7-1
Introduction	7-1
Invoking lex	7-4
The lex Source Format	7-5
The lex Regular Expressions	7-7
Using the Operator Characters	7-8
Writing Actions	7-14
Handling Ambiguous Source Rules	7-19
Specifying Left Context Sensitivity	7-23
Specifying Source Definitions	7-26
Using yacc with lex	7-28
Specifying Character Sets	7-29
Source Format	7-31
A lex Example	7-34
Chapter 8. The yacc Program Generator: A Compiler-Compiler	8-1
Introduction	8-1
Specifications	8-5
Actions	8-9
Lexical Analysis	8-12
How the Parser Works	8-14
Ambiguity and Conflicts	8-21
Precedence	8-27
Error Handling	8-31
The yacc Environment	8-34
Preparing Specifications	8-36
An Example: A Small Desk Calculator	8-44
The yacc Input Syntax	8-48
An Advanced Example	8-52
Out-Dated Features	8-62
Chapter 9. The m4 Macro Processor	9-1
Introduction	9-1
Invoking m4	9-2
Defining Macros	9-3
Quoting	9-5
Using Arguments	9-9
Using Arithmetic Built-Ins	9-11

Using System Commands	9-13
Using Conditionals	9-13
Manipulating Files	9-14
Manipulating Strings	9-16
Cleaning Up Output	9-18
Printing	9-20
Chapter 10. Writing Device Drivers	10-1
Introduction	10-1
Kernel Environment	10-6
Kernel Support Routines	10-12
Parameter Passing to Device Drivers	10-32
Naming Conventions	10-33
Device Drivers for Character Devices	10-33
Device Drivers for Block Devices	10-50
Rules for Writing Installable Device Drivers	10-61
Configuring the System	10-68
Warnings	10-76
Chapter 11. Sample Device Drivers	11-1
Introduction	11-1
Sample Device Driver for Line Printer	11-2
Sample Device Driver for Terminal	11-10
Sample Device Driver for Disk Drive	11-38
Writing Drivers for Memory-Mapped Screens	11-52
Index	X-1

Ordering Additional Copies of This Book

To order additional copies of this publication (*IBM Personal Computer XENIX Application Development Guide*) use either of the following sources:

- To order from your IBM representative, use Order Number SV21-8078.
- To order from your IBM dealer, use Part Number 59x9949.

This publication includes the following:

- *IBM Personal Computer XENIX Application Development Guide*
- Binder
- Slipcase.

Chapter 1. Introduction

Notational Conventions

Throughout the IBM Personal Computer PC XENIX library, different printing styles highlight important information. As you read this book, be aware of the following conventions.

[] Brackets indicate an optional command argument.

bold **Boldface** indicates **commands, options, file names, programs, and functions**. You must enter these **boldface** characters exactly as shown.

italics *Italic* characters indicate variables for such things as *placeholders, arguments, and filenames*. When you enter a command, replace all variables with the appropriate file name, number, or option.

bold italics

Bold italics indicate the first time a glossary term appears in this book. That term is defined in *IBM Personal Computer XENIX Glossary and Master Index*.

monospace

Monospace indicates coding examples, names taken from coding examples, and the exact wording of text on the screen.

. . . Ellipses after an argument indicate you can repeat that argument one or more times.

Overview

The IBM Personal Computer PC XENIX Software Development System provides many utilities to help you design and develop application programs. These utilities help you create C and assembler language programs for execution on the PC XENIX system. They also let you automate program creation, debug these programs, and maintain different versions of the same program.

This chapter introduces you to the PC XENIX Software Development System and some of its utilities. The following chapters explain how to use these and other routines and utilities. Some commands mentioned here are part of the IBM Personal Computer PC XENIX Operating System rather than the Software Development System. The *XENIX Commands Reference* or the *XENIX System Reference* contains a complete explanation of these commands.

Creating C Language Programs

You can efficiently create C language source files with text editors. The most convenient editor for you to use is **vi**. The **vi** editor is a full-screen editor that allows you to see a full screen of text. You can use many **vi** commands to insert, replace, move, and search for text. All commands are invoked from command keys or from a screen displayed command line. You can also use a variety of options in the editor that lets you modify its operation. For more information on **vi**, see *XENIX vi and ed Editors*.

Once you have created your source program, you can compile it with the C language compiler. The **cc** command invokes the compiler. You can also use **cc** to invoke other utilities such as the link editor **ld** and the assembler **as**.

You can debug an executable C program with the debugger **adb**. The **adb** debugger provides a direct interface to the machine instructions that make up an executable program.

To check a program before compiling it, you can use **lint**, the C program checker. The **lint** utility program checks for syntactical and logical errors. It also enforces a strict set of guidelines for proper C language programming style. The **lint** utility is normally used in the early stages of program development.

You can improve a program's format with **cb**, the C program beautifier. The beautifier improves the appearance of C language programs and, therefore, makes them easier to read. A program that is easy to read improves your ability to find logical errors.

Creating Other Programs

The C language can meet the needs of most programming projects. In cases where you require greater control, you can create assembler language programs using the **as** program. This program assembles source files and produces object files. You can then relocate or link the object files to C language programs.

You can use the **ld** command to invoke the link editor, but you should use **ld** only with the PC XENIX compiler. The **ld** command links relocatable object files to produce executable programs.

You can create source files for lexical analyzers and parsers using the program generators **lex** and **yacc**. Lexical analyzers locate patterns of complex input and convert them into meaningful values or tokens. The **lex** utility is a lexical analyzer generator. It generates lexical analyzers, written in C program statements, from specifications you provide. Parsers convert meaningful sequences of tokens and values into actions. The parser generator, **yacc**, generates parsers, written in C program statements, from given specification files. The **lex** and **yacc** program generators are often used together to make complete programs.

You can preprocess C and assembler language source files, or **lex** and **yacc** source files, by using the **m4** macro processor. The **m4** utility performs several preprocessing functions. Two examples of these functions are:

- Converting macros to their defined values
- Calling the contents of one or more files into a source program.

Creating and Maintaining Libraries

You can create and maintain libraries of functions and programs by using the **ar** and **ranlib** utilities. The archiver, **ar**, creates libraries of relocatable object files. The random library generator, **ranlib**, converts archive libraries to random libraries. The **ranlib** command also places a table of contents at the beginning of each library.

The **lorder** command finds the ordering relationship in an object library and produces a list of dependent pairs. The **tsort** command sorts the dependent pairs into an order that shows their dependencies.

Maintaining Program Source Files

The **make** utility is a program maintainer. It automates the steps required to create executable programs and provides a mechanism for ensuring up-to-date programs. You should generally use **make** with large-scale programming projects.

The Source Code Control System (SCCS) is a collection of commands that create, maintain, and control special files called SCCS files. The SCCS commands let you maintain different versions of a single program. They do so by storing the original program and each set of changes. The commands compress all versions of a source file into a single file containing a list of differences. These commands can also restore compressed files to their original size and content.

Many PC XENIX commands let you carefully examine a program's source files. The **ctags** command creates a tags file. From the tags file, you can find C functions in a set of related C language source files. The **mkstr** command creates an error message file by examining a C language source file.

The following commands let you examine object and executable binary files:

- nm** The **nm** command prints the list of symbol names in a program.
- hd** The **hd** command performs a hexadecimal dump of given files. Options available with this command allow you to choose a variety of formats for the printed output.
- size** The **size** command reports the size of an object file.
- strings** The **strings** command finds and prints readable text (strings) in an object or other binary file.
- strip** The **strip** command removes symbols and relocation bits from executable files.
- sum** The **sum** command computes a checksum value for a file and a count of its blocks. It searches for bad spots in a file and verifies transmission of data between systems.
- xstr** The **xstr** command extracts strings from C language programs to implement shared strings.

Creating Programs with Shell Commands

In the PC XENIX system, you can write a program with a series of shell commands. Shell commands provide much of the same control capability as the C language. They also give direct access to all the commands and programs normally available to the PC XENIX user.

The **cs**h command invokes the C-shell, a command interpreter. The C-shell interprets and executes commands it receives from the keyboard or from a command file. Since it uses a syntax similar to the C language, programming with shell commands is easy. It also has an aliasing facility and a command history mechanism.

For more information concerning the C-shell, refer to *XENIX Commands Reference*.

Chapter 2. PC XENIX to DOS: A Cross-Development System

Introduction

The PC XENIX Software Development System contains a DOS cross-development system. The DOS cross-development system allows you to create, compile, and link DOS programs on the PC XENIX system. For execution and debugging, you must transfer these programs to a DOS system.

The complete DOS cross-development system consists of:

- The C program compiler **cc**
- The 8086 assembler **as**
- 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. A special **-dos** option directs the compiler to create code for execution under DOS. When you use **-dos**, **cc** uses special DOS **include** files and libraries to create the program.

The **cc** command invokes the **as** command when you use 8086 assembler language source files. The **cc** command uses **dosld** commands to carry out the last part of the compiling process. You can also invoke **as** and **dosld** directly when you need to perform special tasks.

You cannot execute or debug DOS programs on the PC XENIX system. Therefore, you must copy the programs to a DOS system before executing them. PC XENIX **dos(C)** commands allow you to transfer files from PC XENIX to DOS or from DOS to PC XENIX.

Creating DOS Source Files

You can create program source files by using either PC XENIX or DOS text editors. The most convenient way is to use one of the PC XENIX editors, such as **vi**, the full-screen editor.

When creating source files to be executed on a DOS system, you should follow these rules:

- Use the standard C language format for your source files. DOS source files have the same format as PC XENIX source files. Many DOS programs, compiled without the **-dos** option, can be executed on the PC XENIX system.
- Use the DOS naming conventions when giving file and directory names within a program. For example, use `\` instead of `/` for the path name separator. Since the compiler does not check names, failure to follow DOS 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 PC XENIX counterparts. Others have only slight differences. See *XENIX C Library Guide and Compiler Reference* for:
 - A complete list of the available DOS **include** files and functions
 - A description of the differences between the DOS and PC XENIX files and functions.

If you use a function that does not exist, **dosld** displays an error message and leaves the linked output file incomplete.

Compiling a DOS Source File

You can compile a DOS source file by using the **-dos** option of the PC XENIX **cc** command. The command line has the form:

```
cc -dos options filename . . .
```

where *options* are **cc** command options, and *filename* is the name of the source file you want to compile. You can specify more than one source file, if you desire. Each source file name must end with the **.c** extension.

The **cc** command compiles each source file separately and creates an object file for each. It then links all object files together with the appropriate C language libraries. The object files created by the **cc** command have the same base name as the source file. The **cc** command also changes the **.c** extension to a **.o** extension. If you do not explicitly name the file, **cc** gives the name **a.out** to the resulting program file.

For example, the command:

```
cc -dos test.c
```

compiles the source file `test.c` and creates the object file `test.o`. It then calls the **dosld** command, 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. For a complete listing of these options, see **cc(CP)** in *XENIX Commands Reference*.

Default values for an option may be different for a DOS system than for a PC XENIX system. In particular, the default directory for library files, given with the **-l** option, is **/usr/lib/dos**. Also, note that you cannot use the **-p** (for profiling) option.

For more information on the **cc** command, refer to *XENIX C Library Guide and Compiler Reference*.

Using Assembler Language Source Files

You can direct the **cc** command to assemble 8086 assembler language source files by including the files in the **cc** command line. Like C source files, assembler language source files may contain calls to functions. However, these functions must be in the DOS libraries. Furthermore, the source files must follow the C calling conventions described in Appendix A of *XENIX C Library Guide and Compiler Reference*. The file name of an assembler language source file must end with the **.s** extension.

When you specify an assembler language source file, **cc** automatically invokes **as**, the 8086 assembler. The assembler creates an object file that you can link with any other object file created by **cc**.

You can invoke the assembler directly by using the **as** command. This command creates an object file just as the **cc** command does, but it does not create an executable file. For a description of the command and its options, see **as(CP)** in *XENIX Commands Reference*.

Linking DOS Object Files

You can link DOS object files by including the file names in the **cc** command line. However, these object files must have been created using either **as** or the **-dos** option of **cc**. Also, the object file names must end with the **.o** extension.

When you include an object file, **cc** automatically invokes **dosld**, the DOS linker. The **dosld** command 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 invoke the linker directly by using the **dosld** command. This command creates a DOS program file just as the **cc** command does, but **dosld** does not accept source files. For a description of the command and its options, see **dosld(CP)** in *XENIX Commands Reference*.

Note: DOS programs created by **cc** and **dosld** are designed to be compatible with PC-DOS systems up to and including version 3.1. However, you cannot execute DOS programs on the PC XENIX system.

Running and Debugging a DOS Program

To debug a DOS program, you must transfer it to a DOS system. Use 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 *IBM Personal Computer Disk Operating System (DOS) Reference 3.0*.

Transferring Programs between Systems

You can transfer programs between PC XENIX and DOS systems by using DOS diskettes and the PC XENIX **dosc** command. The **dosc** command lets you copy files to a DOS diskette. The command has the form:

```
dosc -r file-1 dev: file-2
```

where **-r** is the raw option that is required for load modules, *file-1* is the name of the DOS program file you want to transfer, *dev* is the full path name of a PC XENIX system diskette drive, and *file-2* is the *filename*, including the full *pathname*, of the new program file on the DOS diskette. The new *filename* must have the .EXE extension. The **-r** option ensures that the program file is copied byte for byte.

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 PC XENIX to DOS, make sure you rename **a.out** files to an .EXE or .COM file.

Creating DOS Libraries

You can create a library of DOS object files by using the PC XENIX **ar** command. The **ar** command copies object files created by the compiler to a specified 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 want to add to the library.

Note: DOS libraries created on the PC XENIX system are not compatible with libraries created on the DOS system. This means you cannot copy the libraries to the DOS system and expect them to work with the DOS **Link** command.

Chapter 3. The lint Program - a C Program Checker

Introduction

This chapter explains how to use the C language program checker **lint**. The **lint** program examines C language source files and warns of possible compiling or execution problems.

The **lint** utility program checks for:

- Unused functions and variables
- Unknown values in local variables
- Unreachable statements
- 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. The C compiler does not perform the sophisticated type and error checking that many programs require. The **lint** program provides thorough checking of source files but cannot compile these files.

Invoking lint

You can invoke **lint** at the shell command line by typing:

```
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 you want to check, and *lib* is the name of a library to check. You can give more than one option, file name, or library name by separating them with spaces. If you give two or more file names, **lint** checks the files as if they were portions of one complete program. For example, the command:

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

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

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

filename (*num*): *description*

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

```
main.c
```

```
= = = = = = =
```

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

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

Checking for Unused Variables and Functions

The **lint** utility checks a source file for unused variables and functions. The program considers every variable or function that appears in at least one statement. The **lint** program considers a variable or function unused 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 the variable **z** is not.

It is common for a programmer to remove a variable or function but forget to remove its declaration. Unused variables and functions rarely cause working programs to fail. They do, however, make programs harder to understand and change.

The **lint** program does not report external declarations. It assumes such a variable or function is used in an additional source file.

You can direct **lint** to ignore the external declarations in a source file by using the **-x** option. Use of this option causes the checker to skip any line that begins with the **extern** storage class. The **-x** option saves time when checking a program, if 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. This may be true even if these arguments are never used. 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 those declared with **register** storage class. The program considers these arguments to be a preventable waste of register resources.

You can direct **lint** to ignore all unused external variables and functions by using the **-u** (for unused) option. This option prevents **lint** from reporting variables and functions it considers unused. Use the **-u** option to check source files that contain a portion of a large program. Such source files usually contain

declarations of variables and functions intended for use in other source files.

Checking Local Variables

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

The **lint** program searches for the first time a variable receives a value. It also searches for the first time a variable is used. If the first assignment appears later than the first use, **lint** warns of an error. For example, in the program fragment:

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

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

If you use a variable when you assign its first value, **lint** will display an error. For example, in the program fragment:

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

lint warns that the variable `total` is used before its value is set. This warning is accurate because the variable is used in the same statement that assigns its first value.

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

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` is unreachable because execution control is returned before `exit` is reached.

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 common (some programs created by the **yacc** and **lex** programs contain hundreds), so you may want to suppress these reports. You can direct **lint** to suppress the reports by using the **-b** option.

The **-b** option assumes that all functions eventually return control. This option does not report a 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.

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. The **lint** program addresses these problems in a number of ways.

Within a function definition, specifying both:

```
return (expr);
```

and

```
return;
```

may cause an error. In this case, **lint** displays the following error message:

```
function name contains 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. Consider the following example:

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

If the variable `a` is false, then `f` calls the function `g` and returns with no defined return value. This triggers a report from **lint**. If `g` never returns a value, **lint** still displays an error message even though nothing is wrong. This feature can help you discover potentially serious bugs in your programs. It also accounts for a substantial number of the undeserved error messages that **lint** produces.

Checking for Unused Return Values

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

The **lint** program also checks for cases where a function does not return a value, but the value is used anyway. The **lint** program considers this a serious error.

Checking Types

C language compilers do not strictly check your use of data types. The **lint** program is very useful when you need strict type checking. Additional checking occurs in four major areas:

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

A number of operators have an implied balancing between types of operands. The assignment, conditional, and relational operators have this property. The **lint** program handles the argument of a **return** statement and expressions used in initialization in a similar way. In these operations, you can freely mix **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types. The types of pointers must agree exactly, except that you can intermix arrays of **x**'s with pointers to **x**'s.

In structure references, the type-checking rules also require that:

- The left operand of a pointer arrow symbol (->) be a pointer to a structure
- The left operand of a period (.) be a structure
- The right operand of these operators be a member of the structure implied by the left operand.

The **lint** program checks references to unions in a similar manner.

Strict rules apply to the matching of function arguments to return values. You can freely match the types **float** and **double**. You can also freely match the types **char**, **short**, **int**, and **unsigned**. You can also match pointers with the associated arrays. All other actual arguments must agree in type with their declared counterparts.

The **lint** program checker makes sure 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 can also be function arguments and return values.

Checking Type Casts

The type cast feature of the C language was introduced in PC XENIX to help you produce portable programs. Consider the assignment:

```
p = 1 ;
```

where `p` is a character pointer. The **lint** program reports this as a possible error. If you change the assignment to:

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

using a cast to convert the integer to a character pointer, **lint** accepts the assignment. In the second example, your intentions are clear. The `-c` option controls the printing of comments about casts. When `-c` is in effect, **lint** does not check the casts. The `-c` option passes all legal casts without comment, no matter how strange the type mixing seems to be.

Checking for Nonportable Characters

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

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

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

```
nonportable character comparison
```

The solution is to declare **c** an integer, because the **getchar** function is actually returning integer values.

A similar issue arises with bit fields. For example, in the code fragment:

```
struct {  
    int      int2bit   : 2;  
    unsigned un2bit   : 2;  
};  
.  
.  
.  
fields.int2bit = 3;  
fields.un2bit  = 3;
```

lint issues the following warning on the 2-bit field of **int** type **int2bit**:

```
warning: precision lost in assignment to (sign-extended?) field
```

When you assign constant values to bit fields, the field may be too small to hold the value. This is especially true when bit fields 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.

Checking for Assignment of longs to ints

Assigning **long** values to **int** values can cause a loss of accuracy. You can mistakenly assign a **long** to an **int** by changing type definitions with **typedef**. Your program may stop working if you change a **typedef** variable from **int** to **long**:

```
typedef int long;
```

This problem may occur because some intermediate results may be assigned to an integer variable, and the intermediate result is truncated. There are a number of legitimate reasons for assigning longs to integers. You may want to suppress detection of these assignments by using the **-a** option.

Checking for Strange Constructions

The **lint** program encourages better code quality and clearer style. It warns of constructions that appear strange, even though they may be legal. For example, in the statement:

```
*p++ ;
```

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

```
null effect
```

The program fragment:

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

is also strange because the test will never succeed. The **lint** program prints the message:

```
degenerate unsigned comparison
```

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:

```
unsigned comparison with 0?
```

If you specify:

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

then **lint** reports:

```
constant in conditional context
```

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

The **lint** utility also checks variables that you declare in both inner and outer blocks. If their inner and outer use conflicts, **lint** displays an error message. Using variables in this way is legal, but is usually unnecessary. This kind of usage is poor style, and may frequently cause a bug in your program.

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

Checking Pointer Alignment

Certain pointer assignments can be legal on some machines, but illegal on others, due to alignment restrictions. For example, on some machines double-precision values can begin on any integer boundary. On other machines, however, double-precision values must begin on even-word boundaries. The `lint` program warns of possible alignment problems with the message:

```
possible pointer alignment problem
```

Checking for Older C Syntax

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

The older forms of assignment operators (for example, `= +`, `=-`, `. . .`) can make expressions unclear. In the example:

```
a =-1 ;
```

either of the following may be what you intend:

```
a =- 1 ;
```

or

```
a = -1 ;
```

This confusion is greater if it results from macro substitution. The newer, and preferred operators (for example, `+ =`, `- =`) are much clearer. To encourage the use of the newer forms, `lint` checks for these older operators.

A similar issue arises with initialization. To initialize `x` to 1, the older language allowed:

```
int x 1 ;
```

This causes syntactic difficulties. For example:

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

looks somewhat like the beginning of a function declaration:

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

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

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

This construction reduces possible confusion.

Checking Expression Evaluation Order

In complicated expressions, the best order to evaluate subexpressions may depend on the machine. For example, on machines in which the stack runs backwards, function arguments are probably best evaluated from right to left. On machines with a stack running forward, evaluation from 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. The assignment operators and the increment and decrement operators are examples.

To ensure efficiency on a particular machine, the C language leaves the order of evaluation up to the compiler. C compilers have many differences in the order in which they 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.

The **lint** program 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
```

Embedding Directives

The **lint** program is a tool that can, at times, be annoying. You may have valid reasons for illegal type casts. You may need functions with a variable number of arguments. Other constructions that **lint** finds objectionable may be necessary to your program. The flow of control information that **lint** produces often has blind spots. To reduce these annoyances, **lint** recognizes certain key words called *directives*. The directives, when embedded as comments in a C source file, control the output of the **lint** program. Directives are invisible to the compiler. The directives are listed below:

- `/* NOTREACHED */` Indicates the flow of control cannot reach this place in the program.
- `/* ARGSUSED */` Turns on the `-v` option for one function.
- `/* VARARGS */` Turns off comments about a variable number of arguments in calls to a function.

In some cases, you might want 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 number immediately after the `VARARGS` keyword. For example:

```
/* VARARGS2 */
```

causes only the first two arguments to be checked.

- `/* LINTLIBRARY */` At the head of a file, identifies this file as a library declaration file.

Checking for Library Compatibility

The **lint** program accepts certain library directives. It tests the source files for library compatibility with the command option:

-l*libraryname*

where *libraryname* is the name of the library you want **lint** to check. To perform this test, access the library description files whose names are constructed from the library directives. These files all begin with the directive:

```
/* LINTLIBRARY */
```

A series of dummy function definitions follow this directive. These definitions indicate whether a function returns a value and what type the return value is. They also indicate the number and types of arguments the function expects to be returned. Use the **VARARGS** and **ARGSUSED** directives to specify features of the library functions.

When **lint** processes a library file, functions that the file defines but does not use in a source file draw no comments. The **lint** program does not simulate a full library search algorithm. It does, however, check to see if the source files contain redefinitions of library routines.

By default, **lint** checks the programs you specify against a standard library file. This file contains descriptions of the programs that are normally loaded when a C program is run. When you use the **-p** option, **lint** checks the **portable** library file. This library contains descriptions of the standard I/O library routines that are portable across various machines.

The **-n** option suppresses all library checking.

Chapter 4. A Program Maintainer: **make**

Introduction

The **make** utility program provides you with an easy way to automate the creation of large programs. The **make** utility reads commands from a user-defined *makefile*. The *makefile* lists the files to be created, the commands that create them, and the files from which they are created. The **make** utility creates programs by executing given commands. If a file is not up-to-date, **make** updates it before creating the program. The **make** utility updates a program by executing explicitly given commands or one of the many *built-in* commands.

This chapter explains how to use the **make** program to:

- Create *makefiles* for each project
- Invoke **make** for creating programs and updating files
- Automate large programming projects.

For more details about the utility, see **make (CP)** in *XENIX Commands Reference*.

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 **make** requires to bring a file up-to-date. A dependency line has the form:

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

where *target* is the file name of the file to be updated, *dependent* is the file name of the file on which the target depends, and *command* is the PC 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 can specify more than one target file name or dependent file name. At least one space must separate each file name. A colon (;) must separate target file names from dependent file names. You must enter all file names exactly as they appear in the PC XENIX system; although you can use shell metacharacters, such as the asterisk (*) and the question mark (?).

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

You can add a comment to a `makefile` by starting the comment with a number sign (#) and ending it with a newline character. The **make** program ignores all characters after the number sign. If a command contains a number sign, you must enclose the number sign in double quotation marks ("#").

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

Keep the `makefile` in the same directory as the given source files. For convenience, PC XENIX provides the file names **makefile**, **Makefile**, **s.makefile**, and **s.Makefile** as default file names. If you do not specify a file name for your `makefile`, **make** will use a default. If a file name begins with the **s.** prefix, **make** assumes it

is an SCCS file. It then invokes the appropriate SCCS command to retrieve the latest version of the file.

To illustrate dependency lines, consider the following example. Let us assume you have three C language source files : `x.c`, `y.c`, and `z.c`. Compiling each source file gives you: `x.o`, `y.o`, and `z.o`. Linking these three files together gives you a program that you name `prog`. Now, let us assume the `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.

Note: You must precede the `cc` with a tab rather than spaces. 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 that defines how to update the given target file.

Invoking make

Once you have a `makefile`, you can invoke **make** by typing:

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

where *option* is a program option you choose to modify program operation, *macdef* is a macro definition that gives a macro a value or meaning, and *target* is the name of the file to be updated. All arguments are optional. If you specify more than one argument, you must separate them with spaces.

PC XENIX provides four default file names for your `makefile`. The default file names are: **makefile**, **Makefile**, **s.makefile**, and **s.Makefile**. You can specify the name of the `makefile` you want **make** to use by using the `-f` option. The command has the form:

```
make -f filename
```

where *filename* is the name of the `makefile` you want to use. You must supply a full path name if the file is not in the current directory.

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

You can direct **make** to update your files. For example, assume that your current `makefile` contains the dependency lines given in the last section. The **make** program compares the current date of `prog` with each date of each object file and each source file used to make-up `prog`. Because `prog` depends upon current object files which in turn depend upon current source files, **make** recreates all dependencies that need updating. If none of the source or object files need updating, **make** announces this fact and stops.

You can direct **make** to update the first target file in `makefile`, by typing the program name as your only argument.

```
make prog
```

In this case, **make** searches your current directory for **makefile**, **Makefile**, **s.makefile**, and **s.Makefile** and uses the first one it finds.

You can direct **make** to update a specific target file by specifying the file name of the target.

```
make x.o
```

In this case, **make** recompiles `x.o` if `x.c` or `defs` need updating. Remember, you can repeat the *filename* argument in the same command line. For example:

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

This causes **make** to recompile both `x.o` and `z.o` if their corresponding dependents need updating. The **make** program processes target names from the command line in a left to right order.

You can 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 PC XENIX commands.
- k** Abandons work on the current entry, but continues work 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. The **make** program 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, you must be careful with commands that have meaning only within a single shell process. Commands of this type are the **cd** command and shell control commands. The **make** program discards the results of these commands before it executes the next line of the program. If an error occurs, **make** normally stops the command.

Using Pseudo-Target Names

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

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

Because no file exists for a pseudo-target name, **make** always assumes the target is out-of-date. Thus, the associated command is always executed.

The **make** program also has built-in pseudo-target names that modify its operation. The pseudo-target name **.IGNORE** causes **make** to ignore errors during execution of commands, thus allowing **make** to continue after an error. The **-i** option performs this same task. The **make** program 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 user-defined dependency line exists for the given target. You can 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 by the Interrupt or Quit key. The pseudo-target name **.SILENT** has the same effect as the **-s** option.

Note: The Interrupt key is the Del (Delete) key on your keyboard. The Quit key is a combination of the Ctrl key and the \ key. Press and hold down the Ctrl key and press the \ key.

Using Macros

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

A macro definition is a line containing a name, an equal sign (=), and a value. You must not precede the equal sign with a colon or a tab. The name (string of letters and digits) to the left of the equal sign (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 a null string to `LIBES`. A macro that is never explicitly defined has a 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 have identical results.

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 that are linked and another macro for the names of the library.

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

If you invoke this `makefile`, it will load the three object files with the **lex** library specified with the `-lln` option.

You can 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 you use spaces in the definition, you must use double quotation marks to enclose the definition. Macros in a command line override corresponding definitions found in the makefile. For example, the following command loads and assigns the library options **-lln** and **-lm** to `LIBES`.

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

You can modify all or part of the value generated from a macro invocation without changing the macro itself by using the **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 that replace the modified characters. If *st2* is not given, *st1* is replaced by a null character.

The substitution sequence allows user-defined metacharacters in a makefile. For example, suppose that `.x` is used as a metacharacter for a prefix and suppose that a makefile contains the following definition:

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

Then, the macro invocation:

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

generates the value:

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

The actual value of `FILES` remains unchanged.

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

- \$\$** Contains the name of the current target with the suffix removed. Thus, if the current target is `prog.o`, **\$\$** contains **prog**. This macro can be used in dependency lines that redefine the built-in rules.
- \$\$@** Contains the full path name of the current target. It can be used in dependency lines with user-defined target names.
- \$\$<** Contains the file name of the dependent that is more recent than the given target. It can be used in dependency lines with built-in target names or with the **.DEFAULT** pseudo-target name.
- \$\$?** Contains the file names of the dependents that are more recent than the given target. It can be used in dependency lines with user-defined target names.
- \$\$%** Contains the file name of a library member. It can be used with target library names (see “Using Libraries” on page 4-18). 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 with the given file.

/dir/file

If the file is in the current directory, the macro contains a dot (`.`), the current directory designator. A macro with the **F** descriptor contains the name of the given file with the directory name removed. Do not use the **D** or **F** descriptor with the **\$?** macro.

For example, if you have a makefile with the target:

```
/usr/you/prog:    x.o
                  cc -o /usr/you/prog    x.o
                  echo "$@"
                  echo "$(@D)"
                  echo "$(@F)"
```

\$@ is the full path name of the current target. It has the value `/usr/you/prog`.

\$@ with a **D** descriptor produces the directory name for the current target. `$(@D)` has the value `/usr/you`.

\$@ with an **F** descriptor produces the file name for the current target. `$(@F)` has the value `prog`.

Using Shell Environment Variables

The **make** program provides access to current values of the shell's environment variables, such as `HOME`, `PATH`, and `LOGIN`. The **make** program 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
```

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

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

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

The **make** program 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 can 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 can assign any **make** options to `MAKEFLAGS` except `-f`, `-p` and `-d`. If you do not assign a value to the macro, **make** automatically assigns the current options to it, that is, the options given in the current invocation.

The `MAKE` and `MAKEFLAGS` variables, together with the `-n` option, are used to debug makefiles that generate entire software systems. For example, in the following recursive makefile, using the default for `MAKE` (the default being **make**) and invoking this

file with the **-n** option, displays all the commands used to generate the programs `prog1` and `prog2` without actually executing them.

```
prog1 : prog1.c  
        $(MAKE) -$(MAKEFLAGS) -f prog2
```

In this example, `prog2` can be any makefile in your directory.

Using Built-In Rules

The **make** program provides a set of built-in dependency lines, called built-in rules, that automatically check the targets and dependents given in a `makefile` and create up-to-date versions of these files, if necessary. The built-in rules are identical to user-defined dependency lines except that the suffix of the file name is the target or dependent instead of the file name 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 and then executes it. There are built-in rules for the following files.

- `.o` Object file
- `.c` C source file
- `.r` Ratfor 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, `x.c` is compiled. If there is also an `x.l`, that grammar would be run through **lex** before compiling the result.

The built-in rules are designed to reduce the size of your `makefiles`. They provide the rules for creating common files from typical dependents. Reconsider the example given in “Creating a Makefile” on page 4-2. In that 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 depended on the include file `defs`. In the original example, each dependency and corresponding command sequence was explicitly given. Many of these dependency lines were unnecessary, because 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 last line merely shows that two object files depend on the include file `defs`. No explicit command sequence is given to update these files. Instead, **make** uses the built-in rules to locate the desired C source files, compile these files, and create the necessary object files.

Changing Built-In Rules

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

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

The rules and macros are displayed at 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. The **make** program automatically assigns a default value to these macros when you start the program. You can change the values by redefining the macro in your `makefile`. For example, the following built-in rule contains three macros, `CC`, `CFLAGS`, and `LDFLAGS`.

```
.c :
    $(CC) $(CFLAGS) $(LDFLAGS) $< -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 form:

suffix-rule:
 command

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

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

<code>.c</code>	<code>.c~</code>
<code>.sh</code>	<code>.sh~</code>
<code>.c~.o</code>	<code>.c~.o</code>
<code>.c~.c</code>	<code>.s.o</code>
<code>.s~.o</code>	<code>.y.o</code>
<code>.y~.o</code>	<code>.l.o</code>
<code>.l~.o</code>	<code>.y.c</code>
<code>.y~.c</code>	<code>.l.c</code>
<code>.c~.a</code>	<code>.c~.a</code>
<code>.s~.a</code>	<code>.h~.h</code>

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

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

```
.c.o :  
    cc  $< -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 to make *suffix-rules* for the built-in rules. The line has the form:

```
.SUFFIXES: .suffix
```

where *suffix* is usually a lowercase letter preceded by a dot (.). If more than one suffix is given, 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 that has no list, all suffixes are ignored.

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 want 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` can be used for a rule that defines how to create a library member from a C source file. The dependent suffix in the new combination must be different from 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`, respectively.

Troubleshooting

Most difficulties in using **make** arise from the way **make** uses dependencies. If the file `x.c` has the 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`, while it is necessary to recreate `x.o`.)

To determine which commands **make** will execute, without actually executing them, use the **-n** option. For example, the command:

```
make -n
```

prints out the commands **make** would execute.

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

If a change to a file is absolutely certain to be benign (for example, adding a new definition to an include file), the **-t** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command:

```
make -ts
```

causes the relevant files to appear up-to-date. The **s**, following the **t**, causes **make** to touch silently.

An Example: Using make

An example of the use of **make** is shown at the end of this chapter. Examine the `makefile` used to maintain the **make** program itself. The code for **make** is located in a number of C source files and a **yacc** grammar. For more information on **yacc**, see Chapter 8, “The yacc Program Generator: A Compiler-Compiler.”

The **make** program usually prints out each command before issuing it. The following output results from typing the simple 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 the printing. The `$?` macro in the command line then picks up only the names of the files changed since `print` was touched. The printed output can be sent to a different printer or to a file by changing the definition of the `P` macro.

```
# 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

      @size make /usr/bin/make

      cp make /usr/bin/make ; rm make
```

```
print: $(FILES)          # print recently changed files

pr $? ] $P

touch print

test:

make -dp ] grep -v TIME >1zap

/usr/bin/make -dp ] grep -v TIME >2zap

diff 1zap 2zap

rm 1zap 2zap

lint : dosys.c doname.c files.c main.c misc.c vers.c gram.c

$(LINT) dosys.c doname.c files.c main.c misc.c\
        vers.c gram.c

rm gram.c

arch:

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

Chapter 5. SCCS: A Source Code Control System

Introduction

The Source Code Control System (SCCS) is a collection of commands that create, maintain, and control special files called SCCS files. The SCCS commands enable you to create and store multiple versions of a program or document in a single file, instead of one file for each version. With these commands you can retrieve any version 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 is useful wherever you require a compact way to store multiple versions of the same file. The SCCS provides an easy way to update any given version of a file and record the changes made. The commands are used to control changes to multiple versions of source programs, but they can also be used to control multiple versions of manuals, 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.

Basic Information

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

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

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 text editor such as `vi`. Special characters in the files are allowed only if they are also allowed by the given editor.

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

You must not use the `link` command to create multiple copies of an s-file.

Deltas and SIDs

Unlike an ordinary text file, an SCCS file (or s-file for short) contains nothing more than lists of changes. Each list includes the changes needed to precisely construct one version of the file. By combining the lists, SCCS can 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 numbers separated by periods. The string must have at least two numbers and may have as many as four. 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 can also have two optional numbers. The **branch number**, third in the string, indicates changes at a particular level, and the **sequence number**, fourth in the string, indicates changes at a particular branch. For example, the SIDs 1.1.1.1 and 1.1.1.2 indicate two new versions that contain slight changes to the original delta 1.1.

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

When you create a new version, the SCCS usually creates a new SID by increasing the level number of the original version. If you want 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.

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

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

SCCS Working Files

The SCCS uses several different kinds of files to complete its tasks. These files contain actual text or information about the commands in progress. For convenience, the SCCS 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 file name.
- 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 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 original file.
- g-file** An ordinary text file created by applying specific s-file deltas to the original file. The g-file represents a version of the original file, and, as such, receives the same file name as the original. The SCCS places the g-file in the current working directory of the user requesting 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, the 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 file name.
- z-file** A lock file used by SCCS commands to prevent two users from updating a single SCCS file at the same time. Before a command modifies an SCCS file, it creates a z-file and copies its own process ID to it. If a command tries to access a file while the file's z-file exists, SCCS displays an error message. When the original command finishes 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 file name.
- 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 file name.

-
- d-file** A temporary copy of the g-file used to generate a new delta.
- q-file** A temporary file used by the **delta** command when updating the p-file. The file is not directly accessible.

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 want to delete these files to ensure proper operation of subsequent SCCS commands.

SCCS Command Arguments

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

An option indicates a special action 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 file name indicates the file to be acted on. The syntax for SCCS file names is like other PC XENIX file name syntax. Appropriate path names 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 file name must not begin with a minus sign (-).

The special symbol (-) causes the given command to read a list of file names from the standard input. These file names 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 file names, so the options can appear anywhere on the command line.

File names are processed left to right. If an error causes the current process to stop, the command begins processing the next file.

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 access permission. Several SCCS commands let the administrator define who has access to the versions in a given s-file. These commands are described later.

Creating and Using s-files

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

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

Creating an s-file

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

```
admin -ifilename s.filename
```

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

```
#include <stdio.h>

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

To create an s-file, type:

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

This command creates 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, remove the original text file by using the **rm** command. This text file is no longer needed. If you want 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 you create an s-file with the **admin** command, SCCS may display the warning message:

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

This message can be ignored unless you have specifically included keywords in your file (for more information concerning keywords, see “Using Identification Keywords” on page 5-18).

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

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 latest version of the text file and copies it to a regular file. The file has the same name as the s-file but with the *s.* removed. It also has read-only file 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, type:

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

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

The command also displays a message that gives the SID of the retrieved file and the number of lines the file contains. For example, after retrieving the short C program from *s.demo.c*, the command displays the message:

```
1.1  
6 lines
```

You can 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 file name before the information.

Retrieving a File for Editing

You can retrieve a file for editing from a given s-file by using the **-e** 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 can give more than one file name if you want. If you do, you must separate each name with a space.

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

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

The command retrieves the program and copies it to the file named `demo.c`. Edit the file just as you do any other text file.

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

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

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

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

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.` prefix. The user should not attempt to access the p-file directly.

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` (that was retrieved from the file `s.demo.c`), type:

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

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

```
comments?
```

You can type any text (up to 512 characters) that you think is appropriate. 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 (\n). If you do not want to include a comment, just type (\n).

After you give 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 copies 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.%d ", 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 want to retrieve a previous version, you must use the **-r** option of the **get** command as described in the next section.

Retrieving a Specific Version

You can retrieve any version from an s-file by using the **-r** option of the **get** command. The command has the form:

```
get [ -e ] -rSID s.filename . . .
```

where **-e** is the edit option, *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 can give more than one file name. Separate the names with spaces.

The command retrieves the given version and copies it to a file that has the same name as the s-file but with the `s.` removed. The file has read-only permission unless you also give the `-e` option. If multiple file names 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 specify a version number that does not exist, the command displays an error message.

You can omit the level number of a version number and just give 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 the version 1.4.

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 -rrelnum s.filename . . .
```

where **-e** is the required edit option, **-rrelnum** gives the new release number of the file, and *s.filename* gives 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 can have this number. You can give more than one file name.

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.

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.

When you edit a version that already has a succeeding version, the **get** command 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 by using the **delta** command.

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 can 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 increasing 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.

If you omit the sequence number, **get** 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.

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 can combine the **-r** and **-t** options to retrieve the most recent version of a given release number. For example, if the most recent version with release number 3 is 3.5, 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 (for example, 3.2.1.5), then the above command retrieves the branch version and ignores version 3.5.

Displaying a Version

You can display the contents of a version at 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`. Similarly, the command:

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

displays version 2.1.

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 are copied to the standard error file.

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 can display the file, but you cannot edit the file.

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 can 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 suggest a possible remedy. For example, the command:

```
help col
```

displays the message:

```
col:  
"not an SCCS file"
```

This message means that a file you think is an SCCS file does not begin with the `s.` prefix. The **help** command can be used at any time.

Using Identification Keywords

The SCCS provides several special symbols, called identification keywords, that are 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 you retrieve a file for reading, the SCCS automatically replaces any keywords it finds in a given version of a file with the keyword value.

This section explains how keywords are treated by the various SCCS commands, and how you can use the keywords in your own files. Only a few keywords are described in this section. For a complete list of the keywords, see the **get** (CP) command in *XENIX Commands Reference*. The affect that s-file flags have on keywords is described in "Using s-file Flags" on page 5-20.

Inserting a Keyword into a File

You can insert a keyword into any text file. A keyword is simply an uppercase letter enclosed in percent signs (%). 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 you retrieve a program 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, the SID, and the current date in a program statement:

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

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

```
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 want to keep the keywords (and not their values) when you save the new version of the file.

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

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

This message is normally treated as a warning, letting you know that no keywords are present. However, you can change the operation of the system to make this situation a fatal error. This procedure is explained in "Forcing Keywords" on page 5-19.

Assigning Values to Keywords

The values of most keywords are predefined by the system, but some, such as the value for the `%M%` keyword, are 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 in the command:

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

In this example, `-f` is the option, **m** is the keyword, and `cdemo` is the value. This command records `cdemo` as the current value of the `%M%` keyword. If you do not set the **m** flag, the SCCS 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 a list of the corresponding keywords are in *XENIX Commands Reference*. The description is in the section **admin**(CP) and the list is under the **get**(CP) command.

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 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.

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

Using s-file Flags

An s-file flag is a special value that defines how a given SCCS command operates 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 **admin** (CP) command section in *XENIX Commands Reference*.

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 *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*.

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

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

that sets the **m** flag to the module name *dmod*.

Using the **i** Flag

If no keywords are found in the given text file, the **i** flag causes the **admin** and **delta** commands to print an error message and stop the current process. The flag is used to prevent a version of a file that 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.

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 that does not use the **-r** option retrieves version 1.1.

Using the **v** Flag

The **v** flag allows you to include modification requests in an s-file. Modification requests are names or numbers 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 use of the **-m** option in the **delta** and **admin** commands.

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 -dflag 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 -di s.demo.c
```

removes the **i** flag from the s-file *s.demo.c*. When removing a flag that 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.

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 can 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.

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 can 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"Added new routine." s.demo.c
```

saves the comment,

```
Added new routine.
```

in the s-file `s.demo.c`.

The `-y` option is 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.

Changing Comments

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

```
cdc -rSID s.filename
```

where *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 can type any sequence of characters up to 512 characters long. The sequence can contain embedded newline characters if 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.

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 that 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 can 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 set with the **-f** option. For example, the command:

```
admin -i def.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.

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 can type any number of requests. Each request can 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. 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. The response:

```
MRs? error36 !error35
```

adds the request `error36` and removes `error35`.

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

```
admin -tfilename s.filename
```

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

The file to be inserted can contain any amount of text. For example, the 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 can 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 file name. For example, the command:

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

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

Printing from an s-file

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

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 -dspec s.filename
```

where *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 file name 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 *XENIX Commands Reference*.

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 *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.

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 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 that precede version 1.4 (for example, 1.3, 1.2, and 1.1). The command:

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

prints information about versions that succeed version 1.4 (for example, 1.5, 1.6, and 2.1).

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

Editing by Several Users

The SCCS system allows any number of users to access and edit versions of a given s-file. Because 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 multiple edited versions.

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 edits version 1.1. There is no limit to the number of versions that can be edited at any given time.

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

Editing a Single Version

You can allow multiple users to edit a single version of a file 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 **j** 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 for editing version 1.4 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 can a version edited by two separate users result in a single new version.

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 -rSID s.filename
```

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

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

and

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

save version 1.5.

Protecting s-files

The SCCS system uses the normal system file permissions to protect s-files from changes made by unauthorized users. In addition to the 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 versions currently accessible to otherwise authorized users. The following sections explain how to set and use the user list and protection flags.

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 option causes the given name to be added to the user list. The user list defines who can access and edit the versions in the s-file. The command has the form:

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

where *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 can access and edit the files. When you explicitly give a user name or names, only those users can access the files.

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 *name* gives 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*.

Setting the Floor Flag

The floor flag, **f**, defines the release number of the lowest version a user can 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.

Setting the Ceiling Flag

The ceiling flag, **c**, defines the release number of the highest version a user can 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.

Locking a Version

The lock flag, **l**, lists by release number all versions in a given s-file that are locked against further editing. You can set the flag 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.

The special symbol **a** can be used to specify all release numbers. The command:

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

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

Repairing SCCS Files

The SCCS system carefully maintains all SCCS files; therefore, damage to the files very rare. However, damage can result from hardware malfunctions. This can 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.

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 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 can give more than one file name. If you do, the command checks each file. You can also give the name of a directory; in that 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.

Editing an s-file

When an s-file is 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 *XENIX Common File Tasks*.) If this is not possible, the file can be edited using a text editor.

To repair a damaged s-file, use the description of an s-file given in

the section **scsfile(F)** in *XENIX System Reference* to locate the part of the file that is damaged. Use extreme care when making changes; small errors can cause unwanted results.

Changing the Checksum of an s-file

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

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

The command computes and saves the new checksum and replaces the old sum.

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 **-k** option can regenerate a g-file that was accidentally removed or destroyed before it was saved.

Restoring a Damaged p-file

The **-g** option of the **get** command generates a new copy of a p-file that was 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, you must not use this command to change it.

Using other Command Options

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

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 the command syntax. For example, the command:

```
help rmdel
```

displays the message:

```
rmdel:
      rmdel -rSID file . . .
```

Creating a File with 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 file name. For example, the command:

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

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

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

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

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

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 -rrelnum s.filename
```

where *relnum* gives 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.

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's Program" s.demo.c
```

inserts the comment,

```
George's Program
```

in the new s-file `s.demo.c`.

The comment can 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 form:

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

is automatically inserted.

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.

Including and Excluding Deltas

You can explicitly define the deltas you want to include as well as the ones you want 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 can 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 be used 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 want 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. These changes can be saved immediately by issuing a **delta** command.

The **-x** option is useful for removing 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 had 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 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 where the problem exists. Corrective action, if required, is the responsibility of the user.

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 that contains the SIDs of all deltas used to create the given version.

The option creates a history of a given version's development. For example, the command:

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

creates a file named `l.demo.c` that contains 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 option performs the same function as the **-l** option except that 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.

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

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 used to review the history of each line in a given version.

Naming Lines

You can name each line in a given version with the current module name (that is, 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 indicates 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.

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 **diff** command.

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.

Removing a Delta

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

```
rmDEL -rSID s.filename
```

where *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 not remove a protected delta, that is, a delta whose release number is below the current floor value, above the current ceiling value, or equal to a current locked value (see “Protecting s-files” on page 5-32). The command cannot remove a delta that is currently being edited.

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

The **rmDEL** command 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 full in the section **delta** (CP) in *XENIX Commands Reference*.

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

Comparing SCCS Files

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

```
sccsdiff -rSID1 -rSID2 s.filename
```

where *SID1* and *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 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.

Chapter 6. The `adb` Program Debugger

Introduction

The `adb` program is a debugging tool for C and assembler 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 and stop 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.

Starting and Stopping `adb`

The `adb` program debugger 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 want to debug. The following sections explain how to start `adb` and describe the types of files available for debugging.

Starting with a Program File

You can debug any executable C or assembly language program file by typing a command line of the form:

```
adb [ filename ]
```

where *filename* is the name of the program file to be debugged. The **adb** program opens the file and prepares its text (instructions) and data for subsequent debugging. For example, the command:

```
adb sample
```

prepares the program named sample for examination and execution.

Once started, **adb** 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, **adb** displays an error message first, then waits 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 can also start **adb** without a file name. 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 typing:

```
adb a.out
```

The **adb** program debugger displays the prompt and waits for a command if the **a.out** file does not exist.

Starting with a Core Image File

The **adb** program debugger 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 of the error. Therefore, core image files 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 the program file. The command line has the form:

```
adb programfile corefile
```

where *programfile* is the file name of the program that caused the error, and *corefile* is the file name of the core image file generated by the system. The **adb** program debugger 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 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 file name. For example, the command:

```
adb sample -
```

prevents **adb** from searching your current working directory for a core file. 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`, type:

```
adb outdata
```

The **adb** program debugger opens this file and lets you examine its contents.

This method of examining files is very useful if the file contains non-ASCII data. The **adb** program debugger provides a way to look at the contents of the file in a variety of formats and structures. The **adb** command can display a warning when you give the name of 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.

Starting with the Write Option

If you open a program or data file with the **-w** option of the **adb** command, you can make changes and corrections to the file. For example, the command:

```
adb -w sample
```

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. For more information on the **-w** option, see “Patching Binary Files” on page 6-46.

Starting with the Prompt Option

The **-p** option allows you to define the prompt that **adb** uses. The option has the form:

-p *prompt*

where *prompt* is any combination of characters. If you use spaces, enclose the prompt in quotation marks. 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** displays an error message. The **adb** command automatically supplies a space at the end of the new prompt, so you do not have to supply one.

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 typing **Ctrl-D**.

You cannot stop **adb** by pressing the Interrupt (Del) or Quit (Ctrl \) keys. These keys cause **adb** to wait for a new command.

Displaying Instructions and Data

The **adb** program debugger 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 (=), question mark (?), and slash (/) tell **adb** from what source to take the item to be displayed.

The following sections explain how to form addresses, how to choose formats, how to use display commands, and how to form expressions.

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 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.

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
O	2 words 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 symbolic address
A	the current absolute address
n	a new line
r	a blank space
t	a horizontal tab

A format can 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 display **int** type variables; **D** and **X** display **long** variables or 32-bit values. The **f** and **F** formats display single- and double-precision, floating-point numbers. The **c** format displays **char** variables, and **s** is for arrays of **char** variables that end with a null character (null-terminated strings).

The **i** format displays machine instructions in 8086/286 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 can precede each format with a count of the number of times you want 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 following command displays four octal words followed by their ASCII interpretation from the data space of the core image file.

```
<b,-1/4o4^8Cn
```

In this example, the display starts at the address `<b`, the base address of the program's data. Since the negative count (`-1`) causes an indefinite execution of the command, the display continues until an error condition such as the end-of-the-file occurs. In the format, `4o` displays the next four words (16-bit values) as octal numbers. The `4^` then moves the current address back to the beginning of these four words, and `8C` re-displays them as 8 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 `@` itself is displayed as `@@` (double at sign).

Display Commands

The = (equal sign), ? (question mark), and / (slash) tell **adb** from what source to take the item to be displayed. The following sections explain how to use these display commands.

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). The following command displays (in decimal) the sum of the variable `b` and the hexadecimal value `0x2000`.

```
<b+0x2000=D
```

If a count is given, the same value is repeated that number of times. For example, the command below displays the value of `main` twice.

```
main,2=x
```

If no address is given, the current address is used instead. This is the same as the command:

```
.=
```

If no format is given, the previous format given for this command is used. For example in the following sequence of commands, both `main` and `start` are displayed in hexadecimal:

```
main=x
```

```
start=
```

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 want to display; and *format* is the format of the items you want to display.

The (?) command displays instructions in a given text segment. For example, the command:

```
main,5?ia
```

displays five instructions starting at the address, main. The command:

```
main,5?i
```

displays the instructions but no addresses other than the starting address.

The (/) command checks 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.

Forming Expressions

Expressions can 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 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 uppercase 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 must not use the decimal point when typing the number.

Symbols

Symbols are the names of global variables and functions defined within the program being debugged. Symbols are equal to the address of the given variable or function. They 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 *XENIX Commands Reference*.)

In expressions, you can spell the symbol exactly as it is in the source program or as it has been stored in 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`

If the spelling of any two symbols is the same (except for a leading underscore), **adb** ignores one of the symbols and allows 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.

Variables in adb

The **adb** program automatically creates a set of its own variables whenever you start the debugger. These variables are set to the addresses and sizes of various parts of the program file as defined below.

b	base address of the data segment
d	size of data segment
e	entry address of the program
m	execution type (magic number)
n	number of segments
s	size of stack segment
t	size of text segment

The **adb** program debugger 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 (**b**) 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 > *variablename*

where *expression* is the value to be assigned to the variable, and *variablename* is the variable to receive the value. The *variablename* must be a single letter. For example, the assignment:

0x2000>b

assigns to the variable *b* the hexadecimal value 0x2000.

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

The **adb** program debugger has two special variables that keep track of the last address used in a command and the last address 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 usually contain the same address except when implied commands, such as the newline and caret (^) characters, are used. These characters automatically increase and decrease the (.) variable but leave the (") variable unchanged.

Both the (.) and the (") variables can 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 typed.

Register Names

The **adb** program debugger 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 (<). The **adb** program debugger recognizes the following register names:

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

For example, the value of the **ax** register can be given as:

```
<ax
```

Register names cannot be used unless **adb** has been started with a core file, or the program is currently being run under **adb** control.

Operators

You can combine integers, symbols, variables, and register names with the following operators:

Unary

~	Not
-	Negative
*	Contents of location

Binary

+	Addition
-	Subtraction
*	Multiplication
%	Integer division
&	Bitwise AND
“	Bitwise inclusive OR
^	Modulo
#	Round up to the next multiple

Unary operators have higher precedence than binary operators. All binary operators have the same precedence. Thus, the binary expression:

$2*3+4$

is equal to 10, and the binary expression:

$4+2*3$

is 18.

You can change the precedence of the operations in an expression by using parentheses. For example, the expression:

$4+(2*3)$

is equal to 10.

Note that **adb** uses 32-bit arithmetic. This means that values that exceed 2,147,483,647 (decimal) are displayed as negative values.

The unary (*****) operator treats the given address as a pointer. An expression using this operator changes to the value pointed to by that pointer. For example, the expression:

`*0x1234`

is equal to the value at the address 0x1234, whereas:

`0x1234`

is just equal to 0x1234.

An Example: Simple Formatting

This example illustrates how to combine formats in the (?) or (/) commands to display different types of values when stored together in the same program. The program to be examined has the following source statements.

```
char   str1[ ] = "This is a character string";
int    one     = 1 ;
int    number  = 456 ;
long   lnum    = 1234 ;
float  fpt     = 1.25 ;
char   str2[ ] = "This is the second character string";

main() {

    one = 2;
}
```

The program is compiled and stored in a file named `sample`.

To start the session, type:

```
adb sample
```

You can display the value of each individual variable by giving its name and corresponding format in a (/) command. For example, the command:

```
str1/s
```

displays the contents of `str1` as a string:

```
_str1: This a character string:
```

and the command:

```
number/d
```

displays the contents of `number` as a decimal integer:

```
_number:          456.
```

You can choose to view a variable in a variety of formats. For example, you can display the **long** variable **lnum** as a 4-byte decimal, octal, and hexadecimal number by using the commands:

```
lnum/D
```

```
__lnum:          1234
```

```
lnum/O
```

```
__lnum:          02322
```

```
lnum/X
```

```
__lnum:          0x4d2
```

You can also examine all variables as a whole. For example, if you want to see them all in hexadecimal, type:

```
str1,5/8x
```

This command displays eight hexadecimal values on a line and continues for five lines.

Since the data contains a combination of numeric and string values, it is worthwhile to display each value as both a number and a character to see where the actual strings are located. You can do this with one command:

```
str1,5/4x4^8Cn
```

In this case, the command displays 4 values in hexadecimal, then displays the same values as 8 ASCII characters. The caret (^) is used four times just 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:

```
str1,5/4x4^8t8Cna
```

Debugging Program Execution

The **adb** program provides a variety of commands to control the execution of programs being debugged. The following sections explain how to use these commands as well as how to display the contents of memory and registers.

C does not generate statement labels for programs. This means it is not possible to refer to individual C statements when using the debugger. 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 assembler language listing of your C program before using **adb**. Then, refer to the listing as you use the debugger. To create an assembler language listing, use the **-S** option of the **cc** command (see “cc: a C Compiler” in *XENIX C Library Guide and Compiler Reference*).

Executing a Program

You can execute a program by using the **:r** or **:R** commands. The commands have 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 want to skip before one is taken; and *arguments* are the command line arguments, such as file names and options, that you want to pass to the program.

If no *address* is given, then the start of the program is used. Thus, to execute the program from the beginning type:

```
:r
```

If a *count* is given, **adb** ignores all breakpoints until the given number have been encountered. For example, the command:

```
,5:r
```

causes **adb** to skip the first 5 named 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 can use the shell redirection symbols if you want.

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 file names. 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 file names 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.

Setting Breakpoints

You can set a breakpoint in a program by using the **:br** command. Breakpoints cause execution to stop when the program 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 want the breakpoint to be skipped before it causes the program to stop; and *command* is the **adb** command you want 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 used within a function that is called several times during execution of a program, or within the instructions that correspond to a `for` or `while` statement. Such a breakpoint allows the program to continue to execute until the given function or instructions have been executed the specified number of times. For example, the command:

```
light,5:br
```

sets a breakpoint at the fifth invocation of the function `light`. The breakpoint does not stop the function until it has been called at least five times.

No more than 16 breakpoints at a time are allowed.

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.

Use the **\$b** command if you created several breakpoints in your program.

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 want to delete.

The **:dl** command deletes breakpoints you no longer want to use. The following command deletes the breakpoint that was set at the start of the function `main`.

```
main:dl
```

Continuing Execution

You can continue the execution of a program 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 want to continue execution; *count* is the number of breakpoints you want to ignore; and *signal* is the number of the signal to send to the program (see **signal** (S) in *XENIX System 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* breakpoints.

Single-Stepping a Program

You can single-step a program, that is, 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 want to execute, and *count* is the number of times you want 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`.

Stopping a Program with Interrupt and Quit

You can stop execution of a program at any time by pressing the Interrupt (Del) or Quit (Ctrl \) keys. These keys stop the current program and return control to **adb**; these keys are especially useful with programs that have infinite loops or other program errors.

Whenever you press the Interrupt (Del) or Quit (Ctrl \) keys to stop a program, **adb** automatically saves the signal. If you start the program again by using the `:co` command, **adb** automatically passes the signal to the program. This is very useful if you want to test a program that uses these signals as part of its processing.

If you want to continue execution of the program but do not want to send the signals, type:

```
:co 0
```

The command argument `0` prevents a signal from being sent to the program.

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 clears the current contents of the CPU registers and stack and begins the program again.

Displaying the C Stack Backtrace

You can trace the path of all active functions by using the **\$c** command. The command lists the names of all functions that have been called and have not yet returned control. It also lists the address from which each function was called and the arguments passed to each function.

For example, the command:

```
$c
```

displays a backtrace of the C language functions called.

By default, the **\$c** command displays all calls. If you want to display just a few, you must supply a count of the number of calls you want to see. For example, the command:

```
,25$c
```

displays up to 25 calls in the current call path.

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 does not appear in the list generated by the **\$c** command. You can fix this problem by placing breakpoints a few instructions into the function.

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. The display has the form:

```
ax      0x0          fl      0x0
bx      0x0          ip      0x0
cx      0x0          cs      0x0
dx      0x0          ds      0x0
si      0x0          ss      0x0
di      0x0          es      0x0
bp      0x0          sp      0x0

0:0:    addb    a1,b1
```

The value of each register is given in the current default format.

Displaying External Variables

You can display the values of all external variables in the program by using the `$e` command. External variables are the variables in your program that have global scope or have been defined outside of any function. This can include variables 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 a summary 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.
```

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 has the following source statements:

```
int      fcnt,gcnt,hcnt;
h(x,y)
int x,y;
{
    int hi; register int hr;
    hi = x+1;
    hr = x-y+1;
    hcnt++ ;
    hj:
    f(hr,hi);
}

g(p,q)
int p,q;
{
    int gi; register int gr;
    gi = q-p;
    gr = q-p+1;
    gcnt++ ;
    gj:
    h(gr,gi);
}

f(a,b)
int 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 the file named `sample`. To start the session, type:

```
adb sample
```

This starts **adb** and opens the corresponding program file. There is no core image file.

The first step is to set breakpoints at the beginning of each function. You can do this with the **:br** command. For example, to set a breakpoint at the start of the function **f**, type:

```
f:br
```

You can use similar commands for the **g** and **h** functions. Once you have created the breakpoints you can display their locations by typing:

```
$b
```

This command lists the address, optional count, and optional command associated with each breakpoint. In this case, the command displays:

```
breakpoints
count  bkpt          command
1      _f
1      _g
1      _h
```

The next step is to display the first five instructions in the **f** function. Type:

```
f,5?ia
```

This command displays five instructions, each preceded by its symbolic address. The instructions in 8086/286 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 `g` without their addresses by typing:

```
g,5?i
```

In this case, the display is:

```
_g:      push    bp
         mov     bp,sp
         mov     ax,4.
         call    near  __chkstk
         push    di
```

To start program execution, type:

```
:r
```

The **adb** program debugger 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 typing:

```
f:dl
```

and continue the program by typing:

```
:co
```

The **adb** program debugger displays the message:

```
sample: running
```

and starts the program at the next instruction. Execution continues until the next breakpoint where **adb** displays the message:

```
breakpoint      _g:      push    bp
```

You can now trace the path of execution by typing:

```
$c
```

The command shows that only two functions are active: `main` and `f`.

```
_f (1.,1.) from _main+13.  
  
_main (1., 7668., 7676.) from __start+45.  
  
__start() from start0+5.
```

Although a breakpoint has been set, function `g` is not listed in the backtrace until its first few instructions have been executed. To execute these instructions, type:

```
,5:s
```

The **adb** program single-steps the first five instructions. Now you can list the backtrace again. Type:

```
$c
```

This time the list shows three active functions:

```
_g (2.,3.) from _f+34.  
  
_f (1.,1.) from _main+13.  
  
_main (1., 7668., 7676.) from __start+45.  
  
__start() from start0+5.
```

You can display the contents of the integer variable `fcnt` by typing:

```
fcnt/d
```

This command displays the value of `fcnt` found in memory. The number 1 should be the value.

You can continue execution of the program and skip the first 10 breakpoints by typing:

```
,10:co
```

The **adb** program debugger starts the program and displays the running message again. It does not stop the program until exactly 10 breakpoints have been encountered. It displays the message:

```
breakpoint _g:      push    bp
```

To show that these breakpoints have been skipped, you can display the backtrace again using `$c`.

```
_f (2., 11.)          from _h+31:
_h (10., 9.)         from _g+33:
_g (11., 20.)       from _f+34:
_f (2., 9.)         from _h+31:
_h (8., 7.)         from _g+33:
_g (9., 16.)       from _f+34:
_f (2., 7.)         from _h+31:
_h (6., 5.)         from _g+33:
_g (7., 12.)       from _f+34:
_f (2., 5.)         from _h+31:
_h (4., 3.)         from _g+33:
_g (5., 8.)         from _f+34:
_f (2., 3.)         from _h+31:
_h (2., 1.)         from _g+33:
_g (2., 3.)         from _f+34:
_f (1., 7668., 7676) from ___start+45.
___start()          from start0+5.
```

Using the `adb` Memory Maps

The `adb` program debugger 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.

Displaying the Memory Maps

You can display the contents of the memory maps by using the `$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. It uses information taken from either the program and core files or directly from memory.

If you have started `adb` but have not executed the program, the `$m` command display has the form:

Text Segments

Seg #	File Pos	Phys Size	'sample' - File
63.	32.	2048.	
71.	2080.	656.	

Data Segments

Seg #	File Pos	Phys Size	'core' - File
39.	2736.	242.	

Each entry gives a segment number, file position, and physical size of a segment. The segment number is the starting address of the segment. The file position is the offset from the start of the file to the contents of the segment. The physical size is the number of bytes the segment occupies in the program or core file. The file names to the right of the display are the program and core file names.

If you have executed the program, the command display has the form :

```
Text Segments
Seg #   File Pos   Vir Size   'sample' - Memory
63.     32.           2048.
71.     2080.         656.
```

```
Data Segments
Seg #   File Pos   Vir Size   'sample' - Memory
39.     2736.         456.
```

where virtual size is the number of bytes the segment occupies in memory. This size is sometimes different from the size of the segment in the file and often changes as you execute the program. This difference is due to expansion of the stack or allocation of additional memory during program execution. The file names to the right always name the 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= 32.
Physical Size= 2048.
```

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* is the number of the segment map you want to change; *file-position* is the offset in the file to the beginning of the given address; and *size* is 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.

Validating Addresses

Whenever you use an address in a command, **adb** checks the address to make sure it is valid. The **adb** program debugger 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.

Miscellaneous Features

The following sections explain several of the commands and features of **adb**.

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 carry over to the next command. If an error occurs, the remaining commands are ignored.

One such combination is to place a **(?)** command after a **l** command. For example, the command:

```
?l 'Th' ; ?s
```

searches for and displays a string that begins with the characters Th.

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 file name. For example, to read commands from the file `script`, type:

```
adb sample <script
```

The file you supply must contain valid **adb** commands. Such files are called script files and can be used with any invocation of the debugger.

Read commands from a script file when you want to use the same set of commands on several different object files. Scripts display the contents of core files after a program error. For example, a file containing the following commands is used to display most of the relevant information about a program error:

```
120$w
4095$s
$v
=3"
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3"Data Segment"
<b,-1/8xna
```

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 giving the width in characters of the display. You can give any width convenient for your terminal or display device. The default width when **adb** is first invoked is 80 characters.

The command is used when redirecting output to a line printer or special terminal. For example, the command:

```
120$w
```

sets the display width to 120 characters, a common maximum width for line printers.

Setting the Maximum Offset

The **adb** program debugger normally displays memory and file addresses as the sum of a symbol and an offset. This helps you 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 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.

You can disable all symbolic addressing by setting the maximum offset to zero. All addresses are given numeric values instead.

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 want to work with a combination of decimal, octal, and hexadecimal numbers. For example, if you use:

```
$x
```

you can give addresses in hexadecimal numbers without preceding each address with the 0x identifier. Furthermore, **adb** displays all numbers in hexadecimal format except those specifically requested to be in some other format.

When you first start **adb**, the default format is decimal. You can change this at any time and restore it as necessary using the **\$d** command.

Using PC XENIX Commands

You can execute PC XENIX commands without leaving **adb** by using the **adb** escape command (!). The escape command has the form:

```
! command
```

where *command* is the PC XENIX command you want to execute. The command must have any required arguments. The **adb** program debugger passes this command to the system shell that executes it. When finished, the shell returns control to **adb**.

For example, to display the date type:

```
! date
```

The system displays the date at your terminal and restores control to **adb**.

Computing Numbers and Displaying Text

You can perform arithmetic calculations while in **adb** by using the equal (=) command. The command directs **adb** to display the value of an expression in a given format.

The command converts numbers in one base to another, double-checks the arithmetic performed by a program, and displays 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 can have any combination of symbols and operators. For example, the command:

```
<d0-12*<d1+<b+5=X
```

computes a value using the contents of the d0 and d1 registers and the **adb** variable *b*. You can also compute the value of external symbols as in the command:

```
main+5=X
```

This checks the hexadecimal value of an external symbol address.

The (=) command can also be used to display literal strings at your terminal. This is especially useful in **adb** scripts 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.

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 PC XENIX file system. The directory file is assumed to be named `dir` and contains a variety of files. The PC 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. A directory file normally contains one or more entries. Each entry consists of an unsigned inode number (`inumber`) and a 14-character file name. You can display this information by including a command in your script file. The following command, for example, displays the first 20 entries, separating the inode number and file name with a tab:

```
0,20?ut14cn
```

You can change the second number 20 to specify the number of entries in the directory. 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, redirect it as input when you start **adb** with the name of your directory. For example, type:

```
adb dir - <script
```

(The hyphen (-) is used to prevent **adb** from attempting to open a core file.) The **adb** program debugger reads the commands from the script and the resulting display has the form:

```
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 file name of the device associated with the file system (for example, the fixed-disk drive).

The inode table of a file system has a complex structure. Each entry contains:

- A word value for the file's status flags
- A byte value for the number links
- 2-byte values for the user and group IDs
- A byte and word value for the size
- 8-word values for the location on disk of the file's blocks
- 2-word values for the creation and modification dates.

The inode table starts at the address 02000. You can display the first entry by typing:

```
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, type:

```
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
```

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.

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 values.

The **?l** command 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.

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 want to change, and *value* is the new value. The **w** command writes 2-byte values; **W** writes 4-byte values. For example, the following commands change the word `This` to `The`:

```
?l 'Th'
```

```
?W 'The'
```

The **W** changed all four characters.

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 to the file. This command is useful if you want to make changes to a program's data as it runs, for example, to temporarily change the value of program flags or constants.



Chapter 7. The `lex` Program: A Lexical Analyzer

Introduction

The `lex` program generator is designed to construct programs 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 in the input stream. The regular expressions are specified by the user in the source specifications given to `lex`. The `lex` code recognizes these expressions and partitions the input stream into strings matching the expressions. Program sections provided by the user are executed at the boundaries between strings. 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 can supply the additional code needed to complete the tasks, including code written by other generators. The program that recognizes the expressions is generated from the user's C program fragments. The `lex` program is not a complete language, but a generator representing a new language feature added on top of the C programming language.

The `lex` program generator turns the user's expressions and actions (called *source* in this chapter) into a C program named `yylex`. The `lex` program generator then uses the `yylex` program to recognize expressions in a stream (called *input* in this chapter) and to perform the specified actions for each expression as it is detected.

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

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

are all that is required.

The program contains a double percent sign (%%) delimiter to mark the beginning of the rules statement in the program. The rule in this example 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 plus sign (+) 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` ignores these characters. Everything else is 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. It determines whether or not there is a newline character at the end of the string of blanks or tabs, and then executes the desired 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.

The `lex` program generator is used alone for simple transformations, or for analysis and statistics gathering on a lexical level. The `lex` program is also used with a parser generator to perform the lexical analysis phase. It is especially easy to interface `lex` and `yacc`. The `lex` program recognizes only regular expressions; `yacc` writes parsers that accept a large class of context-free grammars, but that require a lower-level analyzer to recognize input tokens. Thus, a combination of `lex` and `yacc` is often appropriate. When used as a preprocessor for a later parser generator, `lex` partitions 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`. The programs `lex` and `yacc` are often used together. Users of `yacc` 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.

The **lex** program generates a deterministic finite automaton from the regular expressions in the source. To save space, the automaton is interpreted rather than compiled. 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 of **lex** rules or the complexity of the rules is not important in determining speed, unless rules that include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton and, therefore, the size of the program generated by **lex**.

In the program written by **lex**, the fragments left for the user (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the flow of control. 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.

The **lex** program generator 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 `abcdefgh`, **lex** recognizes `ab` and leaves the input pointer just before `cd`. Such backup is more costly than the processing of simpler languages.

Invoking `lex`

There are two steps in compiling a `lex` source program. First, the `lex` source must be turned into a C language program. 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 on the usual file `a.out` for later execution. To use `lex` with `yacc` see “Using `yacc` with `lex`” on page 7-28. Also, refer to Chapter 8, “The `yacc` Program Generator: A Compiler-Compiler” on page 8-1. Although the default `lex` I/O routines use the C standard library, the `lex` automata themselves do not do so. If private versions of `input()`, `output()` and `unput()` are given, the library can be avoided.

The lex Source Format

The general format of **lex** source is:

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

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum **lex** program is:

```
%%
```

(no definitions, no rules) which translates into a program that copies the input to the output unchanged.

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

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

This looks for the string `integer` in the input stream and prints the message:

```
found keyword INT
```

whenever it appears in the input text. In this example the C library function `printf()` prints the string. The end of the **lex** regular expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. The **lex** program generator rules such as:

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

would be a start. These rules are not quite enough, since the word `petroleum` would become `gaseum`; a way of dealing with

such problems is described in “Handling Ambiguous Source Rules” on page 7-19.

The 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:

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

Using the Operator Characters

The `lex` program can match any set of strings that you specify using the operator characters. The following sections describe ways of specifying the strings you want matched.

Specifying a Literal Use

If any of the operator characters are 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. Part of a string can be quoted. It is harmless but unnecessary to quote an ordinary text character. The expression:

```
"xyz++"
```

is the same as the one above. Thus, to keep from memorizing the above list of current operator characters, you can quote every non-alphanumeric character used as a text character.

An operator character can also be turned into a text character if you precede it with a backslash (`\`) as in:

```
xyz\+\+
```

That is another equivalent of the above expressions, although it is more difficult to read. The quoting mechanism is also used to get a blank into an expression. Normally, as explained above, 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 an illegal expression, a (`\n`) must be used. You do not need to escape (`\`) a tab or backspace. Every character

except blank, tab, newline, backspace, and backslash is always a text character.

Specifying Character Classes

Classes of characters can be specified by enclosing them within a left bracket and a right bracket. The construction:

```
[abc]
```

matches a single character that can be a, b, or c. Within square brackets, most operator meanings are ignored. Only three characters are special: the dash (-), the caret (^), and the backslash (\).

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 can be given in either ascending or descending 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 you want to include the dash in a character class, the dash should be first or last; thus:

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

matches all the digits and the minus and plus signs.

In character classes, the caret (^) operator must appear as the first character after the left bracket. The caret 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).

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

In the above example, any character that is not a letter is matched.

Specifying an Arbitrary Character

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

Escaping into the octal format is possible although nonportable. For example:

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

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

To match almost any character, the period (`.`) designates the class of all characters except a newline.

Specifying Optional Expressions

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

```
ab?c
```

matches either `ac` or `abc`. Here the meaning of the question mark differs from its meaning in the shell.

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

Specifying Alternation and Grouping

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

`(ab|cd)`

matches either `ab` or `cd`. 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 are necessary for more complex expressions, such as:

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

to match such strings as `abefef`, `efefef`, `cdef`, and `cddd`, but not `abc`, `abcd`, or `abcdef`.

Specifying Context Sensitivity

The **lex** program generator 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 use of the caret can never conflict with the other meaning of the caret (complementation of character classes) because complementation only applies within brackets. If the very last character is 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, and 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 “Specifying Left Context Sensitivity” on page 7-23. 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 the beginning of a line starts condition `ONE`, then the caret (^) operator is equivalent to:

`<ONE>`

Start conditions are explained more in “Specifying Left Context Sensitivity” on page 7-23.

Specifying Expression Repetition

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

```
{digit}
```

looks for a predefined string named `digit` and inserts it at that point in the expression. The definitions are given before the rules in the `lex` input. In contrast,

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

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

Specifying Separate Segments

An initial percent sign (`%`) is the separator for `lex` source segments.

Writing Actions

When an expression is matched by a pattern of text in the input, **lex** executes the corresponding action. Actions are written in the rules section of the **lex** source format. This section of the guide describes some features of **lex** that aid in writing actions. There is a default action that consists of copying the input to the output; therefore, a rule that merely copies can be omitted. This is performed on all strings not otherwise matched. Thus to absorb the entire input without producing any output, provide rules to match everything. The **lex** program used with **yacc** is the normal situation.

One of the simplest things that you can do is to ignore input by specifying a C null statement (;) as an action to a rule. In the following example, the null action causes the three spacing characters (blank, tab, and newline) to be ignored.

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

Another easy way to avoid writing actions is to use the repeat action character (!). The (!) tells **lex** to group this rule with the next until an action is stated. The **lex** program then performs the action on all strings that matched each rule. Therefore, you can also write the previous example as follows:

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

The result is the same although the style is different. The quotation marks around (**\t**) and (**\n**) are not required.

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

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

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

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

prints the string in **yytext**. The C function **printf** accepts a format argument and data to be printed. In this case, the format is *print string* where the percent sign (%) indicates data conversion, the *s* indicates string type, and the data are the characters in **yytext**. This rule places the matched string on the

output and is so common that it is written as ECHO. For example:

```
{a-z}+ ECHO;
```

has the same result as the preceding example. Since the default action is to print the characters found, one might ask the reason for giving a rule that merely specifies the default action. Such a rule is often required to avoid matching some other rule that is not desired. For example, if there is a rule that matches `read`, it matches the instances of `read` contained in `bread` or `readjust`. To avoid this, a rule of the following form is needed.

```
{a-z}+
```

The `lex` program also provides a count of the number of characters matched. This count is kept in the variable `yylen`. To count both the number of words and the number of characters in words in the input, you might write:

```
{a-zA-Z}+      {words++; chars += yylen;}
```

which accumulates in the variable `chars` the number of characters in the words recognized. The last character matched in the string can be accessed with:

```
yytext[yylen-1]
```

Occasionally, `lex` will detect 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 the input. (Normally, the next input string overwrites the current entry in `yytext`.) Second, `yyless(n)` can be called to indicate that not all the characters matched by the currently successful expression are wanted 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 procedure provides the same sort of lookahead ability 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 specifies that a quotation mark in a string must be preceded by a backslash (\). The regular expression that matches this is somewhat confusing, so that it might be preferable to write:

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

When faced with a string such as:

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

the **lex** program first matches the five characters:

```
"abc\"
```

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

```
"def
```

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

The function **yyless()** might be used to re-process 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 print a message. A rule might be:

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

The expressions for the two cases might more easily be written:

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

in the first case and

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

in the second. No backup is required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of ==-3, however, makes the following a better rule:

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

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

- **input()** which returns the next input character
- **output (c)** which writes the character *c* on the output
- **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 they must all be retained or modified consistently. They can be redefined to cause input or output to be transmitted to or from strange places, including other programs or internal memory. However, 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 look-ahead does not work. The **lex** program generator 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 look-ahead:

```
+ * ? $
```

Look-ahead is also necessary to match an expression that is a prefix of another expression. 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 wrap up 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 a 0. This instructs **lex** to continue processing. The default **yywrap()** always returns a 1.

This library routine is also a convenient place to print such things as tables and summaries at the end of a program. You cannot 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, because a value of 0 returned by **input()** is taken to be end-of-file.

Handling Ambiguous Source Rules

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

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

For example, suppose the following rules are given:

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

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

```
[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 (for example, `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, the rule:

```
'.*'
```

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'
```

and that is probably not what was wanted. A better rule is of 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. Don't try to overcome this fact with expressions like:

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

or their equivalents. The **lex** generated program tries to read the entire input file, causing internal buffer overflows.

The **lex** program generator is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for only once. For example, suppose you want to count occurrences of both `she` and `he` in an input text. Some **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 character. Since `she` includes `he`, **lex** does not recognize the instances of `he` included in `she`, since once it has passed a `she` those characters are gone.

Sometimes you may want to override this precedent just mentioned. The action **REJECT** sends **lex** to the next rule. The position of the input pointer is adjusted accordingly. To count the included instances of `he`:

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

This group of rules is one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expressions are then counted. In this example, of course, you could note that `she` includes `he`, but not vice versa, and omit the **REJECT** action on `he`; in other cases, however, 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 4 characters and then the second rule for 3 characters. In contrast, the input `accd` agrees with the second rule for 4 characters and then the first rule for 3 characters.

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. 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` to be incremented, the appropriate source is:

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

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

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

Specifying Left Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be 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, for example, is a prior context operator that recognizes 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:

- Flags, when only a few rules change from one environment to another
- Start conditions with rules
- 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. These rules also set some parameters to reflect the change. Using a flag explicitly tested by the user's action code is the simplest way of dealing with the problem, since `lex` is not involved at all. It can be more convenient, however, to have `lex` remember the flags as initial conditions on the rules. Any rule can be associated with a start condition. It is only recognized when `lex` is in that start condition. The current start condition can be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity can be best achieved by writing several distinct lexical analyzers and switching from one to another as desired.

Consider the following problem:

1. Copy the input to the output.
2. Change the word `magic` to `first` on every line that begins with the letter `a`.
3. Change `magic` to `second` on every line that begins with the letter `b`.

-
4. Change magic to third on every line that begins with the letter c.
 5. Leave all other words and all other lines unchanged.

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

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

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

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

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

```
<name1>expression
```

is a rule recognized only 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, use:

```
BEGIN 0;
```

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

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

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

The example changing magic can now be written:

```
%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.

Specifying Source Definitions

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

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

So far only the rules have been described. You need additional options, however, to define variables for use in your program and for use by **lex**. You can write options 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 source **lex** will not intercept:

1. Any line that is not part of a **lex** rule or action and that begins with a blank or tab is copied into the **lex**-generated program. Such source input prior to the first **%%** delimiter is external to any function in the code; if the input 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.

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

2. Anything included between lines containing only **{** and **}** is copied to the program. The delimiters are discarded. You can use this format for entering preprocessor statements that must begin in column 1, or copying lines that do not look like programs.
3. Anything after the second **%%** delimiter, regardless of format, is copied to the generated program 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

The definition 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.

Note:

1. The **lex** program considers blanks and tabs that follow the translation as part of the translation. This situation may cause errors.
2. A **lex** substitution string name may not be used inside of [] (brackets) because the brackets remove the meaning of the { } operators. For example, if you define A to be 0-9, [{A}] is not equivalent to [0-9].

Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

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

The first two rules for real numbers require that the string have a decimal point and contain an optional exponent field. The first rule 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 (that does not contain a real number), a context-sensitive rule such as the following could be used.

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

This rule is in addition to the normal rule for integers.

The definitions section can also contain other commands, including a character set table, a list of start conditions, or adjustments to the default size of arrays within **lex** itself for larger source programs. These possibilities are discussed in "Source Format" on page 7-31.

Using yacc with lex

The default output for **lex** is **yylex()**. The **yacc** program requires the input from **lex** to have the name **yylex()**. When you use **yacc** with **lex**, you must end each **lex** rule with:

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to the names **yacc** uses 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. If the grammar is named `good` and the lexical rules to be named `better`, the PC XENIX command sequence is:

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

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

Specifying Character Sets

The programs generated by **lex** handle character I/O only through the routines **input()**, **output()**, and **unput()**. Thus the character representation provided in these routines is accepted by **lex** and employed to return values in **yytext**. For internal use, a character is represented as a small integer which, if the standard library is used, 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 as 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 being given 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 form:

```
{integer} {character string}
```

that indicate the value associated with each character.

For example, the following table maps the lowercase and uppercase letters together into the integers 1 through 26, newline into 27, plus (+) and minus (-) into 28 and 29, and the digits into 30 through 39. Observe the escape for 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 can be assigned the number 0, and no character can be assigned a larger number than the size of the hardware character set.

%T	
1	Aa
2	Bb
. . .	
26	Zz
27	\n
28	+
29	-
30	0
31	1
. . .	
39	9
%T	

Source Format

The general form of a **lex** source file is:

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

The definitions section contains a combination of:

- Definitions, in the form:

name translation

- Included code, in the form:

code

Note: *code* must be preceded by a space.

- Included code, in the form:

```
%{
code
%}
```

- Start conditions, given in the form:

```
%S name1 name2 ...
```

-
- Character set tables, in the form:

```
%T
number character-string
%T
```

- 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
t	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form:

```
expression action
```

where the action is continued on succeeding lines by using braces to delimit it.

Regular expressions in **lex** use the following operators:

<code>x</code>	The character <code>x</code> .
<code>"x"</code>	An <code>x</code> , even if <code>x</code> is an operator.
<code>\x</code>	An <code>x</code> , even if <code>x</code> is an operator.
<code>[xy]</code>	The character <code>x</code> or <code>y</code> .
<code>[x-z]</code>	The characters <code>x</code> , <code>y</code> or <code>z</code> .
<code>[^x]</code>	Any character but <code>x</code> .
<code>.</code>	Any character but newline.
<code>^x</code>	An <code>x</code> at the beginning of a line.
<code><y>x</code>	An <code>x</code> when lex is in start condition <code>y</code> .
<code>x\$</code>	An <code>x</code> at the end of a line.
<code>x?</code>	An optional <code>x</code> .
<code>x*</code>	0,1,2, ... instances of <code>x</code> .
<code>x+</code>	1,2,3, ... instances of <code>x</code> .
<code>x y</code>	An <code>x</code> or a <code>y</code> .
<code>(x)</code>	An <code>x</code> .
<code>x/y</code>	An <code>x</code> but only if followed by <code>y</code> .
<code>{xx}</code>	The translation of <code>xx</code> from the definitions section.
<code>x{m,n}</code>	<i>m</i> through <i>n</i> occurrences of <code>x</code> .

A lex Example

The example to follow is a suitable **lex** source program that copies an input file while adding 3 to every positive number divisible by 7.

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

The rule `[0-9]+` recognizes strings of digits; `atoi()` converts the digits to binary and stores the result in `k`. The remainder operator (`%`) checks whether `k` is divisible by 7; if it is, it is incremented by 3 as it is written out. An objection can be raised that this program alters 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. For example:

```
%%
    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 are 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 gathering statistics, here is a program that makes histograms of word lengths, where a word is defined as a string of letters.

```
        int lengs[100];
%%
[a-z]+  lengs[yyvaleng]++;
.      |
\n     ;
%%
yywrap()
{
    int    i;

    printf("Length  No. words\n");
    for(i = 0; i < 100; i++)
        if (lengs[i] > 0)
            printf("%5d%11d\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 wrap up. If `yywrap()` returns zero (false), it implies that further input is available and the program is to continue reading and processing. 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 uppercase and lowercase letters, this routine begins by defining a set of classes including both cases of each letter:

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

An additional class recognizes 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:

```
^"      "[^ O]      ECHO;
```

In the regular expression, the quotation marks surround the blanks. This expression is interpreted as:

Beginning of line, then

Five blanks, then

Anything but blank or zero.

Two different meanings of the caret (^) are used in this example. The first to specify beginning of line and the next to specify a character class.

The example to follow shows some rules to change double-precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+      |
[0-9]+{W}."{W}{d}{W}[+-]?{W}[0-9]+  |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+  {
    /* convert constants */
    char *p;
    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'
```

and then converts the **d** or **D** to the next letter of the alphabet. The modified constant, now single-precision, is written out again. A series of names follow that must be respelled 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 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 initial `d` changed to initial `r`:

```
{d}l{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 ;
```

This program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

Chapter 8. The yacc Program Generator: A Compiler-Compiler

Introduction

The **yacc** program generator provides a general tool for describing input to a computer program. The name **yacc** stands for *yet another compiler-compiler*. The **yacc** user specifies the structures of the input and the code to be invoked when each structure is recognized. The **yacc** program generator turns the specification into a subroutine that handles the input process.

The input subroutine produced by **yacc** calls a user-supplied routine to return the next basic input item. Thus, the user can specify input in terms of individual input characters or in terms of higher-level constructs such as names and numbers. The user-supplied routine can also handle idiomatic features such as comment and continuation conventions, and these defy easy grammatical specification. The class of specifications **yacc** accepts is a general one: LALR grammars with disambiguating rules. (LALR is a look-ahead-left-to-right type of parsing mechanism. A *disambiguating* rule is a rule that describes what choice to make in a given situation.)

In addition to compilers for C, APL, Pascal, RATFOR, etc., **yacc** has also been used for less conventional languages. The **yacc** program has been used to write a phototypesetter language, several desk calculator languages, a document retrieval system, and a FORTRAN debugging system.

The **yacc** program generator 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 specification includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to process the basic input. The **yacc** program generator 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. The parser organizes these

tokens according to the user-supplied input structure rules, called grammar rules. When **yacc** recognizes one of these rules, it invokes user-supplied code (called an *action*). Actions have the ability to return values and use the values of other actions.

The **yacc** program generator is written in a portable dialect of C. The actions and output subroutine are also written in this portable dialect. Moreover, many of the syntactic conventions of **yacc** follow C language conventions.

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; `month_name`, `day`, and `year` must be previously defined. The comma (,) is enclosed in single quotation marks; this implies that the comma is to appear literally in the input. The colon and semicolon serve as punctuation in the rule; they have no significance in controlling the input. Thus, with proper definitions, the input:

```
July 4, 1776
```

is matched by the above rule.

The lexical analyzer carries out an important part of the input process. This routine reads the input stream, recognizing the lower-level structures, and communicates these structures or tokens to the parser. A *terminal symbol* is a structure that is recognized by the lexical analyzer. A *nonterminal symbol* is a structure that is recognized by the parser. To avoid confusion, this manual refers to terminal symbols as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. The following example uses the parser to recognize the structure `month_name`.

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;
.
.
.
month_name : 'D' 'e' 'c' ;
```

Quoting each letter of the `month_name` causes the analyzer to recognize only individual letters. Because each letter is

recognized, `month_name` is considered a nonterminal symbol. Such low-level rules tend to waste time and space. Such rules can also complicate the specification beyond `yacc`'s ability to deal with it. Usually, the lexical analyzer recognizes the month names and returns an indication that a `month_name` was read. In this case, `month_name` is considered a token.

Literal characters, such as the comma, must be passed through the lexical analyzer and are also considered tokens.

Specification files are very flexible. It is relatively easy to add the following rule to the above example:

```
date : month '/' day '/' year ;
```

Adding this rule allows

```
7/4/1776
```

as a synonym for

```
July 4, 1776
```

In most cases, you can slip this new rule into a working system with minimal effort and with little danger of disrupting existing input.

The analyzer reads input with a left-to-right scan. It quickly detects input that does not conform to the given specifications. Because of this early error detection, there is less chance of processing bad data, and errors are found quickly. 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 can be self contradictory, or they may require a more powerful recognition mechanism than is available to `yacc`. Often, you can correct this problem by rewriting some of the grammar rules. While `yacc` cannot handle all possible specifications, its power compares favorably with similar systems. The constructions that are difficult for `yacc` to handle are also frequently difficult for users 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.

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
- Some reasons why **yacc** may be unable to produce a parser from a specification
- A simple mechanism for handling operator precedences (order of arithmetic operation) in arithmetic operations
- Error detection and recovery
- The operating environment and special features of the parsers **yacc** produces
- Suggestions to improve the style and efficiency of the specifications.

Specifications

Names refer to either tokens or nonterminal symbols. The **yacc** program requires token names to be declared as such. It is often desirable to include the lexical analyzer as part of the specification file. Including other programs in the specification file can also be useful. 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 looks like:

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty and the programs section may be omitted. If you omit the programs section, omit the second %% . The smallest legal **yacc** specification is:

```
%%
rules
```

The **yacc** program ignores blanks, tabs, and newlines in the specifications. They must not, however, appear in names or multicharacter reserved symbols. You may use comments wherever the use of a name is legal, but you must enclose all comments within /* and */.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

A : *BODY* ;

The *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 are of any length and may consist of letters, the dot (.), the underscore (_), and digits; however, names must not begin with digits. Uppercase and lowercase letters are distinct. The names in the body of a grammar rule represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotation marks (''). A list of special literals is provided below. As in the C language, the backslash (\) is an escape character within literals, and **yacc** recognizes all of the C language escapes.

'\n'	Newline
'\r'	Return
'\''	Single quotation mark
'\\'	Backslash
'\t'	Tab
'\b'	Backspace
'\f'	Form feed
'\xxx'	"xxx" in octal

Note: Never use the ASCII NUL character ('\0' or 0) in grammar rules.

If several grammar rules have the same left side, the vertical bar (|) can be used to avoid rewriting the left side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Instead of using this form:

```
A : B C D ;
A : E F ;
A : G ;
```

you can specify the same rules as follows:

```
A : B C D
   | E F
   | G
   ;
```

All grammar rules with the same left side do not need to appear together. However, putting them together makes the input much easier to read and easier to change.

If you want a nonterminal symbol to match an empty string, you can specify the match like this:

```
symbol : ;
```

You must declare names representing tokens in the declarations section. For example:

```
%token name1 name2 . . .
```

Every nonterminal symbol must appear on the left side of at least one rule. Of all the nonterminal symbols, the start symbol is the most important. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure you can describe by grammar rules. By default, the start symbol is the left side of the first grammar rule in the rules section. You can declare the start symbol explicitly in the declarations section using the `%start` keyword:

```
%start symbol
```

A special token, called an *endmarker*, signals the end of the input to the parser. If the tokens up to the endmarker form a structure that matches the start symbol, the parser function returns to its caller after the endmarker is read and the parser accepts the input. If the parser reads the endmarker before it matches the start symbol, the parser displays an error message.

Usually the endmarker represents some reasonably obvious I/O status, such as the end of the file or end of the record. It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate.

Actions

When **yacc** matches a grammar rule in the input stream, it performs the actions you specify. These actions can return values and can obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is a statement written in C language conventions. As such, an action can process input and output, call subprograms, and alter external vectors and variables. An action is identified by one or more statements, enclosed in { } (curly braces). For example:

```
A : '(' B ')'  
    {      hello( 1, "abc" ); }
```

and

```
XXX : YYY ZZZ  
    { printf("a message\n");  
      flag = 25; }
```

are grammar rules with actions.

To aid communication between the actions and the parser, use the dollar sign (\$) as a signal to **yacc**.

To return a value, the action statement normally sets the pseudo-variable \$\$ to some value. For example, an action that returns the value 1 is:

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action statement uses the pseudo-variables:

```
$1, $2, . . .
```

These pseudo-variables 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.

Consider the rule:

```
expr : '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. You can indicate this as follows:

```
expr : '(' expr ')' { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in that rule (\$1). Thus, grammar rules of the form:

```
A : B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, you may want to change the process control before a rule is fully parsed. The **yacc** program generator permits you to write an action in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual \$ mechanism, by the actions to the right. In turn, the action may access the values returned by the symbols to its left. Thus, in the rule:

```
A : B                               /* value referred to by $1 */
  { $$ = 1; }                       /* value referred to by $2 */
  C                                 /* value referred to by $3 */
  { x = $2; y = $3; }
  ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

The **yacc** program handles actions that do not terminate a rule by manufacturing a new nonterminal symbol name and a new rule matching this name to an empty string. The action triggered by matching this added rule is an *interior action*. The **yacc** program generator treats the above example as if it had been written:

```
$ACT : /* empty */
      { $$ = 1; }
      ;
A    : B $ACT C
      { x = $2; y = $3; }
      ;
```

In many applications, the actions do not directly cause the output. Instead, 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 if you give 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` and descendants `n1` and `n2`, and returns the index of the newly created node. Then you can build a parse tree by supplying actions such as:

```
expr : expr '+' expr
      { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user can define variables other than `$$` 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 the action statements and the lexical analyzer recognize them. For example, you can place the declaration:

```
%{ int variable = 0; %}
```

in the declarations section, making `variable` accessible to all of the actions. The `yacc` parser uses only names beginning in `yy`; therefore, you should avoid using names that begin with `yy`.

Lexical Analysis

You 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**. This analyzer returns an integer, called a *token number*. The token number represents the kind of token that the analyzer read. If a value is associated with that token, that value should be assigned to the external variable **yylval**.

In order for the parser and lexical analyzer to communicate with each other, they must identify the input stream with the same token numbers. Either you or **yacc** can choose the numbers. 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:

```
yylex(){
    extern int  yyval;
    int  c;
    . . .
    c = getchar();
    . . .
    switch( c ) {
        . . .
    case '0':
    case '1':
        . . .
    case '9':
        yyval = 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. If you place this lexical analyzer code in the programs section of the specification file, the identifier `DIGIT` will be defined as the token number associated with the token `DIGIT`.

This mechanism leads to clear, easily modified lexical analyzers. However, you should avoid using in the grammar any token names that are reserved or significant in the C language or the

parser. For example, the use of the token names **if** or **while** will probably cause severe difficulties when the lexical analyzer is compiled. The token name **error** is reserved for error handling.

As mentioned above, the token numbers are 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), follow the first appearance of the token name or literal in the declarations section with 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.

The token number for the endmarker must be either 0 or a negative number. This token number cannot be redefined by the user. Hence, all lexical analyzers should be prepared to return 0 or a negative number as a token 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. The **lex** program can easily be used to produce some quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) that do not fit any theoretical framework and whose lexical analyzers must be written in C language code.

How the Parser Works

The **yacc** program generator turns the specification file into a C program that parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and is not discussed here. The parser itself, however, is relatively simple, and your understanding how it works will make the treatment of error recovery and ambiguities easier to understand.

You can consider the parser produced by **yacc** to be a finite state machine with a stack. The parser is capable of reading and remembering the next input token (called the look ahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels. Initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called **shift**, **reduce**, **accept**, and **error**. A move of the parser is done as follows:

1. Based on its current state, the parser determines 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 can 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

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

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has read the right side of a grammar rule, and is ready to replace the right side of the rule with the left side. You may have to consult the lookahead token to decide whether or not to reduce the stack, but usually it is not necessary. The default action, represented by a dot (.), 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:

```
.      reduce 18
```

refers to grammar rule 18, while the action

```
IF      shift 34
```

refers to state 34.

Suppose the rule being reduced is:

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 side (three in this case). To reduce, first remove the top three states from the stack (in general, the number of states removed equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack when the parser recognized x, y, and z. These states no longer serve any useful purpose. The state uncovered is the state the parser was in before it began processing 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. Therefore, this action that seems like a shift is called a **goto** action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto action. 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 parser, removing the states off the stack to go back to the state where the right side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right side of the rule is empty, no states are removed from 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 **yyval** is copied onto the value stack. After the 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 Accept and Error Actions

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been read and that it matches the specification. This action appears only when the lookahead token is the endmarker, and it indicates that the parser has successfully done its job. The error action, on the other hand, marks a place where the parser can no longer continue parsing according to the specification. The input tokens it has read, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing. Error recovery (as opposed to the detection of error) is discussed in “Error Handling” on page 8-31.

An Example: yacc Parsing

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 removed) is shown on the following pages.

```
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
```

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 read, and what is yet to come, in each rule. Suppose the input is:

```
DING DONG DELL
```

Initially, the current state is 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 because a match has been made.

```
sound : DING DONG
```

This rule has two symbols on the right side, so two states, 6 and 3, are removed from the stack, uncovering state 0. Consulting the description of state 0, and looking for a goto on `sound`, the action:

```
sound goto 2
```

is obtained; thus 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 (that now has 0, 2, and 5 on the stack) and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This rule has one symbol on the right side, so one state, 5, is removed, 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 removed, 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 `$end` in the **y.output** file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

We urge you to consider how the parser works when confronted with such incorrect strings as `DING DONG DONG`, `DING DONG`, `DING DONG DELL DELL`. A few minutes spent with this and other simple examples will help you when problems arise in more complicated contexts.

Ambiguity and Conflicts

A set of grammar rules is ambiguous if an input string 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 right association).

The **yacc** program generator detects such ambiguities when it is attempting to build the parser. Consider the problem caused by the following input:

```
expr - expr - expr
```

After the parser reads the second `expr`, the input:

```
expr - expr
```

matches the right side of the grammar rule. The parser could reduce the input by applying this rule; after applying the 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, after the parser reads:

`expr - expr`

it could defer the immediate application of the rule, and continue reading the input until it reads:

`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 either a shift or a reduction; it has no way of determining which to do.

This situation, which allows either a shift or a reduce action is called a ***shift/reduce*** conflict. The parser may also be able to perform two reduce actions. This is called a ***reduce/reduce*** conflict. 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 the choice to make in a given situation is called a disambiguating rule.

The **yacc** program generator invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred, whenever there is a choice, in favor of shifts. Rule 2 gives you crude control over the behavior of the parser in this situation, but you should avoid reduce/reduce conflicts whenever possible.

Conflicts 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 actions must be done before the parser can be sure of the rule 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.

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 without conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting 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 the first of the above groupings of the input.

On the other hand, the ELSE can be shifted, S2 read, and then the right hand portion of:

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get:

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things because there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, and this leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as:

```
IF ( C1 ) IF ( C2 ) S1
```

There can 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. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE
```

```
state 23
```

```
stat : IF ( cond ) stat_      (18)
stat : IF ( cond ) stat_ELSE stat

ELSE  shift 45
      .      reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state and the parser actions. Recall that the underline marks the portion of the grammar rules that have been seen. Thus in the example, in state 23, the parser has seen input corresponding to:

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is `ELSE`, it is possible to shift into state 45. State 45 has, as part of its description, the line:

```
stat : IF ( cond ) stat_ELSE_stat
```

since the `ELSE` will have been shifted in this state. Back in state 23, the alternative action, described by `.` (period), 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 states, there will be at most one reduce action possible for that state. This will be the default command. The

user who encounters unexpected shift/reduce conflicts should look at the verbose output to decide whether the default actions are appropriate. In difficult cases, the user might need to know more about the behavior and construction of the parser than can be covered here.

Precedence

The rules given above are not sufficient for resolving conflicts in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be described by the precedence levels for operators, together with information about expressions to the left or right. 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 form creates a very ambiguous grammar, with many parsing conflicts. In the disambiguating rules, the user specifies the precedence, or binding strength, of all the operators. The user also specifies how the binary operators are associated with each other. The information that these rules provide is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules, and to 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 or binding strength.

Thus:

```
%left '+' '-'  
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than the star and slash, that are also left associative. The keyword `%right` describes right associative operators, and the keyword `%nonassoc` describes operators that may not associate with themselves; like the operator `.LT.` in FORTRAN. The following example is illegal in FORTRAN, and in **yacc**, such an operator would be described with the keyword `%nonassoc`.

```
A .LT. B .LT. C
```

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 different precedences. An example is unary and binary minus (-); unary minus can 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. It is followed by a token name or literal. The `%prec` 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 `%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 have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, 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 nonassociative 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 can 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 is gained. The **y.output** file is very useful in deciding whether the parser is actually doing what is intended.

Error Handling

Error handling is an extremely difficult area because 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. However, continuing to scan causes a problem getting the parser restarted after an error. A general class of algorithms to restart the parser involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow control over this restarting process, **yacc** reserves the token name **error** for error handling. You can use this name in grammar rules. In effect, the name suggests places where errors are expected and recovery might take place. The parser removes states from the 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.

To prevent numerous 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 an error state, no message is given, and the input token is deleted.

As an example, a rule of the form:

```
stat : error
```

causes the parser to attempt to skip over the statement where the syntax error was read. The parser scans ahead, looking for three tokens that legally follow a statement, and it starts processing at the first of these. If the beginnings of statements are not sufficiently distinctive, the parser can make a false start in the middle of a statement, and end up reporting a second error where there is none.

Actions can be used with these special error rules. These actions might attempt to reinitialize tables or reclaim symbol table space.

Error rules such as the above are very general but difficult to control. Rules like the one to follow are easier to control:

```
stat : error ';' ;
```

Here, the parser skips over the statement by skipping to the next statement terminator (;). All tokens after the error and before the (;) cannot be shifted and are discarded. When the (;) is read, this rule is reduced, and any cleanup action associated with it performed.

Another form of error rule arises in interactive applications, where it is 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 continues parsing the input stream. If the reentered line contains an error in the first two tokens, the parser deletes these tokens and gives no message. To prevent this, you can use a mechanism to force the parser to continue parsing. The following statement in an action resets the parser to its normal mode.

```
yyerrorok ;
```

An example looks like this:

```
input : error '\n'
      { yyerrok;
        printf( "Reenter last line: " ); }
input
      { $$ = $4; }
      ;
```

The token read immediately after the **error** symbol is the input token where the error was discovered. Sometimes, this action is inappropriate; for example, an error recovery action might attempt to find the correct place to resume input. In this case, the previous lookahead token must be cleared. The following statement in an action clears the previous lookahead token:

```
yyclearin ;
```

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, although crude, 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.

The yacc Environment

When the user inputs a specification to **yacc**, the output is a file of C programs, called **y.tab.c** on most systems. The function produced by **yacc** is called **yyparse**; it is an integer valued function. When **yyparse** is called, it, in turn, 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.

For the parser to obtain a working program, the user must provide an effective environment. For example, you must define your **main** program to call **yyparse** and **yyerror**.

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. The source of these library programs 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`. 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 is detected; this can give better diagnostics. Since the **main** program is probably supplied by the user (to read arguments, etc.) the **yacc** library is useful only for 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 outputs a verbose description

of its actions, including a list of the input symbols that have been read and what the parser actions are. Depending on the operating environment, you can set this variable by using a debugging system.

Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy-to-change, and clear specifications.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file.

- Use uppercase letters for token names, lowercase letters for nonterminal names. This rule helps you locate errors.
- Put grammar rules and actions on separate lines. This allows you to change either the rule or the action without automatically changing the other.
- Put all rules with the same left side together. Write the left side only once, and begin the following rules with **!** (or symbol).
- Put a statement terminator (;) only after the last rule with a given left side and put it on a separate line. This allows you to add new rules easily.
- Indent rule bodies by two tab stops and action bodies by three tab stops.

The examples in the text of this section follow this style (where space permits). You must decide about these stylistic questions; the goal, however, is to make both the rules and actions easily seen.

Left Recursion

The algorithm used by the **yacc** parser encourages left recursive grammar rules. These rules have 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 is reduced for the first item only, and the second rule is reduced for the second and all succeeding items.

With right recursive rules, such as:

```
seq : item  
     | item seq  
     ;
```

the parser is bigger, and the items are read and reduced, from right to left. However, if right recursive rules are used, an internal stack in the parser is in danger of overflowing if a very long sequence were read. Thus, use left recursion wherever reasonable.

Consider whether a sequence with zero elements has any meaning. If so, consider writing the sequence specification with an *empty* rule:

```
seq : /* empty */  
    | seq item  
    ;
```

The first rule is always reduced exactly once, before the first item is read, and then the second rule is reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is to determine which empty sequence it has read, when it has not read enough to make a determination.

Lexical Tie-ins

Some lexical decisions depend on context. You may want the analyzer to delete blanks, but not within quoted strings. Or, you may want to enter names into a symbol table in declarations, but not in expressions. You can handle these situations by creating a global flag that is examined by the lexical analyzer and set by actions. Suppose a program consists of 0 or more declarations, followed by 0 or more statements.

```
%{
    int dflag;
}%
    . . .   other declarations   . . .

%%

prog    : decls  stats
        ;

decls   : /* empty */
        {          dflag = 1;  }
        | decls declaration
        ;

stats   : /* empty */
        {          dflag = 0;  }
        | stats statement
        ;

    . . .   other rules   . . .
```

The flag **dflag** is 0 when reading statements, and 1 when reading declarations, except for the first token in the first statement. The parser must read this token before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

Handling Reserved Words

Some programming languages permit the user to use words that are normally reserved for label or variable names. This use is allowed provided that such use does not conflict with the legal use of these names in the programming language. This kind of usage is extremely hard to use in the framework of **yacc**. It is difficult to pass information to the lexical analyzer, specifying “this instance of ‘if’ is a keyword, and that instance is a variable.” The best practice is to avoid a multiple use of keywords.

Simulating Error and Accept in Actions

The parsing actions of error and accept are simulated in an action by use of macros **YYACCEPT** and **YYERROR**. The macro **YYACCEPT** causes **yyparse** to return the value 0. The macro **YYERROR** causes the parser to function as if the current input symbol had been a syntax error; **yyerror** is called, and error recovery takes place. These mechanisms are used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules

An action can refer to values returned by actions to the left of the current rule. The mechanism is a dollar sign followed by a digit, but in this case the digit is 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`. This action is only possible when a great deal is known about the data that precedes the symbol `noun` in the input. This mechanism is somewhat unstructured but, at times, it saves a great deal of trouble. This savings is especially true when a few combinations are excluded from an otherwise regular structure.

Supporting Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. The **yacc** program generator 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 is 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** automatically inserts the appropriate union name, so that no unwanted conversions take place. In addition, type checking commands such as **lint** (C) will be far more silent.

Three mechanisms provide for this typing. First, a way of defining the union must be done by the user since other programs, notably the lexical analyzer, must be able to identify 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 **yyval** and **yyval**, to have a type equal to this union. If **yacc** is invoked with the **-d** option, the union declaration is copied onto the **y.tab.h** file. Alternatively, the union can 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 declarations section, using the `%{` and `%}` delimiters.

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`, or `%nonassoc`, the union member name is associated with the tokens listed. Thus, saying:

```
%left <optype> '+' '-'
```

causes any reference to values returned by these two tokens to be tagged with the union member name `optype`.

Another keyword, `%type`, associates union member names with nonterminals. An example of this is:

```
%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 `$0` previously discussed) leaves `yacc` with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between `<` and `>`, immediately after the first `$`. An example of this usage is:

```
rule : aaa { $<intval>$ = 3; } bbb
      { fun( $<intval>2, $<other>0 ); }
;
```

A sample specification is given in a later section. The facilities in this subsection are not triggered until used. In particular, the use of `%type` triggers these mechanisms. When 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 is historically true.

An Example: A Small Desk Calculator

This example gives the complete **yacc** specification for a small desk calculator. The desk calculator has 26 registers, labeled *a* through *z*, and accepts arithmetic expressions made up of the operators `+`, `-`, `*`, `/`, `%` (mod operator), `&` (bitwise and), `|` (bitwise or), and assignment.

If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is 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 probably 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 */
%%      /* beginning of rules section */

list  : /* empty */
      | list stat '\n'
      | list error '\n'
        { yyerrok; }
      ;

stat  : expr
        { printf( "%d\n", $1 ); }
      | LETTER '=' expr
        { regs[$1] = $3; }
      ;

```

```

expr  : '(' expr ')'
        { $$ = $2; }
| expr '+' expr
        { $$ = $1 + $3; }
| expr '-' expr
        { $$ = $1 - $3; }
| expr '*' expr
        { $$ = $1 * $3; }
| expr '/' expr
        { $$ = $1 / $3; }
| expr '%' expr
        { $$ = $1 % $3; }
| expr '&' expr
        { $$ = $1 & $3; }
| expr '|' expr
        { $$ = $1 | $3; }
| '-' expr %prec UMINUS
        { $$ = - $2; }
| LETTER
        { $$ = regs[$1]; }
| number
;

number : DIGIT
        { $$ = $1; base = ($1==0) ? 8 : 10; }
| number DIGIT
        { $$ = base * $1 + $2; }
;

```

```

%%      /* 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 );
}

```

The yacc Input Syntax

This section describes the **yacc** input syntax, as a **yacc** specification. Context dependencies, and the like, are not considered. Ironically, the **yacc** input specification language is most naturally specified as an LR(2) grammar. The difficulty 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 contains an embedded action. As implemented, the lexical analyzer looks ahead. After reading an identifier, the analyzer looks for the next token, skipping blanks, newlines, and comments. If the next token read is a colon, it returns the token `C_IDENTIFIER`. Otherwise, it returns `IDENTIFIER`. Literals (quoted strings) are also returned as `IDENTIFIER`, but never as part of `C_IDENTIFIER`.

```

    /* 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 ';'
        ;
```

An Advanced Example

This section gives an example of a grammar that uses 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:

(x , y)

where x is less than or equal to y. There are 26 interval-valued variables a through z that are also 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 end point values, stored as a 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). This entire strategy depends strongly on the ability to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

Note also the use of `YYERROR` to handle error conditions, such as division by an interval containing 0 and an interval presented in the wrong order. In effect, the error recovery mechanism of **yacc** discards the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates a use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. A scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts (18 shift/reduce, 26 reduce/reduce) when the grammar is run through **yacc**. The problem is seen by looking at the two input lines:

2.5 + (3.5 - 4.)

and

2.5 + (3.5 , 4.)

The 2.5 is 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 the value. 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 is 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 are 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` 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 then error recovery.

Now, consider the desk calculator example, modified to provide floating-point interval arithmetic.

```
%{  
  
#include      <stdio.h>  
#include      <ctype.h>  
  
typedef struct interval {  
    double lo, hi;  
} INTERVAL;  
  
INTERVAL      vmul(), vdiv();  
  
double        atof();  
  
double        dreg[ 26 ];  
INTERVAL      vreg[ 26 ];  
  
%}  
  
%start lines  
  
%union {  
    int          ival;  
    double       dval;  
    INTERVAL     vval;  
}
```

```

%token <ival> DREG      VREG      /* indices into dreg,
                                   ** vreg arrays */
%token <dval> CONST    /* floating-point constant */

%type <dval> dexp      /* expression */

%type <vval> vexp      /* interval expression */

      /* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS          /* precedence for unary minus */

%%

lines :
      | lines line
      ;

line  : dexp '\n'
      { printf( "%15.8f\n", $1 ); }
      | vexp '\n'
      { printf( "(%15.8f, %15.8f )\n",
                  $1.lo, $1.hi ); }
      | DREG '=' dexp '\n'
      { dreg[$1] = $3; }
      | VREG '=' vexp '\n'
      { vreg[$1] = $3; }
      | error '\n'
      { yyerrok; }
      ;

```

```

dexp      : CONST
          | DREG
            { $$ = dreg[$1]; }
          | dexp '+' dexp
            { $$ = $1 + $3; }
          | dexp '-' dexp
            { $$ = $1 - $3; }
          | dexp '*' dexp
            { $$ = $1 * $3; }
          | dexp '/' dexp
            { $$ = $1 / $3; }
          | '-' dexp %prec UMINUS
            { $$ = - $2; }
          | '(' dexp ')'
            { $$ = $2; }
          ;

vexp      : dexp
            { $$ .hi = $$ .lo = $1; }
          | '(' dexp ',' dexp ')'
            {
              $$ .lo = $2;
              $$ .hi = $4;
              if( $$ .lo > $$ .hi ){
                printf("interval out of order\n");
                YYERROR;
              }
            }
          | VREG
            { $$ = vreg[$1]; }

```

```

| vexp '+' vexp
  {
    $$ .hi = $1 .hi + $3 .hi;
    $$ .lo = $1 .lo + $3 .lo;
  }
| dexp '+' vexp
  {
    $$ .hi = $1 + $3 .hi;
    $$ .lo = $1 + $3 .lo;
  }

| vexp '-' vexp
  {
    $$ .hi = $1 .hi - $3 .lo;
    $$ .lo = $1 .lo - $3 .hi;
  }
| dexp '-' vexp
  {
    $$ .hi = $1 - $3 .lo;
    $$ .lo = $1 - $3 .hi;
  }

| vexp '*' vexp
  { $$ = vmul( $1 .lo, $1 .hi, $3 ); }
| dexp '*' vexp
  { $$ = vmul( $1, $1, $3 ); }
| vexp '/' vexp
  {
    if (dcheck($3))
      YYERROR;
    $$ = vdiv( $1 .lo, $1 .hi, $3 );
  }

```

```

| dexp '/' vexp
  {
    if (dcheck($3))
      YYERROR;
    $$ = vdiv($1, $1, $3);
  }
| '-' vexp %prec UMINUS
  {
    $$hi = -$2.lo.;
    $$lo. = -$2.hi;
  }
| '(' vexp ')'
  { $$ = $2; }
;

%%

#define BSZ 50 /* buffer size for fp numbers */

/* lexical analysis */

yylex()
{
  register int c;

  while ((c = getchar()) == ' ')
    ; /* skip over blanks */
  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 /* push back last character read */
        ungetc(c, stdin);
    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
    */
    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;
{
    INTERVAL      hilo();

    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;
{
    INTERVAL      hilo();

    return(hilo(a/v.hi, a/v.lo, b/v.hi, b/v.lo));
}
```

Out-Dated Features

This section mentions synonyms and features supported for historical continuity, but their use is not encouraged.

- Literals can be delimited by double quotation marks (").
- Literals can be more than one character long. If all the characters are alphabetic, numeric, or underscore, the type number of the literal is defined, as if the literal did not have the quotation marks around it. Otherwise, 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 actually be done by the lexical analyzer.
- Most places where percent (%) is legal, backslash (\) can be used. In particular, the double backslash (\\) is the same as %, and the \left is the same as %left.

- 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
```

- Actions also have the form:

```
={ . . . }
```

and the braces can be dropped if the action is a single C language statement.

- C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Chapter 9. The m4 Macro Processor

Introduction

The **m4** 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 the C language and the analogous **define** statement 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 with 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 can also be called with arguments, in which case the arguments are collected and substituted in the right places in the defining text before **m4** rescans the text.

The **m4** macro provides a collection of about twenty built-in macros. In addition, the user can define new macros. Built-in 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.

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 it can be redirected as in the following example:

```
m4 file1 file2 - >outputfile
```

The use of the dash in the above example indicates processing of the standard input, after the files *file1* and *file2* have been processed by **m4**.

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`. The string `name` must be alphanumeric and must begin with a letter (the underscore (`_`) counts as a letter). The string `stuff` is any text, including text that contains balanced parentheses; it can 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.

Macros can be defined in terms of other macros. For example:

Input:

```
define(N, 100)
define(M, N)
x = M;
y = N;
```

Output:

```
x = 100;
y = 100;
```

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; therefore, even if `N` subsequently changes, `M` does not. `M` remains 100 because **m4** expands macro names into their defining text as soon as it possibly can. That means that when the string `N` is seen as the arguments of **define** are being collected, it is immediately replaced by 100. The result is the same as if you had first specified:

```
define(M, 100)
```

If you really want `M` to equal `N`, even as `N` changes, you have two options. 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 always get the value of `N` (because the `M` is replaced by `N` which, in turn, is replaced by 100).

The second option is to use quoting.

Quoting

The more general solution of the previous example is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by single quotation marks (' ') is not expanded immediately, but rather has the quotation marks removed. If you specify:

```
define(N, 100)
define(M, 'N')
```

the quotation marks around the **N** are stripped off as the argument is collected, but they serve their purpose, because **M** is defined as the string **N**, not 100. As a general rule, **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;
```

For another example, consider defining **N** as 100 and then later redefining it as 200:

Input:

```
define(N, 100)
x = N;
define(N, 200)
x = N;
```

Output:

```
x = 100;

m4:e:3 bad macro name
define(100,200)
```

In this case, **N** is evaluated and changed to 100 as soon as it is seen; therefore, **N** is replaced with 100. When the second define statement is reached, **N** no longer exists because it was changed to 100. As the message in the example shows, **m4** reads the second define statement as if you had specified:

```
define(100, 200)
```

This statement is ignored by **m4**, because you can only define values that look like names. To correctly 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 grave (‘) and acute (’) quotation marks 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
```

Two additional built-ins are related to **define**. The **undefine** built-in and the **ifdef** built-in.

The **undefine** built-in removes the definition of a macro or built-in. The following example removes the definition of `N`.

Input:

```
changequote([,])
define(N,100)
define([N], 200)
o = N;
changequote
undefine('N')
o = N;
```

Output:

```
o = 200;
```

```
o = N;
```

The **undefine** built-in can also remove built-ins; however, once removed, these built-ins cannot be restored. The following example removes the **define** built-in:

```
undefine('define')
```

The built-in **ifdef** provides a way to determine if a macro is currently defined. For instance, suppose that either the word `vanilla` or the word `chocolate` is defined according to a particular implementation of a program. To perform operations according to which system you have, you might specify:

```
ifdef('vanilla', 'define(system,1)' )
ifdef('chocolate', 'define(system,2)' )
```

ifdef actually permits three arguments. If the name is undefined, the value of **ifdef** is then the third argument, as in:

```
ifdef('vanilla', on system 1, not on system 1)
```

In this case, the instruction produces the message on system 1 if vanilla is defined on the system, and the message not on system 1 if vanilla is not defined. Putting these examples together produces the following complete example:

Input:

```
define(vanilla, anything)
ifdef('vanilla', 'define(system,1)')
ifdef('chocolate', 'define(system,2)')
x = system;
undefine('system')
ifdef('vanilla', on system 1, not on system 1)
```

Output:

```
x = 1;

on system 1
```

Using Arguments

So far we have discussed the simplest form of macro processing, replacing one string by another (fixed) string. User-defined macros can also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**), any occurrence of $\$n$ is replaced by its definition 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 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(d, (e,f))
```

there are only two arguments; the second is literally `(e,f)`. And of course, a comma or parenthesis can be inserted by quoting it. The following example shows how **define** statements can be used:

Input:

```
define(bump, $1 = $1 + 1)
define(cat, $1$2$3$4$5$6$7$8$9)
define(a, b c)
define(d, (e,f))
bump(x)
cat(x,y,z)
m = a
n = d
```

Output:

```
x = x + 1
xyz
m = b c
n = (e,f)
```

Using Arithmetic Built-Ins

The **m4** macro 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 can group operations. 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 (for example, a number). Quoting usually gives the result you want, and is, therefore, a good habit to get into.

The following example shows the input and output for the preceding examples:

Input:

```
define(N, 100)
define(N1, 'incr(N)')
a = N
b = N1
define('N', 3)
define(M, 'eval(2**N+1)')
x = N
y = M
```

Output:

```
a = 100
b = 101
```

```
x = 3
y = 9
```

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**.

Use the built-in **maketemp** to make unique file names. The specifications of **maketemp** are identical to the system function **mktemp**. A string of X's (XXXXX) in the argument is replaced by the process id of the current process.

Using Conditionals

A built-in called **ifelse** 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** that compares two strings and returns yes, if they are the same, or no, if they are different.

Input:

```
define(compare, 'ifelse($1, $2, yes, no)')
compare('abcd', 'efgh')
compare(aaa, aaa)
```

Output:

```
no
yes
```

The quotation marks prevent a too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

The built-in **ifelse** can actually have any number of arguments, and thus it 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* matches *e*, the result is *f*. Otherwise, the result is *g*. If the final argument is omitted, the result is null. For example:

```
ifelse(a, b, c)
```

if *a* matches *b*, the result is *c*; otherwise, the result is null.

Manipulating Files

You can include a new file in the input at any time by the built-in function **include** :

```
include(filename)
```

The **include** statement 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 value 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*) continues if it cannot access the file.

It is also possible to divert the output of **m4** to temporary files during processing, and output the collected material upon command. The **m4** macro maintains nine of these diversions, numbered 1 through 9. If you specify:

```
divert(n)
```

all subsequent output is put at 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 by appending them to the current diversion. Specifying:

```
undivert
```

brings back all diversions in numeric order. Specifying **undivert** with arguments brings back the selected diversions in the order specified.

Undiverting discards the diverted data. Diverting data into a diversion whose specified number is not between 0 and 9 inclusive also discards the data. The diverted data does not become the value of **undivert**. Furthermore, the diverted data is not rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. During normal processing, this number is 0. If output is redirected, the output of **divnum** is also redirected.

The following example shows the use of **divert** and **undivert**:

Contents of File1:

```
define(N, 100)
define(M, 200)
define(O, 300)
```

Input:

```
include(file1)
divert(1)
z = O;
divert(2)
y = N;
divert(3)
x = M;
divert
undivert
```

Output:

```
z = 300;

y = 100;

x = 200;
```

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

```
ow is the time
```

The built-in did not return the letter *n* because the starting position of the substring was set at 1 rather than 0. 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 doesn't occur. As with `substr`, the origin for strings is 0.

The built-in `translit` performs character transliteration. For example:

```
translit(s, f, t)
```

modifies *s* by replacing any character found in *f* by the corresponding character of *t*. For example:

```
translit(s, aeiou, 12345)
```

replaces the vowels, *aeiou*, with the corresponding digits, *12345*. 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*. For example, the following built-in deletes vowels from *s*.

```
translit(s, aeiou)
```

The following example shows the input and output for operations similar to the preceding examples:

Input:

```
len(abcdef);
len((a,b))
substr('now is the time',1)

define(s1, 'first string')
define(s2, 'string')
define(s3, 'not here')
index(s1,s2);
index(s1,s3);

define(s,'Thisisastringwithvowels.')
define(s2,s)
translit(s, aeiou, 12345);
translit(s2, aeiou);
```

Output:

```
6;
5
ow is the time
```

```
6;
-1;
```

```
Th3s3s1str3ngw3thv4w21s.;
Thssstrngwthvwls.;
```

Cleaning Up Output

A built-in called **dnl** deletes all characters that follow it up to and including the next newline character. Use it to throw away empty lines that otherwise clutter up **m4** output. For example, if you specify:

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline character at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newline characters disappear.

```
define(N, 100)dnl
define(M, 200)dnl
define(L, 300)dnl
```

You can also achieve this with the following statement:

```
divert(-1)
    define( . . . )
    . . .
divert
```

The following example shows the use of the previous examples and the output that they produce:

Input :

```
define(N, 100)dn1
define(M, 200)dn1
define(L, 300)dn1
    o = N;
    p = M;
    q = L;

undefine('N','M','L')dn1

divert(-1)
    define(N, 100)
    define(M, 200)
    define(L, 300)
divert
    o = N;
    p = M;
    q = L;
```

Output :

```
o = 100;
p = 200;
q = 300;
```

```
o = 100;
p = 200;
q = 300;
```

Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus, you can specify:

```
errprint('fatal error')
```

A debugging aid that dumps the current definitions of defined terms is **dumpdef**. If there are no arguments, you get everything; otherwise, you get the ones you name as arguments. Don't forget the single quotation marks.

Chapter 10. Writing Device Drivers

Introduction

This chapter, along with Chapter 11, “Sample Device Drivers” on page 11-1, discusses:

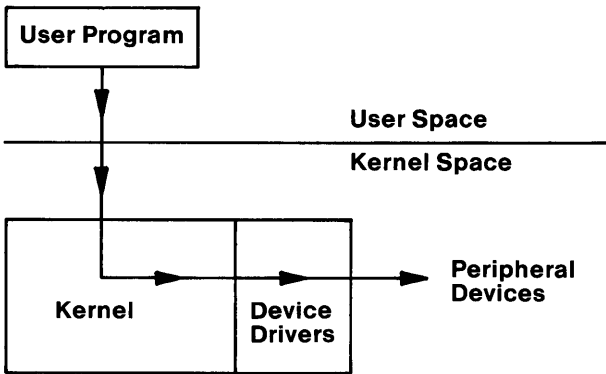
- The role of device drivers in a PC XENIX-based system
- The PC XENIX model of devices in terms of files, tasks, and interrupts
- Special considerations involved in writing a device driver
- Sample device drivers to be used as a guide.
- How to write and install device drivers in a PC XENIX environment

The Role of Device Drivers

The kernel communicates with hardware devices through a set of routines called *device drivers*. For each peripheral device in the system, there must be a device driver. A device driver is part of the kernel. When a process is executing device driver routines the process is executing kernel code.

PC XENIX supports linkable device drivers and installable device drivers. You can link linkable drivers into the kernel by remaking the kernel. Installable drivers can be loaded by the operating system during the system boot. This feature allows you to use much simpler device installation procedures.

Device drivers control the flow of data and the execution of a process between user programs and peripheral devices. The figure below shows the path of an I/O request. The path begins with a system call from a program and ends at the device driver.



User Program Requesting I/O

Device Models Supported by PC XENIX

The PC 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**; most other types are **character devices**. For example, disk drives and tape drives are block devices, while terminals and line printers are character devices. The PC XENIX Operating System presents a uniform interface to the PC XENIX file system, so character and block devices look like files to users and user programs.

Character device drivers communicate directly with the user program. The process begins when a program requests a transfer of data between memory and a device. The operating system transfers control to the appropriate device driver and the user program supplies the parameters for the request.

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 need not be capable of random access.) The two distinguishing factors of block I/O are:

- The size of the data transfer requests from the kernel to the device is always a multiple of the system block size (BSIZE).

This is true 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 in PC XENIX. The device's physical block size may be smaller than BSIZE. If it is, the device driver initiates multiple physical transfers to transfer a single logical block.

- Transfers are never done directly into a user process' memory. The operating system always stages transfers through a pool of BSIZE buffers. PC XENIX satisfies I/O requests directly from the buffers. By using these buffers, PC XENIX performs services such as the blocking, unblocking, and caching of data.

Special Device Files

Each device used by the operating system must be identified by a special device file. In the PC XENIX system, a device appears to function like a file. In this chapter, files that represent devices are referred to as *special device files*. The special device files, by convention, are located in the directory named `/dev` and are maintained by the system administrator.

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 will handle requests for this special file. The *minor number* indicates additional information about a particular device unit that the driver controls (such as a unit identification number).

A program that will use a peripheral device must first open the special device file. You can create a special device file by using the utility program `mknod(C)` (For more information about `mknod(C)`, see *XENIX Commands Reference*). The special device file appears in a directory and has owner and permission fields. The command `ls -l` displays the special device files and shows their major and minor device numbers.

```
crw--w--w- 1 bin          5, 1  Sep 21 09:49 /dev/tty01
brw----- 1 sysinfo     3, 2  Sep 21 09:49 /dev/hd01
```

In this example, the file `/dev/tty01` has a major device number of 5 and a minor device number of 1.

When a user program opens a special device file, PC XENIX recognizes that it is a special device file rather than a data file. PC XENIX uses the major device number to index a table of

device driver entry points. If the file designates a character device, PC XENIX uses the **cdevsw** table. If the special file designates a block device, PC XENIX uses the **bdevsw** table. The definition of these two tables is in the `/usr/sys/conf/c.c` file. The **make** program generates this definition file when it builds the kernel.

These tables may have their contents dynamically altered during the bootstrap sequence if any installable device drivers are loaded. Several blank entries are included in **bdevsw** and **cdevsw** for this purpose. The system configuration file `/lib/sys/config.sys` describes the drivers that are installed at boot time.

PC XENIX then calls the device driver's open entry point through either the **bdevsw** or **cdevsw** table. It passes the minor device number as an argument to the function. The device driver must interpret any unique meaning assigned to the value of the minor device number.

Note: In Version 7 of UNIX, the **dev** parameter passed to the **open()**, **close()**, **read()**, **write()**, and **ioctl()** driver routines included the major and minor device numbers. In System 3 and System 5, 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.

The normal convention for special device files uses meaningful file names and places these files in the `/dev` directory. For example:

```
/dev/tty01
```

The **tty01** indicates the minor device number of the serial device or the device port. Its minor number (not shown) indicates the second port. It is important to note that this is just a convention. The system administrator could assign the same major and minor numbers yet give the file a different name. For example:

```
/dev/magtape
```

or

```
/dev/mt00
```

The system administrator can use either name to identify the same device; both are meaningful. The file name that you use is for your convenience. The PC XENIX kernel uses major and minor numbers to identify devices.

Sample Device Drivers

Chapter 11, “Sample Device Drivers” on page 11-1 discusses sample device driver source code for a line printer, a terminal, a hard disk drive, and a memory-mapped screen. You can use these source code samples as a guide for writing your specific devices drivers.

Each of these samples conforms to the model required for an installable device driver. You can also use them in a kernel that is statically configured to support each of these devices.

Kernel Environment

The following paragraphs briefly discuss a few functional aspects of the PC XENIX operating system:

- Modes of operation
- Context switching
- System-mode stack use
- Task-time processing
- Interrupt-time processing.

It also describes the services the kernel provides to device drivers and the recommended device driver protocol.

Modes of Operation

When a process is executing instructions in a user program, the process is in *user mode*. When the process is executing instructions in the kernel, it is in the *system* or *kernel mode*. When PC XENIX receives an interrupt from an external device, it switches to system mode (if it was in user mode). Then it passes control to the interrupt routine of the appropriate device driver. When the driver is finished, control returns to the kernel and the interrupted process resumes. The interrupted processing is *task-time processing*, and the processing that took place as a result of the interrupt is *interrupt-time processing*.

Although all processes originate as user programs, a given process can run in either system or user mode. In system mode, the process executes 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. PC XENIX provides a high level of protection around processes in user mode to prevent a user program from inadvertently damaging the kernel or other user programs. A process voluntarily enters system mode when it makes a system call. When a process, executing in user mode, encounters an interrupt or a *trap*, the process switches into system mode to handle the interrupt. At this time, the process may lose the CPU (central processing unit) and the kernel may switch control, or *context*, to a different process.

Context Switching

Context switching occurs when the kernel transfers control of the CPU from the currently executing process to a different process.

In user mode, the kernel switches context whenever:

- The time slice of the currently executing process ends.
- A process makes a system call that cannot be completed immediately; for example, a read from a slow input device.
- It receives an interrupt that allows a blocked process to proceed. If the priority of the sleeping process is higher than that of the currently running process, a context switch occurs. This switch occurs when the interrupt handler calls **wakeup()** to a waiting process to indicate the completion of an I/O request.

In system mode, switching contexts is always voluntary. A process voluntarily gives up the processor when a task time handler calls the **sleep()** routine. Interrupts can still arrive and control always passes back to the interrupted process.

System Mode Stack

The user area (`u_` area) is a special area of memory that the kernel uses to manage each process. This area is not directly accessible to the user process (That is, it is not in the process' normal address space). The `u_` area contains process information for the kernel and space for a system mode stack. When any process makes a system call, the kernel preserves the process registers in the `u_` area. The kernel also moves the stack pointer to the beginning of the system mode stack area. When the system call finishes, the kernel restores the registers from the `u_` area, restores the stack pointer to the process' stack, and returns control to the process. Because each process has its own `u_` area, a system running n processes has n user stacks and n system stacks.

The operating system and the task-time portions of the device drivers use a fixed-size system mode stack in the `u_` area. The size of this *per-process* stack is 1024 bytes. It is critical that device driver procedures not create local (frame) buffers of any significant size. For example, consider the following declaration:

```
open()
{
    char buf [512] ;
    char buf2 [512];
```

This declaration can cause problems because the routine requires 1024 bytes of stack space allowing no stack space for the process to use. Furthermore, interrupt service routines make use of whatever system stack space is available at the time of the interrupt. If the interrupt occurs while the currently running process is in user mode, the interrupt service will use the entire `u_` area. However, if the interrupt takes place while the process is in system mode, the interrupt routine will be sharing the `u_` area. For this reason, interrupt service routines must minimize their frame variable declarations, keeping their frame requirements below 512 bytes.

Task-Time Processing

The operating system manages a number of processes, each corresponding to a user program. A process can 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 can be as many as 50 simultaneous processes in system mode, each with its own local variables. This capability requires that all kernel code be re-entrant. However, it otherwise greatly simplifies things because each system process instance has to deal only with servicing the specific system call that the user program requests. The active process' `u_` area is always mapped into the kernel's address space, so when kernel code is executing, the kernel has information about the request and current process.

Often the kernel cannot execute a request immediately. The request may require I/O activity or a pause while waiting for the next action. When a process, in system mode blocks, is awaiting some event, the system scheduler allows some other process to continue.

The operating system passes I/O requests from the user process to the task-time portion of the device driver via system calls. The kernel keeps in the `u_` area some parameters of the request, such as byte count and transfer address. These task-time portions of the driver can reference and perhaps modify the `u_` area cells. This is true because the `u_` area of the currently running process is always mapped in the kernel address space during task-time processing.

Interrupt-Time Processing

Interrupt-time processing refers to the functions the kernel performs when it receives a device interrupt. These functions are called the interrupt service routines. When the kernel receives an interrupt, 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; the interrupted process is probably not the process that initiated the request. Instead, all interrupt-time portions of device driver routines must store information for the task-time portion of the device driver routines in static buffer locations in order 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 be passed via static buffers.

The local (frame) variables of the task portion of the device driver are kept in its system mode stack, which is in the `u_` area. This `u_` area is not mapped into the kernel address space at interrupt time; the `u_` area there belongs to some other process. The correct `u_` area might even be out on the swap device. Therefore, the interrupt service routine must never attempt to store data in the `u_` area or in user memory; and the I/O device itself, via DMA or some other means, must not attempt to transfer directly into the user's memory area.

Character devices typically make use of character lists (clists), small buffers that the system supplies. Block devices use BSIZE buffers from the system buffer pool. The task-time portion of the driver transfers the data from the buffers into the user's memory. It may be important that the transfer take place directly into user's memory. In such cases, it is necessary to lock the user program into physical memory so that it is not swapped.

Typically, the task-time portion of the device driver issues a **sleep()** call when it must wait for the completion of an I/O request. The interrupt service routines service device responses to I/O requests. These routines also notify the task-time portions of the driver, via the **wakeup()** call, that I/O is completed. The interrupt routines then return to the operating system to reschedule the running process and the awakened process. When the task-time portion of the driver awakens, data and status from the driver can be accessed from the static buffers. Critical regions of code that access static variables that can be compromised at interrupt-time can be protected via the **spl5()**, **splbuf()**, and **splcli()** routines.

The **spl5()**, **splbuf()**, and **splcli()** routines raise the interrupt priority of the CPU so that interrupts that might alter the data in the static buffer are locked out until the **splx()** routine is called. This lockout period must be kept as short as possible. Refer to "Kernel Support Routines" on page 10-12 for a more detailed description of these routines.

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. Block type buffered I/O device drivers note the outcome of the data transfer in the buffer headers associated with the buffer used in the transfer. The header for the list of transfers the driver is working on is defined in **/usr/sys/h/iobuf.h**. 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 data and information about the I/O request.

Interrupt Routine Rules

An interrupt routine operates in a more restricted environment than a task-time routine. This is true because 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.

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).

You cannot assume that the `u_` area is mapped into memory during the execution of an interrupt routine. No interrupt routine, nor any routine that is called at interrupt time, can make any reference to user memory, the `u_` area, or non-static memory locations. This means that the task-time portion of the driver must not try to pass addresses of its frame variables, buffers to devices, or interrupt-service routines. Those locations are valid only when that individual user process is executing.

Kernel Support Routines

This section describes the routines that the kernel provides for device drivers to use. All of these routines are entered using *far* calls and must be declared as **extern int far routine()** by the driver.

Note: The memory model that PC XENIX uses to provide for the loading of installable device drivers prohibits the calling of any routines that are not described in this routines section. Therefore, you must restrict every device driver to call only these routines.

in(), out(), inb(), and outb()

This section describes the routines you can use to interface to the registers that access and control a particular device. These registers can reside either in main memory (memory mapped) or in I/O space. There are four routines that provide a portable interface to the registers. The following text describes these routines.

in(port) : word

Purpose: This routine returns the value of the word you specify by the given port or register address.

Parameters: *port* is an integer value that specifies the address of the word.

Result: The value of *word* is returned.

For example, to read the status of a word register at address 20 (hex), you can use the following lines of code:

```
int    val;  
val=in(0x20);
```

inb(port) :byte

Purpose: This routine returns the value of the byte you specify by the given port or register address.

Parameters: *port* is an integer value that specifies the address of the given byte.

Result: The value of *byte* is returned.

out(port, value)

Purpose: This routine sets the word at the specified address to the specified value.

Parameters: *port* is an integer value that specifies the address of the word.

value is the integer value set to the word.

Result: The word at the specified address is set to the specified value.

outb(port, value)

Purpose: This routine sets the byte at the specified address to the specified value.

Parameters: *port* is an inter value that specifies the address of the byte.

value is the byte value set to the byte.

Result: The byte at the specified address is set to the specified value.

splbuf(), splcli(), splx(), spl5(), and spl7()

This section describes the routines used to enable and disable interrupts during task-time processing.

splbuf(): *level*

Purpose: This routine can be called if interrupts should not be acknowledged during task-time processing. It disables all interrupts which would otherwise cause the execution of code that would manipulate data structures associated with block devices, and returns the pre-empted interrupted level. This value is used when restoring interrupts with the **splx()** routine.

Parameters: None

Result: This routine returns an integer value that specifies the interrupt level pre-empted by this routine.

splcli(): *level*

Purpose: This routine can be called if interrupts should not be acknowledged during task-time processing. It disables all interrupts that would otherwise cause the execution of code that would manipulate data structures associated with character devices, and returns the pre-empted interrupted level. This value is used when restoring interrupts with the **splx()** routine.

Parameters: None

Result: This routine returns an integer value that specifies the interrupt level pre-empted by this routine.

splx(oldspl)

Purpose: This routine takes the return value of the **splbuf()** or **splcli()** routines and enables the interrupt levels that were accepted before the call. Calls to **splbuf()** or **splcli()** and **splx()** nest correctly.

Parameters: The integer value **oldspl** specifies the level of interrupts disabled by **splbuf()** or **splcli()**.

For example, to restrict interrupts during critical device driver processing, you can use the following lines of code:

```
int x;  
x = splbuf();  
/*  
** do uninterruptable work  
*/  
splx(x);
```

spl5() : level

Purpose: You can call this routine if interrupts should not be acknowledged during task time processing. This routine disables all character and block device interrupts, and returns the preempted interrupt level. This routine is used when restoring interrupts with the **splx()** routine. Note that this routine is provided for backward compatibility and you should use **splbuf()** and **splcli()** wherever possible.

Parameters: None.

Result: This routine returns an integer value that specifies the interrupt level preempted by this routine. The value is then used in a subsequent call to **splx()**.

spl7(): level

Purpose: You can call this routine to disable all interrupts. You should only use it for extremely short periods, when updating critical data structures that could be accessed by a high priority device.

Parameters: None

Result: The routine returns the value of the pre-empted interrupt level. This is used in a subsequent call to **splx()**.

sleep() and wakeup()

This section describes the routines used to suspend and reawaken requests that cannot be performed immediately. For example, a device driver can receive a write request when the output buffer is full. In this case, the requesting process can suspend itself by calling the **sleep()** routine. When the buffer-full condition ends, the suspended process is awakened in either of two ways: some other process or interrupt routine may awaken the suspended process by calling the **wakeup()** routine, or it can be awakened by a signal from the clock interrupt routine.

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 that identifies the sleeping process. The convention for generating this unique number is to use the address of some data structure the device driver uses. Because no 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 is suspended with **sleep()** at a priority lower than manifest constant PZERO, the clock interrupt routine signal will not awaken the process. Typically, the priority is below PZERO if the condition is likely to disappear almost immediately; otherwise, it is above PZERO.

wakeup(*chan*)

Purpose: This routine wakes up processes that have been suspended by the **sleep()** routine. All the processes that have called **sleep()** with the *chan* specified are awakened. When a process is awakened, the call to **sleep()** returns, and the process should check to be sure that the reason for going to sleep has disappeared.

Parameters: *chan* is a unique number that identifies the sleeping process to be awakened. To generate this number use the address of a data structure the device driver uses. Because no data structures have the same address, uniqueness is guaranteed.

timeout() and delay()

This section describes the routines you can use to schedule a call to a routine at some later time.

timeout(*function*, *arg*, *tim*)

Purpose: This routine allows a function to be called at a scheduled time in the future.

Parameters: *function* is an integer value specifying 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 should elapse before the call.

For example, this routine can be used, along with **sleep()** and **wakeup()**, to provide **busy waiting**. The following code sample illustrates this:

```
#define PERIOD      HZ/10          /* 1/10 second */
#define BUSYPRI    (PZERO -1)    /* somewhat arbitrary */
int stopwait();
int status;

int busywait() /* wait until status is non-zero */
{
    while (status == 0) {
        timeout(stopwait, 0, PERIOD);
        sleep(&status, BUSYPRI);
    }
}

int stopwait()
{
    wakeup(&status);
}
```

Warning: A driver should never loop while waiting for a status change unless the delay involved is shorter than 100 microseconds.

delay(*ticks*)

Purpose: This routine delays **sleep()** and **wakeup()** calls for a number of clock ticks to allow other system functions to continue properly.

Parameters: *ticks* is an integer specifying the number of clock ticks to delay.

Result: After the specified time, the delayed function will continue running.

Warning: This routine should not be called at device initialization (**init**) time. The **delay()** routine uses **timeout()**, **sleep()**, and **wakeup()** mechanisms that depend on the system being fully functional.

dscralloc(), dscrfree(), dscraddr(), and mmudescr()

This section describes the routines used to access memory that is not within kernel data. A descriptor from the Global Descriptor Table (GDT) can be initialized to map the memory area and then used to access the memory.

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.

Parameters: None

Result: This routine returns an unsigned short value that specifies the selector number of the allocated descriptor. If no more descriptors are available, it returns 0 and prints the following message on the system console:

Out of device descriptors, increase gdt size (NGDT) and relink PC
XENIX

Note: It is very important that the driver make sure that the return value is valid (not 0). An attempt to use descriptor 0 can 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 that was 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 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 provided as the argument.

Parameters: *sel* is an unsigned short value that specifies the selector number provided as the argument.

Result: This routine returns the 32-bit physical address of the memory addressed by the selector.

mmudscr(selector, address, limit, access)

Purpose: This routine initializes a descriptor to map a certain area of memory.

Parameters: *selector* is an unsigned short that specifies the selector number of the descriptor allocated by **dscralloc()**.

address is a long that specifies the address of the beginning of the memory area to be mapped.

limit is an unsigned short that specifies the limit of the memory area (its size in bytes - 1).

access is a byte value that specifies an access designation.

For example, the **mmudescr()** routine can be used to map a section of memory 1024 bytes long at address 0xb0000 for reading and writing as follows:

```
mmudescr(sel, 0xB0000, 0x3FF, RW);
```

RW is a define in the **mmu.h** file that specifies read/write access for the driver. RO specifies read-only access to the memory area.

Mapping Memory

The normal sequence of events for a device driver that needs to use a selector to map memory is:

1. Use the **dscralloc()** routine 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 the **mmudescr()** routine to set the descriptor to map the area of memory that the driver needs to access.

ttinit(), ttiocom(), ttstrt(), and ttyflush()

This section describes the routines you can use to initialize the **tty** structures, start **tty** output, and empty the **tty** queue.

ttinit(*tp*)
Purpose: This routine initializes the **tty** structure to specific default values. It should be called immediately after the first **tty** device is opened if you want default settings for your **tty** device.

Parameters: *tp* is a **struct tty*** that points to the **tty** data structure associated with the device being used.

ttiocom(*tp, cmd, addr, mode*)

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.

mode specifies whether the command is a read (FREAD) or write (FWRITE) operation.

ttstrt(*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 the input (FREAD) queue or the output (FWRITE) queue is to be flushed.

copyio()

This section describes the routine used to copy bytes to and from specific locations in the kernel.

copyio(*addr, faddr, cnt, mapping*) : error

Purpose: This routine copies bytes to and from a physical address (buffer address) in the kernel or to and from a long address (user data pointer) in the kernel.

Parameters: *addr* is a long value that specifies the physical kernel address to or from which the data is to be transferred.

faddr is a character value that specifies the segment and offset of the user address to or from which the data is to be transferred.

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 **user.h** and listed below:

U_WUD	Transfer from user data to a kernel data (buffer)
U_RUD	Transfer from kernel data (buffer) to user data
U_WUI	Transfer from user text to a kernel data (buffer)
U_RUI	Transfer from kernel data (buffer) to user text
U_WKD	Transfer from kernel data to file (buffer)
U_RKD	Transfer from file (buffer) to kernel data
U_READ	Copy from <i>addr</i> to <i>faddr</i>
U_WRITE	Copy from <i>faddr</i> to <i>addr</i>

Result: If successful, this routine performs the specified data transfer; otherwise, it returns -1.

For the following `copyio()` example, note that the `ioctl()` interface to a driver actually has two calling sequences:

```
ioctl(fd, cmd, arg)
int fd, cmd, arg;
```

and,

```
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;
```

If `arg` is a pointer to a data structure, you can copy your data in and out using the following `copyio()` example:

```
struct foo dst;
.
. other ioctl code
.
/* copy from arg to dst */
if ( copyio ((caddr_t) &dst, arg,
    sizeof(foo), U_WUD) == -1 )
{
    u.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)

`ftoof(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 to a kernel logical address

`ktop(x)`
Converts a kernel logical address to a physical address

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 3 and System 5; however, adding the following code will provide similar capability:

```
#include ../h/param.h
#include ../h/dir.h
#include ../h/user.h
/*
** iomove - equivalent to the V7 version
**           except we don't provide
**           any of the standard segflg
**           machinations for writing
**           to instruction 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(cp,
                        u.u_base, cnt, U_WUD);
    else
        ret_val = copyio(cp,
                        u.u_base, cnt, U_RUD);
    if(ret_val == -1) {
        u.u_error = EFAULT;
        return;
    }
    u.u_base += cnt;
    u.u_count -= cnt;
    u.u_offset += cnt;
}
```

putchar() and printf()

putchar (c)

Purpose: This routine is used for printing error and system-crash messages when the device driver is used to handle the console. It 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: This 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 only special format characters recognized by this routine are: %s, %c, %d, %ld, %lx, %u, %D, %X, %x, and %o. It also recognizes the newline and carriage return characters.

Note: 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.

Result: None.

panic(), signal(), 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 passed as a parameter on the system console and reboots the system.

Note: This routine should only be called under extreme circumstances.

Parameters: *s* is a *char* pointer addressing a message explaining the reason for the system panic.

Result: None. This routine does not return.

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 giving the process group number.

signum is the signal to be sent.

Result: None.

suser()

Purpose: This routine is used to determine whether the user associated with the currently executing process is the superuser. This can be useful, for example, in determining whether special device operations (such as disk formatting) are allowed.

Parameters: None

Result The routine returns 0 if the current user is not the superuser and 1 if the user is the superuser.

Parameter Passing to Device Drivers

The task-time portion of the device driver has access to the user's `u_` area, because this area is mapped into kernel address space. The kernel routines that handle I/O requests place information describing the request into the `u_` area of the process. The parameters passed into the `u_` area are:

- `u.u_base` Address in user data for read/write data transfers
- `u.u_count` The number of bytes to transfer
- `u.u_offset` The start address within the special file for transfer
- `u.u_segflg` Indicates the direction of the transfer

Refer to the `/usr/sys/h/user.h` file for the values to use for `u.u_segflg`. In addition to the parameters passed into the `u_` area, the kernel I/O routines pass the minor device number as a parameter to the driver when it is called. Thus, 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 to examine the parameters in the `u_` area. Kernel routines that provide these standard interfaces convert 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 that describes the data transfer. Refer to "Device Drivers for Block Devices" on page 10-50 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 "Device Drivers for Character Devices" on page 10-33 for more information on using clists and the character I/O interface routines.

Naming Conventions

There is a naming convention for all driver routines called by the kernel and for some driver variables. Each driver uses a unique 2- to 4-character prefix to identify its routines. For example, a fixed-disk driver might use the prefix `hd`. In the following sections, the prefix `xx` is used to identify driver routines.

Device Drivers for Character Devices

This section describes PC XENIX character device drivers. Character devices conform to the PC XENIX file model; their data consists of a stream of bytes delimited only by the beginning and end of file. The system provides programs with direct access to devices through the special device files described in “Special Device Files” on page 10-3.

Most character device drivers should be designed around the special requirements of terminal devices. There are facilities provided for programming functions on input and output (character erase, line kill, tab functions, etc.), and for setting line options such as speed. Other character-oriented devices, such as line printers, use the same program interface as terminals, but with a different driver.

The character device drivers for slow devices use a data buffering mechanism known as a character list or *clist*. Clists are used for transferring relatively small amounts of data between the driver and the user program. Clists are described in more detail in “Character List and Character Block Architecture” on page 10-44.

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 which device driver to use from the major device number of the device. The driver’s job is to take the requests from the user’s process, 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 are described as follows (*xx* is a mnemonic that refers to the device handle).

xxinit()

Purpose: This routine initializes the device when PC XENIX is first booted. If present, it is called indirectly through the **cinit**, **binits**, or **dinit** table defined in the kernel configuration file (**/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 **oflag** argument that was passed to the open system call.

id is an integer value specifying whether the device is a character device (0) or a block device (1).

xxclose(dev, flag)

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 or clearing device registers.

Parameters: *dev* is an integer that specifies the minor number of the device.

flag is the **oflag** argument passed to the last open system call.

xxstart()

Purpose: If the task-time portion of the driver detects that the device is idle, this routine may be called to start it. This routine 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, it starts the request, usually by sending it a control word. The **xxstart()** routine is not used by device drivers that control **tty** devices.

xxintr(*vec_num*)

Purpose: This routine is called by the kernel when the device issues an interrupt. Because the interrupt typically signals completion of a data transfer, the interrupt routine must determine the appropriate action; perhaps by taking the received character and placing it in the input buffer, or by removing the next character from the output buffer and starting the transmission.

Parameters: *vec_num* is an integer that specifies the interrupt vector number.

xxread(dev)

Purpose: This routine is called when a program makes a read system call. It transfers data to the user's address space. The **cpass()** function is available to transfer one character at a time to the user. 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. It transfers data from the user's address space. A function, **passc**, is available to transfer one character at a time from the user. 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 effect a desired change in the output, such as to perform output character expansion, to perform character output, and to halt or restart character output.

Parameters: *tp* specifies the **tty** structure associated with the device.

cmd specifies the process to be performed. The sample **tty** driver in Chapter 11, "Sample Device Drivers" on page 11-1 documents the list of *cmd* argument values that **xxproc()** can expect.

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 specifies the flags passed on the open system call for the device.

cpass(): c

Purpose: This routine is used to return the next character in a user output request.

Parameters: None.

Result This routine returns *c* which can be either a character or the value -1. A value of -1 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.

Character List Routines

There is a pool of small buffers in the kernel called **character lists** (clists). A clist structure is the head of a linked list queue of characters. **Character blocks** (cblocks) are elements in the linked list. Each cblock can hold a small number of characters. The primary use of the clist buffers is for terminal devices that must interface with the common terminal interface. Clists are also used for buffering low-speed character devices. Refer to "Character List and Character Block Architecture" on page 10-44 for further information about clists.

A driver that uses the clist buffer mechanism must declare a queue header of type clist. If both input and output are buffered, the driver needs two headers. The driver can use six routines to manipulate clist buffers. These routines are described below:

getc(cp): c

Purpose: This routine gets one character from a clist buffer.

Parameters: *cp* specifies the clist buffer from which the character is moved.

Result: This routine returns *c* which can be either the next character in the buffer or the value -1. A value of -1 indicates that there are no characters left in the buffer.

putc(c, cp)

Purpose: This routine puts one character into a clist buffer.

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.

getc(cp): cbp

Purpose: This routine removes one cblock from a clist buffer.

Parameters: *cp* specifies the clist buffer from which the cblocks are moved.

Result: This routine returns *cbp* which can be either a pointer to the next cblock in the buffer or the value -1. A value of -1 indicates that the clist is empty.

putc(cbp, cp)

Purpose: This routine moves one cblock to a clist buffer.

Parameters: *cbp* is a pointer that specifies the cblock to be moved.

cp is a pointer that specifies the clist buffer to which the cblock is moved.

Result: This routine places the specified cblock in the buffer or returns -1 if there is no free space.

getc(): cbp

Purpose: This routine takes a cblock from the cfreelist and returns a pointer to it.

Parameters: None

Result: This routine returns *cbp* which can be either a pointer to the cblock or the value -1. A value of -1 indicates that the clist is empty.

putc(cbp)

Purpose: This routine puts the specified cblock onto the freelist.

Parameters: *cbp* is a pointer to a cblock.

Note: All the cblocks not currently used are kept on a list of free memory blocks. Because there are a limited number of cblocks in the system, each driver must be judicious in determining how many cblocks are used for buffering input and output.

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 buffer contents reaches the high water mark. At that point, the requesting processes are suspended via the **sleep()** routine. When the buffer decreases below the low water mark, the suspended processes are awakened, and can fill the buffer again.

For input buffering, the driver usually buffers data up to some limit. When this limit is reached, data is discarded to make room for more recent data.

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 (which should denote the maximum size of the available buffer).

Result: This routine returns the number of characters actually copied (which is less than or equal to *n*).

putcbp(p, cp, n)

Purpose: This routine copies characters from a buffer to the **clist** given as an 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**.

Result: None

Line Discipline Routines

If you use a serial device as an interactive terminal, it must support various functions, such as character and line erase, echoing, and buffered input. The code needed to perform 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 `Lopen()` routine should be called on the first open of a device.
- The `Lclose()` routine should be called on the last close of the device.
- The `Lread()` and `Lwrite()` routines are called by the drivers read and write routines, to pass characters to and from the calling process.
- The `Linput()` routine is called to buffer incoming characters at interrupt time.
- The `Loutput()` routine is called to get the next block of characters for output at interrupt time.
- The `Lioctl()` routine is used to call specific routines related to the line discipline.
- The `Lmdmint()` routine is unused.

The default line discipline table provided addresses the following routines:

<code>l__open()</code>	→	<code>ttopen()</code>
<code>l__close()</code>	→	<code>ttclose()</code>
<code>l__read()</code>	→	<code>ttread()</code>
<code>l__write()</code>	→	<code>ttwrite()</code>
<code>l__input()</code>	→	<code>ttin()</code>
<code>l__output()</code>	→	<code>ttout()</code>
<code>l__ioctl()</code>	→	<code>ttioctl()</code>
<code>l__mdmint()</code>	→	<code>nulldev()</code>

Interrupt Routines for Character Device Drivers

The device interrupt routine is entered whenever one of its devices raises an interrupt. Generally one driver controls several devices, and all interrupts are vectored through a single function entry point, usually called `xxintr()`. (Note that `xx` is a mnemonic that refers to the device type.) It is the driver's responsibility to determine which device caused the interrupt.

When a device raises an interrupt, it usually provides status information to indicate the reason for the interrupt. The driver interrupt routine decodes this information. If a transfer was completed, the `wakeup()` routine alerts any waiting processes. Then the routine checks to determine if the device is idle, and if so, it looks for more work to start up. In the case of output to a terminal, the interrupt routine looks for more work in the `ccblocks` each time a transfer is completed.

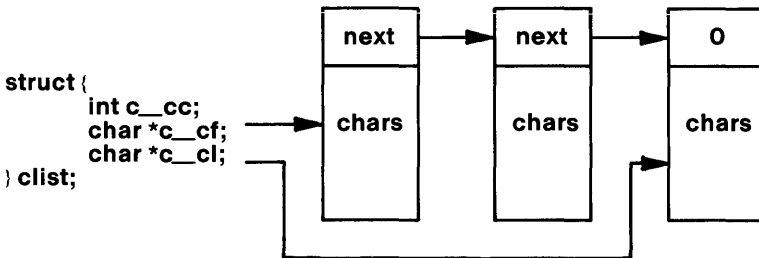
Character List and Character Block Architecture

The character lists (clist) provide a general character buffering mechanism for use by character device drivers. This mechanism is designed for buffering small amounts of data from relatively slow devices, particularly terminals.

The kernel has a pool of character blocks (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 in “Character List Routines” on page 10-38) for putting characters into a clist and removing characters from a clist. These routines should be used by all drivers using clists. Note that these routines are not the same as the Standard I/O Library routines of the same name.

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 cblock in the list, and a pointer to the last cblock. The clist buffers form a single linked list as shown below:



Character List Buffers

There is a protocol defined for use of the clists that prevents a process or driver from consuming all available resources. Two constants for the clist high and low water marks are defined in the file named `tty.h`. A process is allowed to issue write requests until the corresponding clist hits the high water mark. The process is then suspended and I/O performed. When the list reaches the low water mark, the process is awakened. A similar protocol is used for read requests.

Character Device Drivers

There are three character device drivers commonly found on PC XENIX systems: terminal, line printer, and magnetic tape drivers. Line printer and magnetic tape drivers tend to use existing kernel facilities, with little special handling.

Terminal Device Drivers

Terminal device drivers use clists extensively. For each terminal line (each minor device number), the driver declares static clist headers for three clists and two cblocks. These clists are the *raw queue*, the *canonical queue*, and the *output queue*. The cblocks are the transmit control block and the receive control block.

When a process writes data to a terminal device, the task-time portion of the driver puts the data into the output queue, and calls a routine to move the outgoing data into the transmit control block. Then, the interrupt routine transfers the data from the transmit control block to the device.

When a process requests a read of data from the terminal, the situation is slightly more complicated. This complication occurs because PC XENIX gives the requesting process the option of processing characters on input. For example, in normal input the Backspace key means to delete the last character input, and the line kill character means to forget the whole current line. In reading from the terminal, certain special characters (such as backspace) have to be treated in context; that is, they depend upon surrounding characters. To handle this context processing, PC XENIX drivers use two queues and a cblock for incoming data. The two queues are the raw queue and the canonical queue and the cblock is the receive control block.

Data received by the interrupt routine is placed in the receive control block. If the block is full, the interrupt routine calls the **Linput()** function from the linesw table to move a cblock of data to the raw queue. (The raw queue contains data that has not been processed.) Then, the interrupt routine returns, having left the unprocessed data in the raw queue. At task time, the driver determines how much processing to do. The user process has the option of requesting *raw* input. If it does, it receives data directly from the raw queue. (*Cooked* input refers to input after processing for erase, line, kill, delete, and other special line editing functions.)

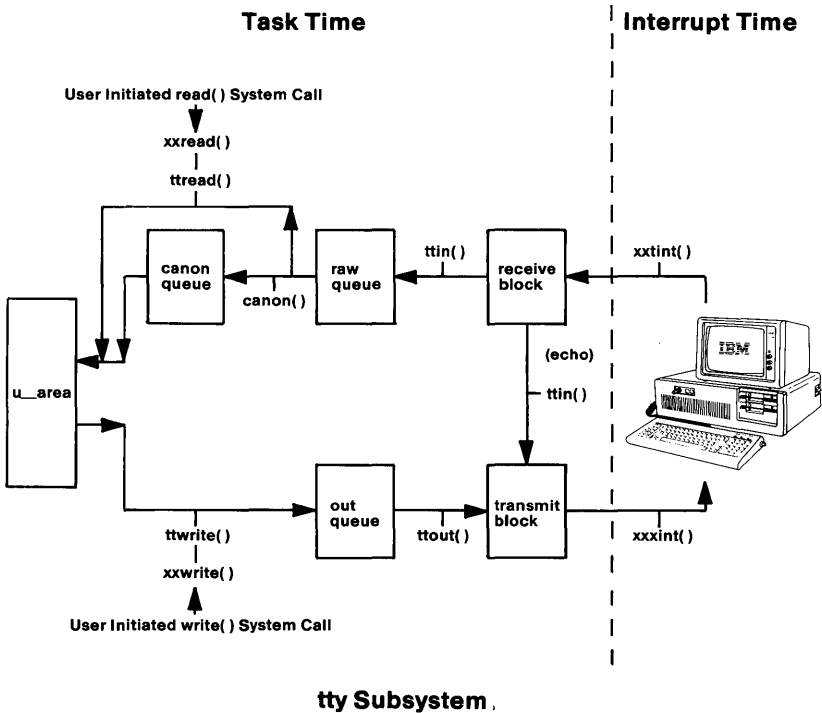
In this case, a task-time routine, **canon()**, is used to transfer data from the raw queue to the canonical queue. This routine performs backspace and line kill functions, according to the options set by the process using the **ioctl(S)** system call. Canonical data refers to the data after processing for erase, line kill, delete, and other special treatment.

In PC XENIX, the specific line discipline normally handles the direct clist processing for **tty** device drivers. 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 cblocks. Each **tty** structure has a cblock for transmitter control (**t_tbuf**) and a cblock for receiver control (**t_rbuf**). The cblock structure has the following format:

```
struct cblock {
    caddr_t    c_ptr;        /* buffer address */
    ushort_t  c_count;     /* character count */
    ushort_t  c_size;      /* buffer size */
} ;
```

At receiver interrupt time, the driver fills a receiver cblock with characters, decrements the character count, and calls the line discipline routine `Linput()`. At transmitter interrupt time, the driver calls `xxproc()` and the line discipline routine `Loutput()` to get a transmitter cblock and then outputs as many characters as possible. Refer to Chapter 11, "Sample Device Drivers" on page 11-1 for code.

The basic flow of data through the system during terminal I/O is shown in the diagram below:



There are two slight complications to the scheme presented in the diagram above. They are output character expansion and input character echo.

Output expansion occurs for a few special characters. In canonical mode, tabs may be expanded into spaces, and the newline character is mapped into carriage return plus line feed. There is a facility for producing escape sequences for uppercase terminals and delay times for certain characters on slow terminals. Note that these examples are simple expansions, or mapping single characters, and so they do not require a second list, as is 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 option, 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 typing at a terminal gets fast echo, regardless of whether his program is in memory and running, or swapped out on disk.

Line Printer Drivers

These are usually relatively slow character-oriented devices. The drivers use the clist mechanism for buffering data. However, a line printer driver is generally simpler than a terminal driver because there is less processing of output characters to do and no input.

Magnetic Tape Drivers

Magnetic tape device drivers differ greatly from terminal and line printer drivers because of the special way magnetic tape devices handle data. Magnetic tape devices generally handle data in the following ways:

- Data is arranged in blocks.
- Data is accessed serially.
- Data is moved in large amounts.
- The device is accessed by only one program at a time.

Therefore, the elaborate kernel buffer management scheme is not applicable to tape drive devices. Furthermore, the clist mechanism is inappropriate because of the large amounts of data involved.

Usually tape drivers provide two interfaces, a character and a block interface. The character interface is used for raw data I/O directly between the device and the address space of the user's process. The block interface makes use of the kernel buffer pool and buffer manipulation routines to store data in transit between device and process. Refer to "Character Interface to Block Devices" on page 10-51 for information on providing the facility for raw I/O.

Device Drivers for Block Devices

Block devices are devices that must be addressed in terms of large blocks of data, rather than individual bytes. Disks and some magnetic tape systems fall into this category. PC XENIX file systems always reside on block devices but block devices do not have to be accessed in large data block fashion.

Unlike the case with character devices, a block I/O transfer request is not a private transaction between a driver and a user process. The kernel provides a comprehensive buffer management scheme which is used by block device drivers.

The kernel maintains a pool of buffers. The kernel also keeps track of the data in the buffers and whether or not the block has modified data that needs to be written to disk. When a user process issues a transfer request to a block device, the kernel buffer routines check the buffer pool to see if the data is already in memory. If not, a request is passed to the driver to get the data. All the driver ever sees are fixed-size requests (BSIZE bytes long) coming in from one source. This is true regardless of the size of the I/O request from the user's process. Large requests are broken down into BSIZE blocks, and handled individually, because some may be in memory, and some may not.

When a process issues a read request, this request generally translates into one or more disk blocks. The kernel checks to see which of these is already in memory and causes the driver to get the rest. The kernel then copies the data from each filled buffer into the memory of the process.

In the case of a write request, the kernel copies the data from the user process' memory into the buffer pool. If there are insufficient free buffers, the kernel will have 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.

Character Interface to Block Devices

Sometimes block device drivers provide a character I/O interface as well as an interface for block I/O. When the device provides a character interface, you can create a separate special device file 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 *XENIX Commands Reference*. This utility creates a character special device file that has the same major and minor number as the block special device file. To implement character I/O, the block device driver must provide the routines **xxread**() and **xxwrite**() described below.

When a block device is accessed through a character interface, data transfer takes place directly between the device and the memory space of the process. There is no intermediate buffering in the kernel buffer pool or in the clists. 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. This type of data transfer is referred to as physical (or raw) I/O.

Raw I/O 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. This is true because the device can be accessed sequentially, and large transfers can be made. There is also less copying of data between buffers than is done in the block interface. Therefore, disk backup programs, or utilities that copy entire volumes, typically operate through character interface.

However, because the driver must be able to locate the data, this extra efficiency requires that the process be locked in memory during the transfer. The routine **physio**() called by the **xxread**() and **xxwrite**() driver routines handles the locking of the process in memory for the duration of the data transfer.

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 **param.h** file. The 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:

<i>b_dev</i>	The major and minor numbers of the device
<i>b_bcount</i>	The number of bytes to transfer
<i>b_paddr</i>	The physical address of the buffer
<i>b_blkno</i>	The block number on the device
<i>b_error</i>	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 doubly 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 sorted by 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. If the transfer is completed successfully, the driver returns a success code to the kernel. The kernel buffer routines are responsible for correlating the completion of an individual buffer transfer with the particular process that requested the transfer.

The interface between the kernel and the block device driver consists of the routines described in the following paragraphs.

xxinit()

Purpose: This routine is called to initialize the device when PC XENIX is first booted. If present, it is called indirectly through the *binitsw* table defined in the kernel configuration file (*/usr/sys/conf/c.c*).

xxopen(dev, flag, id)

Purpose: This routine is called each time the device is opened. This routine initializes the device and performs any error or protection checking.

Parameters: *dev* is an integer that specifies the minor device number.

flag is the argument that was passed to the open system call.

id is an integer value specifying whether the device is a character device (0) or a block device (1).



xxclose(dev, flag)

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 ejecting media.

Parameters: *dev* specifies the minor device number of the device being closed.

flag is the **oflag** argument that the open system call passes.

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 before it inserts 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 the task-time and the interrupt-time portions of the driver. It checks whether the device is ready to accept another transfer request, and if so, it starts the device, usually by sending it a control word.

xxintr(*vec_num*)

Purpose: This routine is called whenever the device issues an interrupt. Depending on the meaning of the interrupt, it marks the current request as complete, starts the next request, continues the current request, or retries 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.

Parameters: *vec_num* is an integer that specifies the interrupt vector number.

xxread(*dev*)

Purpose: The only action taken by this routine is to call the **physio()** routine 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 want 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 the **physio()** routine with appropriate arguments.

Parameters: *dev* specifies the minor 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 fixed 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.

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.

mode specifies the flags that were set on the **open()** system call for the specified device.

physio(*strategy, 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.

Parameters: *strategy* is a pointer to the disk strategy routine for the block device.

bp is a pointer to the buffer header describing the request to be filled.

dev is an integer specifying the minor device number.

flag specifies the I/O operation to be performed.

brelse(*bp*)

Purpose: This routine is used to release 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 *** which 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, msg)

Purpose: This routine prints an error message on the system console together with some device specific information acquired from the parameters passed 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=%x0,
      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.

msg is a pointer to a string identifying the device.

Result: None.

disksort(*disktab*, *bp*)

Purpose: This routine is called to add 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 *disktab* parameter is the head of the request queue, and the *bp* parameter addresses the buf structure containing the request. The queue of requests is sorted in ascending order by the **disksort()** routine, in an attempt to reduce disk head movement.

Parameters: *disktab* is the address of a **struct iobuf** which is declared within the driver to form the head of the I/O request queue.

bp is a **struct buf *** which points to the I/O request to be added to the queue.

Result: This routine does not return a result.

For an example, see Chapter 11, "Sample Device Drivers" on page 11-1.

getablk(): *bp*

Purpose: This routine is used to acquire 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 **brelease()** or **iodone()**.

Parameters: None.

Result: This routine returns *bp* which is a **struct buf *** that addresses the allocated buffer.

iodone(bp)

Purpose: This routine will signal completion of an I/O operation involving the buffer addressed by *bp*. This routine is called when the driver wants to signal either successful or erroneous completion of an I/O operation. It differs from the **brelease()** 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 **brelease()**.

Parameters: *bp* is a **struct buf *** which addresses the buffer.

Result: None.

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 should not be called within a device driver since it may call the **sleep()** routine.

Parameters: *bp* is 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.

Rules for Writing Installable Device Drivers

PC XENIX makes provision for *installable device drivers*. This facility enables the creation of object code versions of device drivers which can be loaded during system boot . Once loaded, the driver is installed into the kernel image in order to provide access to the device which it is designed to control. The user of the system perceives no changes other than during the initial bootstrap phase. To the programmer, the creation of an installable device driver requires only a small additional effort over that required for ordinary drivers which are linked into the kernel image during a configuration process. The format of an installable driver is such that it can also be configured into the kernel during a configuration process. There is nothing inherent to an installable driver which makes it incompatible with this process.

Note: Due to the memory model to which the kernel conforms, **installable device drivers must call only those kernel routines which are specified within this section.** An attempt to call any other kernel routine will probably cause the kernel to crash.

In order to create an installable device driver, the programmer must add additional data structures to an existing device driver, or must include these data structures when developing a new device driver. Examples of these data structures are given in the example drivers in the following chapter. In summary, the additional data structures are:

`struct iddsw` This data structure contains control information for the program which performs the process of installing device drivers during the bootstrap. The fields in this structure are described in detail below.

`struct bdevsw` This data structure is included if the device driver is a block device driver. It is an identical copy of the structure that would be generated by the configuration program if the kernel were to be configured statically to include the device driver. The fields in this structure are described below.

`struct cdevsw` This data structure is included if the device driver is a character device driver. It is an identical copy of the structure that would be generated by the configuration program if the kernel were to be configured statically to include the device driver. The fields in this structure are described below.

Note that the precise details of the function and purpose of the device driver routines and data structures mentioned in the subsequent description are described elsewhere in the documentation. This section restricts itself to a description of the functions unique to the installable device driver.

The iddsw Structure

This structure has the following format (see the kernel header file **idd.h**):

```
struct iddsw{
    ushort          idd_tag;
    ushort          idd_vers;
    ushort          idd_type;
    ushort          idd_mask;
    struct bdevsw   *idd_bdevsw;
    struct cdevsw   *idd_cdevsw;
    struct linesw   *idd_linesw;
    int             (*idd_init)();
    int             (*idd_intr)();
    int             idd_bdev;
    int             idd_cdev;
    int             idd_line;
    int             idd_livec[IDD_NVEC];
    int             idd_cmd;
    char            idd_name[IDD_LNAME];
};
```

The one writing the device driver is required to fill in the following fields: *idd_tag*, *idd_vers*, *idd_type*, *idd_mask*, *idd_bdevsw*, *idd_cdevsw*, *idd_intr*, and *idd_name*. Note that the *idd_name* field contains a character string denoting the device's mnemonic name. This is explained below. Other fields use this string in the construction of names for device driver entry points. These are denoted by *xxroutine()* below.

These fields are completed as follows:

idd_tag Always contains the value **IDD_IDD**.

idd_vers Always contains the value **IDD_VERS**.

`idd_type` This field is composed of an inclusive OR of the following values: `IDD_BDEV` (if the driver is a block device driver) and `IDD_CDEV` (if the driver is a character device driver). Note that drivers that function as both block and character device drivers (particularly disk drivers) will OR these two values together. Other drivers will only use one of these identifiers.

`idd_mask` This field is used to identify which driver entry points are present within the device driver module. The field is composed of an inclusive OR of the following possible values:

`IDD_CLEAN` Not presently used.

`IDD_INIT` Set if the device driver has an initialization entry point which should be called after system bootstrap. The name of this entry point must be `xxinit()`.

`IDD_POWER` Not presently used.

`IDD_OPEN` Set if the device driver has a device open routine. The name of this entry point must be `xxopen()`.

`IDD_CLOSE` Set if the device driver has a device close routine. The name of this entry point must be `xxclose()`.

`IDD_READ` Set if the device driver has a device read routine. The name of this entry point must be `xxread()`.

`IDD_WRITE` Set if the device driver has a device write routine. The name of this entry point must be `xxwrite()`.

`IDD_IOCTL` Set if the device driver has a device control routine. The name of this entry point must be `xxioctl()`. This entry point is only applicable to character device drivers.

IDDD_TTY	Set if the device driver requires a tty structure.
idd_bdevsw	This field must be initialized with a pointer to the struct bdevsw included within the driver module or (char*)0 if there is no such structure.
idd_cdevsw	This field must be initialized with a pointer to the struct cdevsw included within the driver module or (char*)0 if there is no such structure.
idd_init	This field must be initialized with a pointer to the device initialization routine xxinit() or (char*)0 if the device has no entry in the dinit table.
idd_intr	This field must be initialized with a pointer to the device interrupt handler routine (xxintr() or (char*)0 if the device has no interrupt handler.
idd_name	This field is initialized with the character string denoting the name of the device. This character string must be that which is used in naming the entry points (that is, the <i>xx</i> of the above discussion) and must be the same as the character string used to denote the device when it is installed on the system using the config.sys file.

The *idd_linesw* and *idd_line* fields are not presently supported and should be filled with (char*)0 and -1, respectively.

The *idd_bdev*, *idd_cdev*, *idd_ivec*, and *idd_cmd* fields are filled in by the bootstrap loader when the device driver is installed. These fields should be initialized to -1.

When declaring this structure, you should be careful to ensure that all forward references to device driver entry points, for example, have been correctly declared and that the name chosen for the *struct iddsw* is a global name. The names chosen for the *bdevsw* and *cdevsw* structures can be declared as static, since they are accessed via the pointers in the *struct iddsw*.

The bdevsw Structure

This structure has the following format (see the kernel header file **conf.h**):

```
struct bdevsw {
    int      (*d_open)();
    int      (*d_close)();
    int      (*d_strategy)();
    struct iobuf *d_tab;
};
```

The fields are initialized as follows:

<code>d_open</code>	The address of the device open routine, or nulldev() , if there is none.
<code>d_close</code>	The address of the device close routine, or nulldev() , if there is none.
<code>d_strategy</code>	The address of the device strategy routine.
<code>d_tab</code>	The address of the buffer queue anchor.

The cdevsw Structure

This structure has the following format (see the kernel header file **conf.h**):

```
struct cdevsw {
    int      (*d_open)();
    int      (*d_close)();
    int      (*d_read)();
    int      (*d_write)();
    int      (*d_ioctl)();
    struct tty *d_ttys;
};
```

The fields are initialized as follows:

<code>d_open</code>	The address of the device open routine, or nulldev() , if there is none.
<code>d_close</code>	The address of the device close routine, or nulldev() , if there is none.
<code>d_read</code>	The address of the device read routine, or nodev() if this is an illegal operation for this device.
<code>d_write</code>	The address of the device write routine, or nodev() , if this an illegal operation for this device.
<code>d_ioctl</code>	The address of the device control routine, or nulldev() , if there is none.
<code>d_ttys</code>	The address of the struct tty associated with this device, or (char*)0 , if there is none.

Configuring the System

After you write the device driver for your system, you can configure it into your system in two ways:

- You can remake the kernel and link your device driver with the new image. This produces a linkable device driver.
- You can prepare the device driver object module to be an installable image that the boot program can dynamically link to the kernel at bootstrap time. This produces an installable device driver.

The following two sections explain the procedure for configuring your device driver as a linkable device driver or as an installable device driver.

Building Linkable Device Drivers

To build a linkable device driver follow these steps:

1. Move your device driver source code to the `/usr/sys/io` directory.
2. Add the name of the device driver file objects (for example, `driver.o`) to the `OBJS` list in the `makefile` in the `/usr/sys/io` directory.
3. Type **make** in the `/usr/sys/io` directory to create the device driver object files. The `makefile` passes the `-NT` flag to the compiler therefore forcing all objects to be loaded in the same segment. This allows device drivers to make near calls instead of far calls to support routines, if necessary. Using near calls to frequently called routines improves the performance of the device drivers.

After the device driver object files are compiled, the `makefile` adds the object files to the `/usr/sys/io/lib_io` library. The `make` program can handle drivers written in either C or assembly language.

4. Modify the `master` file in the `/usr/sys/conf` directory to include information about the new drivers. The `master` file describes all devices that could ever be present on a system. To make the changes, refer to the `master(F)` command in *XENIX System Reference*. An example of the possible entries in the `master` file for three new device drivers is shown below:

```
* The following devices are those that can be specified
* in the system description file. The name specified must
* agree with the name shown.
*
*name vsiz  msk   typ hndlr na bmaj cmaj #  na vec1 vec2 vec3 vec4
* 1      2      3    4     5   6   7   8   9  10  11  12  13  14
hd      1      0027 014  hd   0   3   3   1   0   36   0   0   0
td      2      0137 004  sa   0   0   5   1   0   3    4   0   0
lp      1      0022 004  pa   0   1   6   1   0   5    0   0   0
```

Note that no two devices can have the same block major numbers (`bmaj`) or the same character major numbers (`cmaj`).

5. Modify the `xenixconf` file in the `/usr/sys/conf` directory to include information about the new drivers. The `xenixconf` file specifies which devices are part of the particular machine for which the kernel is being built. The format of an entry in the `xenixconf` is very straightforward. It consists of the name of the device, followed by the number of devices present

in the configuration. Add one line for each new device driver. For example, the entries for the device drivers shown in Step 4 might be:

```
hd 1
td 2
lp 1
```

6. Build a version of the kernel containing your new drivers by typing `make` in the `/usr/sys/conf` directory.
7. You can now copy the new version of the kernel to the root (`/`) and enter it as the executable image to the boot program.

Building Installable Device Drivers

In order to prepare a device driver for use as an installable device, you must perform a few steps. This involves:

- Modifying the object code format of the driver module, so that it is acceptable to the boot loader
- Preparing the necessary system management files.

This section contains a description of how you, a programmer, should prepare a device driver for use as an installable driver. The “Installing Device Drivers” section in *XENIX System Administration* contains a description of the system management issues.

Informal conventions are available for installable device drivers:

- The file name suffix `.x` is used for modules of a certain format
- The files associated with installable device drivers are placed in the `/lib/sys` directory.

If you are familiar with a UNIX system, you will quickly see how to modify the examples given here to meet your own requirements. If you are not familiar with developing device drivers for UNIX, please follow the conventions established here.

Compiling a driver requires you to use the PC XENIX C language compiler and assembler to create an object module. The `cc` command line must include the following switches:

- K
Disable stack probes.
- DM_Kernel
Required for conditional code in standard header files.
- NT `io_text`
Names the text segment for the driver code as `io_text`.
- M2em
Enable 286 instructions, `near` and `far` keywords.
Build as a middle model program.
(These are required to conform with the kernel program model.)

The **.o** object module the compiler creates is in OMF86 format. Use the **xcvt** utility to convert the module to relocatable **x.out** format. Then link the driver module with a standard module **KMseg.x**. The driver is now ready for use as an installable device driver. The example below shows how to carry out these steps for a driver in the **fd.c** source file.

```
cc -K -DM_KERNEL -NT io_text -M2em -c fd.c
/etc/xcvt fd.o; mv a.o fd.x
/etc/xld -r -o /lib/sys/fd.x /lib/sys/KMseg.x fd.x
```

Note these conventions in the example:

- The **.x** suffix for relocatable **x.out** modules
- The storing of the final output module in the **/lib/sys** directory
- The use of the **/lib/sys/KMseg.x** module.

You can use the **xld** command to link a driver prepared as more than one source module. To do this, compile each source file. Then, process each file with the **xcvt** utility. Finally, link each object file together into a single installable driver module with the **xld** command. The following example shows this for a driver composed of three source files **d1.c**, **d2.c**, and **d3.c**.

```
for i in 1 2 3
do
    cc -K -DM_KERNEL -NT io_text -M2em -O -c d$i.c
    /etc/xcvt d$i.o
    mv a.o d$i.x
done
/etc/xld -r -o d.x /lib/sys/KMseg.x d1.x d2.x d3.x
```

You can see a practical example of preparing installable device drivers by using the makefile **/usr/sys/io/idd.mk** to prepare installable drivers from the normal linkable driver modules in the standard kernel libraries. This makefile compiles and links the normal drivers with modules containing the data structures required to describe them as installable device drivers.

Determining Interrupt Vector Numbers

When you modify the `/usr/sys/conf/master` file to contain information about the device drivers that you have written, you need to specify the interrupt vector number each device uses to interrupt the kernel. The `config` utility uses the `master` file to generate the C language file `/usr/sys/conf/c.c`. The `c.c` file will contain the `vecintsw` table that has information about the vector level that each device uses to interrupt the kernel.

The i8259 interrupt controller can have up to eight 8259 interrupt controllers cascaded on it, providing up to 64 vectored interrupts.

Interrupt levels 0 through 7, on the master 8259 interrupt controller, map to entries 0 through 7 in the `vecintsw` table. Thus, if a device interrupts on the master interrupt controller, the vector number to specify in the `master` file for the device is simply the vector number it uses on the master.

Entries 8 through 72 in the `vecintsw` table define interrupts from slave controllers. To determine the proper index for the `vecintsw` table for any interrupt coming in on a slave 8259, use the following formula:

```
vector = ((slave_number + 1) * 8) + slave_interrupt_level
```

If the slave interrupts the master on interrupt vector 2, then:

```
vector = 24 + slave_interrupt_level
```

For example, if the diskette controller interrupts on level 6 of a slave 8259 interrupt controller that interrupts the master on level 2, then its index in the **vecintsw** table is:

```
vector = 24 + 6
```

```
vector = 30 = 36(octal)
```

Note: The vector entries in the **master** file are designated in octal.

Sharing Interrupt Vectors

I/O devices can only share interrupt vectors if there is a way to poll each device that is using the shared vector. This poll must determine which device has posted an interrupt. The configuration utility **config** allows for the user to specify that two devices share an interrupt level. Refer to the **master(F)** command in *XENIX System Reference* and the **config(CP)** command in *XENIX Commands Reference* for further information on the **config** utility.

If there are two devices `aa` and `bb` that share interrupt level 3, the code in the `c.c` file generated by `config` should be as follows:

```
vector3(level)
int level;
{
    aaintr(level);
    bbintr(level);
}
```

```
int (*vecintsw[])( ) =
{
    clock,
    consintr,
    novect,
    vector3,
    novect,
    etc
    .
    .
    .
}
```

The interrupt routines `aaintr()` and `bbintr()` should have the following format:

```
xxintr(level)
int level;
{
    IF NOT MY INTERRUPT
        return;

    NORMAL INTERRUPT PROCESSING
}
```

Warnings

The following warnings will help you avoid problems when writing a device driver:

- Do not defer interrupts with **spl5()** 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 call **spl5()** at interrupt time.
- Make interrupt-time processing as short as possible.
- Protect buffer and clist processing with **spl5()** calls.
- Avoid busy waiting whenever possible.
- Never use floating-point arithmetic operations in device driver code.
- If any assembler language device driver sets the direction flag (using **std**), it must clear the flag (using **cld**) before returning.
- Keep the local (stack) data requirements for your driver very small.

Chapter 11. Sample Device Drivers

Introduction

This chapter provides sample device driver code for line printers, terminals, and fixed-disk drives. After each segment of code, some general comments are provided that describe the routines and explain key lines in the program.

Note: The example device drivers that follow do not implement interrupt level sharing.

Sample Device Driver for Line Printer

```
1  /*
2  ** lp- prototype line printer driver
3  */
4  #include "../h/param.h"
5  #include "../h/dir.h"
6  #include "../h/a.out.h"
7  #include "../h/user.h"
8  #include "../h/file.h"
9  #include "../h/tty.h"
10 #include "../h/conf.h"
11
12 #define LPPRI    PZERO+5
13 #define LOWAT    50
14 #define HIWAT    150
15
16 /* register definitions */
17
18 #define RBASE    0x00    /* base addr of rgtrs */
19 #define RDATA    (RBASE + 0) /* put char here */
20 #define RSTATUS (RBASE + 1) /* nonzero = busy */
21 #define RCONTRL (RBASE + 2) /* write ctrl here */
22
23 /* control definitions */
24 #define CINIT    0x01    /* initialize interface */
25 #define CIENABL 0x02    /* +Interrupt enable */
26 #define CRESET   0x04
```

Description of Device Driver for Line Printer

This device driver is for a single parallel interface to a printer. The program 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 is stopped. Because the priority is greater than PZERO, the process can be awakened by a signal.
- 13: LOWAT is the minimum number of characters in the buffer. When there are fewer than LOWAT characters in the buffer, a sleeping process can be restarted.
- 14: HIWAT is the maximum number of characters in the buffer. If a process fills the buffer to HIWAT, the **sleep()** routine suspends the process.
- 18-21: The device registers in this interface occupy a contiguous block of addresses, starting at RBASE and running through RBASE + 2. The data to be printed is placed in the RDATA register one character at a time. Printer status can be read from the RSTATUS register, and the interface can be configured by writing into the RCONTRL register.

```

27
28 /* flags definitions */
29 #define FIRST 0x01
30 #define ASLEEP 0x02
31 #define ACTIVE 0x04
32
33 struct clist lp_queue;
34 int lp_flags = 0;
35
36 int lpopen(), lpclose(), lpwrite(), lpintr();
37
38 static struct cdevsw lpcdev = {
39     lpopen, lpclose, nodev, lpwrite, nulldev, NUL
40 };
41
42 struct iddsw lpiddsw = {
43     IDD_IDD, IDD_VERS, /* Tag/version number
44     IDD_CDEV, /* Character device */
45     /* Entry points */
46     IDD_OPEN|IDD_CLOSE|IDD_WRITE,
47     NULL, /* bdevsw entry */
48     &lpcdev, /* cdevsw entry */
49     NULL, NULL /* linesw entry/init routines
50     lpintr, /* Interrupt handler */
51     -1,-1,-1, /* Filled from config.sys */
52     { -1 },
53     -1,
54     "lp" /* Name */
55 };
56

```

-
- 29-31:** The flags defined in these lines are kept in the variable *lp_flags*. The **FIRST** flag is set if the interface has been initialized. The **ASLEEP** flag is set if a process is asleep and waiting for the buffer to decrease below **LOWAT**. The **ACTIVE** flag 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 flags mentioned above are kept.
- 38-55:** *lpcdev* is the **struct cdevsw** which will be copied into the kernel **cdevsw** when this device driver is installed. The entry points are given exactly as if the kernel had been statically configured. *lpiddsw* is the **struct iddsw** which provides all the necessary information for this driver to be dynamically installed.

```

57  lpopen(dev)
58  int dev;
59  {
60      if ((lp_flags & FIRST) == 0) {
61          lp_flags != FIRST;
62          outb(RCONTRL, CRESET);
63      }
64      outb(RCONTRL, CIENABL);
65  }
66
67  lpclose(dev)
68  int dev;
69  {
70  }
71  lpwrite(dev)
72  int dev;
73  {
74      register int c;
75      int x;
76
77      while ((c = cpass()) >= 0) {
78          x = splcli();
79          while (lp_queue.c_cc > HIWAT) {
80              lpstart();
81              lp_flags != ASLEEP;
82              sleep(&lp_queue, LPPRI);
83          }
84          splx(x);
85          putc(c, &lp_queue);
86      }
87      x = splcli();
88      lpstart();
89      splx(x);
90  }
91
92  lpstart()
93  {
94      if (lp_flags & ACTIVE)
95          return; /* interrupt chain is
96                  ** keeping printer goi
97
98      lp_flags != ACTIVE;
99      lpintr(0);
100 }

```

lpopen() - lines 57 to 65

The **lpopen()** routine is called when a process makes an **open()** system call on the special file that represents this driver. Its single argument, *dev*, represents the minor number of the device. Because this driver supports only one device, the minor number is ignored.

- 60-62:** If this is the first time (since PC XENIX was booted) that the device has been touched (or contacted), the interface is initialized by setting the CRESET bit in the control register.
- 64:** Interrupts from this device are enabled by setting the CIENABL bit in the control register.

lpclose() - lines 67 to 70

The **lpclose()** routine is called on the last close of the device, that is, when the current **close()** system call results in zero processes referencing the device. No action is taken.

lpwrite() - lines 71 to 90

The **lpwrite()** routine is called to move the data from the user process to the output buffer. Code is defined as follows:

- 77:** While there are still characters to be transferred, do what follows.
- 78-86:** Raise the processor priority so the interrupt routine cannot 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.
- 87-88:** Make sure the printer is running by locking out interrupts and calling **lpstart()**.

lpstart() - lines 92 to 98

The **lpstart()** routine ensures that the printer is running. This routine is called twice from **lpwrite()**, and it avoids duplicate code. Code is defined as follows:

94-97: If the printer is running, return. Otherwise, turn on the ACTIVE flag, and call **lpintr()** to start the transfer of characters.

```
101  lpintr(vec)
102  int vec;
103  {
104      int tmp;
105
106      if ((lp_flags & ACTIVE) == 0)
107          return; /* ignore spurious interrupt */
108
109      /* pass chars until busy */
110      while (inb(RSTATUS) == 0 &&
111             (tmp = getc(&lp_queue)) >= 0)
112          outb(RDATA, tmp);
113
114      /* wakeup the writer if necessary */
115      if (lp_queue.c_cc < LOWAT &&
116          lp_flags & ASLEEP) {
117          lp_flags &= ~ASLEEP;
118          wakeup(&lp_queue);
119      }
120
121      /* wakeup writer if waiting for drain */
122      if (lp_queue.c_cc <= 0)
123          lp_flags &= ~ACTIVE;
124  }
```

lpintr() - lines 102 to 123

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:

- 107-108:** If **lpintr()** is called unexpectedly or if the driver does not have anything to do, it returns.
- 111-112:** While the printer indicates it can receive 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.
- 115-117:** If the buffer contains fewer than LOWAT characters, and a process is asleep, waiting for room, wake it up.
- 121-122:** 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 because it occurs with the ACTIVE flag reset.

Sample Device Driver for Terminal

```
1  /*
2  ** td- terminal device driver
3  */
4  #include "../h/param.h"
5  #include "../h/dir.h"
6  #include "../h/user.h"
7  #include "../h/file.h"
8  #include "../h/tty.h"
9  #include "../h/conf.h"
10 #include "../h/idd.h"
11
12 /* registers */
13 #define RRDATA    0x01 /* received data */
14 #define RTDATA    0x02 /* transmitted data */
15 #define RSTATUS   0x03 /* status */
16 #define RCONTRL   0x04 /* control */
17 #define RIENABL   0x05 /* interrupt enable */
18 #define RSPEED    0x06 /* data rate */
19 #define RIIR      0x07 /* interrupt id */
20
21 /* status register bits */
22 #define SRRDY     0x01 /* received data ready */
23 #define STRDY     0x02 /* transmitter ready */
24 #define SOERR     0x04 /* rcvd data overrun */
25 #define SPERR     0x08 /* rcvd data par err */
26 #define SFERR     0x10 /* rcvd data frame err */
27 #define SDSR      0x20 /* status of dsr (cd) */
28 #define SCTS      0x40 /* clear to send status */
29
30 /* control register */
31 #define CBITS5    0x00 /* five bit chars */
32 #define CBITS6    0x01 /* six bit chars */
33 #define CBITS7    0x02 /* seven bit chars */
34 #define CBITS8    0x03 /* eight bit chars */
35 #define CDTR      0x04 /* data terminal ready */
36 #define CRTS      0x08 /* request to send */
37 #define CSTOP2    0x10 /* two stop bits */
38 #define CPARITY   0x20 /* parity on */
39 #define CEVEN     0x40 /* even parity (not odd) */
40 #define CBREAK    0x80 /* set xmitter to space */
41
```

Description of Device Driver for Terminal

This driver supports one serial terminal on a hypothetical UART type interface.

- 13-19:** The interface for each line consists of seven registers. The values that would be defined here represent offsets from the base address, which is defined in line 72. 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. The status of the UART can be determined by examining the contents of the RSTATUS register. The UART configuration is adjusted by changing the contents of the RCONTRL 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.
- 31-40:** The two low order-bits of the control register control 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 forces the transmitter to continuous spacing if the bit is set.

```
42 /* interrupt enable */
43 #define EXMIT      0x01 /* transmitter ready */
44 #define ERECV     0x02 /* receiver ready */
45 #define EMS       0x04 /* modem status change */
46
47 /* interrupt ident */
48 #define IRECV     0x01
49 #define IXMIT     0x02
50 #define IMS       0x04
51
52 #define NTDEVS    2
53 #define VECT0     3
54 #define VECT1     5
55 int tdopen(), tdclose(), tdcread(),
    tdwrite(), tdioc1(), tdintr();
56
57 /* Cdevsw entry for installable device driver */
58
```

-
- 43-45:** 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.
- 48-50:** After an interrupt, the interrupt identification register will contain a value that indicates the reason for the interrupt.

```

59 static struct cdevsw tdcdev = {
60     tlopen, tdclose, tdrread, tdwrite, tdiocctl, NU
61 };
62
63 /* Descriptor for installable device driver */
64
65 struct iddsw tdiddsw = {
66     IDD_IDD, IDD_VERS, /* Tag/version number
67     IDD_CDEV, /* Character device */
68     /* Entry points */
69     IDD_OPEN|IDD_CLOSE|IDD_READ|IDD_WRITE
70         |IDD_IOCTL,
71     NULL, /* bdevsw entry */
72     &tdcdev, /* cdevsw entry */
73     NULL, NULL /* linesw entry/init routine
74     tdintr, /* Interrupt handler */
75     -1, -1, -1, /* Filled from config.sys */
76     {-1 },
77     -1,
78     "td" /* Name */
79 };
80 /* data rates */
81 int td_speeds[] = {
82     /* B0 */ /* 0,
83     /* B50 */ /* 2304,
84     /* B75 */ /* 1536,
85     /* B110 */ /* 1047,
86     /* B134 */ /* 857,
87     /* B150 */ /* 768,
88     /* B200 */ /* 0,
89     /* B300 */ /* 384,
90     /* B600 */ /* 192,
91     /* B1200 */ /* 96,
92     /* B1800 */ /* 64,
93     /* B2400 */ /* 48,
94     /* B4800 */ /* 24,
95     /* B9600 */ /* 12,
96     /* EXTA */ /* 6, /* 19.2k bps */
97     /* EXTB */ /* 58 /* 2000 bps */
98 };
99
100 struct tty td_tty[NTDEVS];
101 int td_addr[NTDEVS] = { 0x00, 0x10 };

```

-
- 59:** *tdcdev* is the **struct cdevsw** which will be copied into the kernel **cdevsw** when this device driver is installed. The entry points are given exactly as if the kernel had been statically configured. *tdiddsw* is the **struct iddsw** which provides all the necessary information for this driver to be dynamically installed. Note that no **struct linesw** is needed for this driver since it uses the standard (i.e., zero) line discipline.
- 80-98:** The values to be loaded into the RSPEED register to get various data rates are defined here.
- 100:** Each line must have a **tty** structure allocated for it.
- 101:** Here, the base addresses of the registers are defined for each line.

```

102
103
104 tdopen(dev, flag)
105 int dev, flag;
106 {
107     register struct tty *tp;
108     int     addr;
109     int     tdproc();
110     int     x;
111
112     if (UNMODEM(dev) >= NTDEVS) {
113         seterror(ENXIO);
114         return;
115     }
116     tp = &td_tty[UNMODEM(dev)];
117     addr = td_addr[UNMODEM(dev)];
118     if((tp->t_lflag & XCLUDE) && !suser()) {
119         seterror(EBUSY);
120         return;
121     }
122     if ((tp->t_state&(ISOPEN!WOPEN)) == 0) {
123         ttinit(tp);
124         tp->t_proc = tdproc;
125         tp->t_oflag = OPOST!ONLCR;
126         tp->t_iflag = ICRNL!ISTRIP!IXON;
127         tp->t_lflag = ECHO!ICANON!
                    ISIG!ECHOE!ECHOK;
128         tdparam(dev);
129     }
130     x = splcli();
131     if (!ISMODEM(dev) ||
132         tp->t_cflag & CLOCAL ||
133         tdmodem(dev, TURNON))
134         tp->t_state != CARR_ON;
135     else
136         tp->t_state &= ~CARR_ON;
137     if (!(flag&FNDELAY))
138         while ((tp->t_state&CARR_ON)= 0) {
139             tp->t_state != WOPEN;
140             sleep((caddr_t)&tp->t_canq, TTIPRI);
141         }
142     (*linesw[tp->t_line].l_open)(tp);
143     splx(x);
144 }

```

tdopen() - lines 104 to 144

The **tdopen()** routine is called whenever a process makes an **open()** system call on the special file corresponding to this driver. Code is defined as follows:

- 112-114:** If the minor number indicates a device that does not exist, indicate the error and return.
- 118-120:** If the line is open for exclusive use but the current user is not the superuser, indicate the error and return.
- 122-129:** 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, and configure the line by calling **tdparam()**. Note that the flag initialization allows the terminal to perform well if the terminal is used as the console in single-user mode.
- 130:** Defer interrupts so that interrupt routines cannot change the state of the process while it is being examined.
- 131-136:** 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.
- 137-140:** If the **open()** routine is supposed to wait for the carrier, wait until the carrier is present.
- 142:** Call the **L_open()** routine indirectly through the **linesw** table. This completes the work required for the current line discipline to open a line.
- 143:** Allow further interrupts.

```
145
146 tdclose(dev)
147 int dev;
148 {
149     register struct tty *tp;
150
151     tp = &td_tty[UNMODEM(dev)];
152     (*linesw[tp->t_line].l_close)(tp);
153     if (tp->t_cflag & HUPCL)
154         tdmodem(dev, TURNOFF);
155     tp->t_lflag &= ~XCLUDE;
156     /* turn off exclusive use and interrupts */
157     out(td_addr[UNMODEM(dev)] + RIENABL, 0);
158 }
159
160 tddread(dev)
161 int dev;
162 {
163     register struct tty *tp;
164     tp = &td_tty[UNMODEM(dev)];
165     (*linesw[tp->t_line].l_read)
166         (&td_tty[UNMODEM(dev)]);
167 }
168
169 tddwrite(dev)
170 int dev;
171 {
172     register struct tty *tp;
173     tp = &td_tty[UNMODEM(dev)];
174     (*linesw[tp->t_line].l_write)
175         (&td_tty[UNMODEM(dev)]);
176 }
```

tdclose() - lines 146 to 158

The **tdclose()** routine is called on the last close on a line.

- 152:** Call the **close()** routine through the **linesw** table to do the work required by the current line discipline.
- 153-154:** If the hang-up-on-last-close bit is set, drop the data-terminal-ready and request-to-send signals.
- 155:** Reset the exclusive use bit.
- 157:** To prevent spurious interrupts, disable all interrupts for this line.

tdread() and tdwrite() - lines 160 to 172

These routines call the relevant routine via the **linesw** table; the called routine performs the action appropriate for the current line discipline.

```

173  tdpparam(dev)
174  int dev;
175  {
176      register int cflag;
177      register int addr;
178      register int temp, speed, x;
179
180      addr = td_addr[UNMODEM(dev)];
181      cflag = td_tty[UNMODEM(dev)].t_cflag;
182
183      /* if speed is B0, turn line off */
184      if ((cflag & CBAUD) == B0){
185          outb(addr + RCONTRL, inb(addr+RCONTRL)
186              ~CDTR & ~CRTS);
187          return;
188      }
189
190      /* set up speed */
191      outb(addr + RSPEED, td_speeds[ cflag & CBAUD ]);
192
193      /* set up line control */
194      temp = (cflag & CSIZE) >> 4; /* length */
195      if (cflag & CSTOPB)
196          temp |= CSTOP2;
197      if (cflag & PARENB) {
198          temp |= CPARITY;
199          if ((cflag & PARODD) == 0)
200              temp |= CEVEN;
201      }
202      temp |= CDTR | CRTS;
203      outb(addr + RCtrl, temp);
204
205      /* setup interrupts */
206      temp = EXMIT;
207      if (cflag & CREAD)
208          temp |= ERECV;
209      outb(addr + RENABL, inb(RENABL) | temp);
210  }

```

tdparam() - lines 173 to 209

The **tdparam()** routine configures the line to the mode specified in the appropriate **tty** structure.

180-181: Get the base address and flags for the referenced line.

184-186: The speed B0 means “hang up the line.”

190: The remainder of the **tdparam()** routine loads the device registers with the correct values.

```
211  tdmodem(dev, cmd)
212  int dev, cmd;
213  {
214      register int addr;
215
216      addr = td_addr[UNMODEM(dev)];
217      switch(cmd){
218          /* enable modem interrupts, set DTR & RTS true
219          case TURNON:
220              outb(addr + RENABL, inb(RENABL) | EMS);
221              outb(addr + RCONTRL, inb(RENABL) |
222                  CDTR | CRTS );
223
224              break;
225          /* disable modem interrupts, reset DTR, RTS */
226          case TURNOFF:
227              outb(addr + RENABL, inb(RENABL) & ~EMS);
228              outb(addr + RCONTRL, inb(RENABL) *
229                  ~(CDTR | CRTS) );
230
231              break;
232          }
233      return (inb(addr + RSTATUS) & SDSR);
234  }
```

tdmodem() - lines 211 to 228

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.

218-221: If **cmd** is **TURNON**, turn on modem interrupts and assert data-terminal-ready and request-to-send.

222-225: If **cmd** is **TURNOFF**, disable modem interrupts and drop data-terminal-ready and request-to-send.

227: Return a zero value if there is no data-set-ready on this line; otherwise return a non-zero value.

```
230 tdintr(vec)
231 int vec;
232 {
233     register int iir, dev, inter;
234
235     switch(vec) {
236         case VECT0:
237             dev = 0;
238             break;
239         case VECT1:
240             dev = 1;
241             break;
242         default:
243             printf(tdint: wrong level interrupt
244                    (%x)\n,vec);
245             return;
246     }
247     while((iir = inb(td_addr[dev]+RIIR)) != 0) {
248         if((iir & IXMIT) != 0)
249             tdxint(dev);
250         if((iir & IRECV) != 0)
251             tdrint(dev);
252         if((iir & IMS) != 0)
253             tdmint(dev);
254     }
255 }
```

tdintr() - lines 230 to 254

The **tdintr()** routine determines which line caused the interrupt and the reason for the interrupt. The routine then calls the appropriate routine to handle the interrupt.

- 235-244:** Different lines result in different interrupt vectors being passed as the **tdintr()** routine's argument. Here, the minor number is determined from the interrupt vector that is passed to **tdintr()**.
- 246-252:** 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 resets the bit in the register.

```
256 tdxint(dev)
257 int dev;
258 {
259     register struct tty *tp;
260     register int addr;
261
262     tp = &td_tty[UNMODEM(dev)];
263     addr = td_addr[UNMODEM(dev)];
264     if (inb(addr + RSTATUS) & STRDY)
265     {
266         tp->t_state &= ~BUSY;
267         if (tp->t_state & TTXON) {
268             outb(addr + RTDATA, CSTART);
269             tp->t_state &= ~TTXON;
270         } else if (tp->t_state & TTXOFF) {
271             outb(addr + RTDATA, CSTOP);
272             tp->t_state &= ~TTXOFF;
273         } else
274             tdproc(tp, T_OUTPUT);
275     }
276 }
277
```

tdxint() - lines 256 to 276

The **tdxint()** routine is called when a transmitter-ready interrupt is received. This routine does one of the following tasks:

- Issues a CSTOP character to indicate that the device on the other end must stop sending characters
- Issues a CSTART character to indicate that the device on the other end may resume sending characters
- Calls **tdproc()** to send the next character in the queue.

264: If the transmitter is ready, reset the busy indicator.

267-269: If the line is to be restarted, send a CSTART, and reset the indicator.

270-272: If the line is to be stopped, send a CSTOP, and reset the character.

273-274: Otherwise, call **tdproc()** and ask it to send the next character in the queue.

```

278 tdrint(dev)
279 int dev;
280 {
281     register int c, status;
282     register int addr;
283     register struct tty *tp;
284
285     tp = &td_tty[UNMODEM(dev)];
286     addr = td_addr[UNMODEM(dev)];
287
288     /* get char and status */
289     c = inb( addr + RRDATA);
290     status = inb(addr + RLSR);
291
292     /*
293      * Were there any errors on input?
294      */
295     if(status & SOERR)           /* overrun error */
296         c != OVERRUN;
297     if(status & SPERR)          /* parity error */
298         c != PERROR;
299     if(status & SFERR)          /* framing error */
300         c != FRERROR;
301
302     if (tp->t_rbuf.c_ptr == NULL)
303         return;
304     flg = tp->t_iflag;
305     if (flg&IXON) {
306         register int ctmp;
307         ctmp = c & 0177;
308         if(tp->t_state & TTSTOP) {
309             if (ctmp == CSTART || flg&IXANY)
310                 (*tp->t_proc)(tp, T_RESUME);
311         } else {
312             if (ctmp == CSTOP)
313                 (*tp->t_proc)(tp, T_SUSPEND);
314         }
315         if (ctmp == CSTART || ctmp == CSTOP)
316             return;
317     }
318     if (c&PERROR && !(flg&INPCK))
319         c &= ~PERROR;
320     if (c&(FRERROR|PERROR|OVERRUN)) {
321         if ((c&0377) == 0) {
322             if (flg&IGNBRK)
323                 return;

```

```

324         if (flg&BRKINT) {
325             (*linesw[tp->t_line].l_input)
326                 (tp, L_BREAK);
327             return;
328         }
329     } else {
330         if (flg&IGNPAR)
331             return;
332     }
333 } else {
334     if (flg&ISTRIP)
335         c &= 0177;
336     else {
337         c &= 0377;
338     }
339 }
340 *tp->t_rbuf.c_ptr = c;
341 tp->t_rbuf.c_count--;
342 (*linesw[tp->t_line].l_input)(tp, L_BUF);
343 }
344
```

tdrint() - lines 278 to 343

The **tdrint()** routine is called when a receiver interrupt is received. It passes the character, along with any errors, to the appropriate routine via the **linesw** table.

288-290: Get the character and status.

295-340: If any errors were detected, set the appropriate bit in *c*.

305-317: This code determines whether the character is X-ON and if output is stopped, it restarts it. If the character is X-OFF, output is suspended.

Further error checking is then carried out and characters in error are discarded. The character is then placed in the queue.

342: Pass the character and errors to the **Linput()** routine for the current line discipline.

```

345 tdmint(dev)
346 int dev;
347 {
348     register struct tty *tp;
349     register int addr,c;
350
351     tp = &td_tty[UNMODEM(dev)];
352     if (tp->t_cflag & CLOCAL) {
353         return;
354     }
355     addr = td_addr[UNMODEM(dev)];
356
357     if (inb(addr + RSTATUS) & SDSR) {
358         if ((tp->t_state & CARR_ON) = 0) {
359             tp->t_state != CARR_ON;
360             wakeup(&tp->t_canq);
361         }
362     } else {
363         if (tp->t_state & CARR_ON) {
364             if (tp->t_state & ISOPEN) {
365                 signal(tp->t_pgrp, SIGHUP);
366                 tdmodem(dev, TURNOFF);
367                 ttyflush(tp, (FREAD|FWRITE));
368             }
369             tp->t_state &= ~CARR_ON;
370         }
371     }
372 }
373
374 tdiocctl(dev, cmd, arg, mode)
375 int dev;
376 int cmd;
377 faddr_t arg;
378 int mode;
379 {
380     if (ttiocom(&td_tty[UNMODEM(dev)],
381                cmd, arg, mode))
382         tdparam(dev);
383 }

```

tdmint() - lines 345 to 372

The **tdmint()** routine is called whenever a modem interrupt occurs.

352-353: If there is no modem support for this line, just return.

357-360: If a data-set-ready is present for this line but it was not before, mark the line as having a carrier. Wake up any processes that are waiting for the carrier before their **tdopen()** call can be completed.

287-296: If no data-set-ready is present for this line but one existed before, send a hang up signal to all of the processes associated with this line. Call **tdmodem()** to hang up the line and flush the output queue for this line by calling **ttyflush()**. Finally, mark the line as having no carrier.

tdioctl() - lines 374 to 382

The **tdioctl()** routine is called when a process makes an **ioctl()** system call on a device associated with the driver. This routine calls the **ttiocom()** routine which returns a non-zero value if the hardware must be reconfigured.

```
384  tdproc(tp, cmd)
385  register struct tty *tp;
386  int cmd;
387  {
388      register c;
389      register int addr;
390
391      extern ttrstrt();
392
393      addr = td_addr[tp - td_tty];
394      switch (cmd) {
395
396      case T_TIME:
397          tp->t_state &= ~TIMEOUT;
398          outb(addr + RCtrl, inb(addr + RCtrl) &
399              ~CBREAK);
400          goto start;
401
402      case T_WFLUSH:
403          tp->t_tbuf.c_size -= tp->t_tbuf.c_count;
404          tp->t_tbuf.c_count = 0;
405      case T_RESUME:
406          tp->t_state &= ~TTSTOP;
407          goto start;
```

tdproc() - lines 384 to 461

The **tdproc()** routine is called to make a change to the output, such as emitting the next character in the queue or halting or restarting the output.

- 394:** The *cmd* argument determines the action taken.
- 386-399:** 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 the **start()** routine. A **WFLUSH** command resets the character buffer pointers and the count.
- 405-406:** Either a line on which output was stopped is restarting, or someone is waiting for the output queue to decrease. Reset the flag indicating that output on this line is stopped, and start the output again by jumping to the **start()** routine (line 409).

```

408     case T_OUTPUT:
409     start:
410         if (tp->t_state & (TIMEOUT|TTSTOP|BUSY))
411             break;
412         {
413             register struct ccblock *tbuf;
414
415             tbuf = &tp->t_tbuf;
416             if (tbuf->c_ptr == NULL ||
417                 tbuf->c_count == 0) {
418                 if (tbuf->c_ptr)
419                     tbuf->c_ptr -= tbuf->c_size
420                                 - tbuf->c_count;
421                 if (!(CPRES &
422                     (*linesw[tp->t_line].l_output)(tp))
423                     break;
424             }
425             tp->t_state != BUSY;
426             outb(addr + RTHR, *tbuf->c_ptr++);
427             tbuf->c_count--;
428         }
429         break;
430
431     case T_SUSPEND:
432         tp->t_state != TTSTOP;
433         break;
434

```

-
- 410-411:** 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.
- 412-427:** 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**) adjusted appropriately.
- 431-433:** 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.

```

435     case T_BLOCK:
436         tp->t_state &= ~TTXON;
437         tp->t_state != TBLOCK;
438         if (tp->t_state&BUSY)
439             tp->t_state != TTXOFF;
440         else
441             outb(addr + RTDATA, CSTOP);
442         break;
443
444     case T_RFLUSH:
445         if (!(tp->t_state&TBLOCK))
446             break;
447     case T_UNBLOCK:
448         tp->t_state &= ~(TTXOFF!TBLOCK);
449         if (tp->t_state&BUSY)
450             tp->t_state != TTXON;
451         else
452             outb(addr + RTDATA, CSTART);
453         break;
454
455     case T_BREAK:
456         outb(addr + RCtrl, inb(addr + RCtrl) | CBRE);
457         tp->t_state != TIMEOUT;
458         timeout(ttrstrt, tp, HZ/4);
459         break;
460     }
461 }

```

-
- 435-442:** To tell the device on the other end to stop sending characters, reset the flag to stop the line and mark the line stopped. If the line is already busy, set the flag; otherwise, output a CSTOP character.
- 444-446:** A process is waiting to flush the input queue. If the device has not been blocked, just return. Otherwise, drop through and unblock the device.
- 447-453:** 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.
- 455-459:** 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.

Sample Device Driver for Disk Drive

```
1  /*
2  ** hd- prototype fixed disk driver
3  */
4
5  #include "../h/param.h"
6  #include "../h/buf.h"
7  #include "../h/iobuf.h"
8  #include "../h/dir.h"
9  #include "../h/conf.h"
10 #include "../h/user.h"
11 #include "../h/idd.h"
12
13 /* disk parameters */
14 #define NHD 4 /* number of drives */
15 #define NCPD 600 /* # cylinders/disk */
16 #define NTPC 4 /* # tracks/cylinder */
17 #define NSPT 10 /* # sectors/track */
18 #define NBPS 512 /* # bytes/sector */
19 #define NSPB (BSIZE/NBPS) /* sectors/block */
20 #define NBPC (NTPC*NSPT*NSPB) /* blocks/cyl */
21
22 /* addresses of controller registers */
23 #define RBASE 0x00 /* base of all registers */
24 #define RCMD (RBASE+0) /* command register */
25 #define RSTAT (RBASE+1) /* stat-nonzero=err */
26 #define RCYL (RBASE+2) /* target cylinder */
27 #define RTRK (RBASE+3) /* target track */
28 #define RSEC (RBASE+4) /* target sector */
29 #define RADDRL (RBASE+5) /* t mem addr lo 16 bits */
30 #define RADDRH (RBASE+6) /* t mem addr hi 8 bits */
31 #define RCNT (RBASE+7) /* # sectors to xfer */
32
33 /* bits in RCMD register */
34 #define CREAD 0x01 /* start a read */
35 #define CWRITE 0x02 /* start a write */
36 #define CRESET 0x03 /* reset the controller */
37
```

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.

- 14:** NHD defines the number of drives the controller can be attached to.
- 15-20:** Each disk drive attached to the controller has NCPD cylinders; each cylinder has NTPC tracks, and each track has NSPT sectors. The sectors are NBPS bytes long, and each cylinder has NBPC blocks.
- 22-31:** The controller registers occupy a region of contiguous address space starting at RBASE and running through RBASE + 7.
- 34-36:** To make the controller perform your specified 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.

```

38 /*
39 ** minor number layout is 0000dppp
40 ** where d is the drive number
41 ** and ppp is the partition */
42 #define drive(d)      (minor(d) >> 3)
43 #define part(d)      (minor(d) & 0x07)
44
45 /* partition table */
46 struct partab {
47     daddr_t len;
48     /* # of blocks in partition */
49     int      cyloff;
50     /* starting cylinder of partition */
51 };
52
53 int hread(), hwrite(), hintr(), hdstrategy();
54
55 /* Bdevsw and cdevsw entries
56 ** for installable driver */
57 static struct cdevsw hdcdev = {
58     nulldev, nulldev, hread, hwrite, nulldev, NI
59 };
60
61 static struct bdevsw hdbdev = {
62     nulldev, nulldev, hdstrategy, &hdtab
63 };
64
65 /* Installable driver descriptor */
66
67 struct iddsw hiddsw = {
68     IDD_IDD, IDD_VERS, /* Tag/version number */
69     IDD_CDEV!IDD_BDEV, /* Char/block device */
70     /* Entry points */
71     IDD_READ!IDD_WRITE,
72     &hdbdev, /* bdevsw entry */
73     &hdcdev, /* cdevsw entry */
74     NULL, NULL, /* linesw entry/init routine */
75     hintr, /* Interrupt handler */
76     {-1, -1, -1, /* Filled from config.sys */
77      {-1 }},
78     -1,
79     "hd" /* Name */
80 };

```

-
- 42-43:** 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.
- 46-48:** Large disks are usually divided into partitions of a 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.
- 51-77:** *hdbdev* is the **struct bdevsw** which will be copied into the kernel **bdevsw** when this device driver is installed. *hdcdev* is the **struct cdevsw** which will be copied into the kernel **cdevsw** when this device driver is installed. The entry points are given exactly as if the kernel had been statically configured. *hdiddsw* is the **struct iddsw** which provides all the necessary information for this driver to be dynamically installed.

```

80 struct partab hd_sizes[8] = {
81     NCPD*NBPC,      0,      /* whole disk */
82     ROOTSZ*NBPC,   0,      /* root area */
83     SWAPSZ*NBPC,   ROOTSZ, /* swap area */
84     USERSZ*NBPC,   USROFS, /* usr area */
85     0,             0,      /* spare */
86     0,             0,      /* spare */
87     0,             0,      /* spare */
88     0,             0,      /* spare */
89 };
90
91 struct iobuf      hdtab; /* start of request queue
92 struct buf        rhdbuf; /* header for raw i/o */
93 /*
94 **      Strategy Routine:
95 **      Arguments:
96 **          Pointer to buffer structure
97 **      Function:
98 **          Check validity of request
99 **          Queue the request
100 **          Start up the device if idle
101 */

```

-
- 81-84:** This driver can divide a disk into as many as eight partitions. For now, only four partitions are used. The first partition covers the entire disk. The remaining three divide the disk three ways, one partition for the root directory, one for the swap directory, and one for the usr directory.
- 91:** 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**.
- 92:** Each block driver that wants to allow raw I/O allocates one buffer header for this purpose.

```

102 int hdstrategy(bp)
103 register struct buf *bp;
104 {
105     register int dr, pa;
106     /* drive and partition numbers */
107     daddr_t sz, bn;
108     int x;
109     dr = drive(bp->b_dev);
110     pa = part(bp->b_dev);
111     bn = bp->b_blkno * NSPB;
112     sz = (sz + BMASK) >> BSHIFT;
113     if (dr<NHD && pa<NPARTS && bn>=0 &&
114         bn<hd_sizes[pa].len &&
115         ((bn + sz < hd_sizes[pa].len) ||
116          (bp->b_flags & B_READ)))
117     {
118         if (bn + sz > hd_sizes[pa].len) {
119             sz = (hd_sizes[pa].len - bn) * NBPS;
120             bp->b_resid = bp->b_bcount -
121                 (unsigned) sz;
122             bp->b_bcount = (unsigned) sz;
123         }
124     } else {
125         bp->b_flags |= B_ERROR;
126         iodone(bp);
127         return;
128     }
129     bp->b_cylin = (b_blkno / NBPC) +
130                 hd_sizes[pa].cyloff;
131     x = splbuf();
132     disksort(&hdtab, bp);
133     if (dp->b_active == NULL)
134         hdstart();
135     splx(x);
136 }

```

hdstrategy() - lines 102 to 131

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. This routine validates the request and links it into the queue of outstanding requests.

108-111: First, compute various useful numbers that will be used repeatedly during the validation process.

112-124: The **B_ERROR** bit in the *b_flags* field of the header is set to indicate that the request has failed if any of the following conditions are met:

- If the request is for a non-existent drive or a non-existent partition
- If the requested target lies completely outside the specified partition
- If the request is a write request that ends outside the partition.

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.

125: Compute the target cylinder of the request for the benefit of the **disksort()** routine.

126: Block interrupts, in order to prevent the interrupt routine from changing the queue of outstanding requests.

127: Sort the request into the queue by passing it and the head of the queue to **disksort()**.

-
- 128:** If the controller is not already active, start it up.
- 129:** Re-enable interrupts and return to the user process.

```
133 /*
134 * Startup Routine:
135 * Arguments:
136 * None
137 * Function:
138 * Compute device-dependent parameters
139 * Start up device
140 * Indicate request to I/O monitor routines
141 */
142 hdstart()
143 {
144     register struct buf *bp; /* BUFFER POINTER */
145     register unsigned sec;
146
147     if ((bp = hdtab.b_actf) == NULL) {
148         hdtab.b_active = 0;
149         return;
150     }
151     hdtab.b_active = 1;
152
153     sec = (unsigned)bp->blkno * NSPB);
154     out(RCYL, sec / NSPC); /* cylinder */
155     sec %= NSPC;
156     out(RTRK, sec / NSPT); /* track */
157     out(RSEC, sec % NSPT); /* sector */
158     out(RCNT, bp->b_count / NBPS); /* count */
159     out(RDRV, drive(bp->b_dev)); /* drive */
160     out(RADDRL, bp->b_paddr & 0xffff);
161         /* memory address low */
162     out(RADDRH, bp->b_paddr >> 16);
163         /* memory address high */
164     if (bp->b_flags & B_READ)
165         out(RCMD, CREAD);
166     else
167         out(RCMD, CWRITE);
168 }
```

hdstart() - lines to 142 to 166

The **hdstart()** routine performs the calculation of the physical address on the disk and starts the transfer.

147-149: If there are no active requests, mark the state of the driver as idle and return.

151: Mark the state of the driver as active.

153-157: Calculate the starting cylinder, track, and sector of the request. Then, load the controller registers with these values.

159-161: Load the controller with the drive number and memory address of the data to be transferred.

162-165: If the request is a read request, issue a read command; otherwise, issue a write command.

```

168 /*
169  * Interrupt routine:
170  *   Check completion status
171  *   Indicate completion to i/o monitor routines
172  *   Log errors
173  *   Restart (on error) or start next request
174  */
175 hdintr()
176 {
177     register struct buf *bp;
178
179     if (hdtab.b_active == 0)
180         return;
181
182     bp = hdtab.b_actf;
183
184     if (in(RSTAT) != 0)
185         out(RCMD, CRESET);
186     if (++hdtab.b_errcnt <= ERRLIM) {
187         hdstart();
188         return;
189     }
190     bp->b_flags |= B_ERROR;
191     de verr(&hdtab, bp, in(RSTAT), 0);
192 }
193 /* Flag current request complete,
194  * start next one
195  */
196 hdtab.b_errcnt = 0;
197 hdtab.b_actf = bp->av_forw;
198 bp->b_resid = 0;
199 iodone(bp);
200 hdstart();
201 }
202

```

hdintr() - lines 175 to 201

The kernel calls the **hdintr()** routine through the **vecintsw** table whenever the controller issues an interrupt.

179-180: If an unexpected call occurs, just return.

182: Get a pointer to the first buffer header in the chain; this is the request that is currently being serviced.

184-192: If the controller indicates an error and the operation has not 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.

196-201: Mark this request completed, remove it from the request queue, and call **hdstart()** to start on the next request.

```

203 /* raw read routine:
204 *   This routine calls physio which
205 *   computes and validates a physical
206 *   address from the current logical address.
207 *
208 * Arguments
209 *   Full device number
210 * Functions:
211 *   Call physio to do raw (physical) I/O
212 *   The arguments to physio are:
213 *     pointer to the strategy routine
214 *     buffer for raw I/O
215 *     device
216 *     read/write flag
217 */
218 hread(dev)
219 int dev;
220 {
221     physio(hdstrategy, &rhdbuf, dev, B_READ);
222 }
223
224 /*
225 * Raw write routine:
226 * Arguments(to hwrite):
227 *   Full device number
228 * Functions:
229 *   Call physio to do raw (physical) I/O
230 */
231 hwrite(dev)
232 int dev;
233 {
234     physio(hdstrategy, &rhdbuf, dev, B_WRITE);
235 }

```

hbread() - lines 218 to 222

The kernel calls the **hbread()** routine when a process requests a raw read of the device. This routine calls the **physio()**, routine and passes to it the name of the strategy routine, a pointer to the raw buffer header, the device number, and a flag indicating a read request. The **physio()** routine does all the preliminary work and queues the request by calling the device strategy routine.

hdwrite() - lines 231 to 235

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 **hbread()**, except that the flag it passes indicates a write request.

Writing Drivers for Memory-Mapped Screens

This section provides the information you need to write a device driver for a memory-mapped screen. To produce a `tty` style driver for a memory-mapped device, you need to write two pieces of code.

The first piece of code you will need to write is the device driver. This part of the code is described in “Sample Device Driver for Terminal” on page 11-10. It includes the `open()`, `close()`, `read()`, `write()`, `ioctl()`, and interrupt routines. Note that because the output display is memory mapped, there are no output interrupts, only input interrupts. In addition, this device driver should include the supplied `xxproc()` routine. This routine pulls the characters off the output queue in blocks and passes them, one block at a time, to the supplied ANSI support code.

The second piece of code to be written consists of a set of routines to manipulate the screen memory. The screen model is a screen with `NROWS` rows and `NCOLS` columns. Addresses in this memory are specified as a (row, column) pair. The functions that need to be written are:

v_scroll(*i*) Scroll the text on the screen *i* number of lines. This will move *i* lines of text off the top of the screen, and *i* blank lines onto the bottom of the screen. If *i* is negative, the text moves downward off the bottom of the screen and blank lines appear at the top.

v_copy(*sr*, *sc*, *dr*, *dc*, *cnt*)
sr and *sc* specify a source row and column.

dr and *dc* specify a destination row and column.

Count characters (*cnt*) are copied from the source to the destination, with the copy proceeding from left to right, and top to bottom. If the source and destination overlap, the copy is done correctly.

v_clear(*r, c, cnt*)

Characters starting at row *r* and column *c* are cleared to the space character.

cnt is the number of characters cleared.

v_pchar(*r, c, ch*)

The character *ch* is placed on the screen at row *r*, column *c*, using the current graphic rendition. The return value is the number of character positions the active position is to be adjusted. Zero means the character has no graphic representation.

v_scurs(*r, c*)

The cursor is moved to row *r*, column *c*.

v_init() The screen and all data structures are initialized.

v_sgr(*i*) The current graphic rendition (for example, font and color) is set to *i*. See **Console(M)** for encoding.

v_beep() Causes a beep, bell, or other alarm indication to sound. Used for the ASCII “bel” character.

You must also provide an initialized declaration for the **crtsw** data structure; the ANSI code indirectly calls the routines through this structure. The data declaration for this data structure is provided in the **/usr/sys/io/crt.h** file.

Index

A

- a option
 - lint 3-11
- accessing registers 10-12
- adb
 - addresses,
 - validating 6-37
 - core image 6-3
 - data files 6-3
 - debugging program 1-3
 - displaying
 - instructions 6-6
 - input format 6-41
 - locating values 6-46
 - memory maps 6-34
 - patching binary files 6-46
 - prompt option 6-5
 - starting 6-1, 6-3
 - stopping 6-1
 - write option 6-4
 - writing to a file 6-47
- adb, program
 - debugger 6-1
- aliasing 1-7
- allocating
 - descriptors 10-20
- ar
 - description 1-5
- arguments 9-9
- arithmetic built ins 9-11
- as, program assembler
 - assembler program 1-4
- assembler

- See as, program
 - assembler
- assembler language
 - source 2-4
- awaking processing 10-16

B

- b option
 - lint 3-5
- block devices
 - device drivers 10-50
- breakpoints 6-23

C

- C compiler
 - expression
 - evaluation
 - order 3-16
 - lint directives,
 - effect 3-17
- C language
 - compiler 1-3
 - usage check 1-3
 - yacc 8-2
- c option
 - lint 3-9
- C programming language 1-2
- C programs
 - creating 1-3

- string extraction 1-6
- C-shell
 - aliasing 1-7
 - command history
 - mechanism 1-7
 - command language 1-7
- character devices
 - device drivers 10-33
- character lists 10-44
- clists
 - See clists
- command
 - execution 1-7
 - interpretation 1-7
- conditionals 9-13
- configuring the system 10-68
- controlling registers 10-12
- copyio() 10-25
- copyio() routine 10-25
- cross-development system 2-1
- crtsw data structure 11-53
- csh command description 1-7

D

- data string
 - crtsw 11-53
- debugger
 - See adb
- debugging a DOS program 2-5
 - between systems 2-6
 - transferring programs 2-6
- defining registers 11-11
- delta

- See SCCS
- desk calculator
 - specifications 8-44
- determining interrupt vector numbers 10-73
- device driver
 - block devices 10-50, 10-51
 - character devices 10-33, 10-45
 - character interface 10-51
 - definition 10-1
 - disk drives 11-38
 - freeing
 - descriptors 10-21
 - GDT descriptors 10-20
 - guidelines for writing 11-1
 - I/O control 11-31
 - initializing
 - descriptors 10-21
 - interrupt routines 10-35, 11-25, 11-27, 11-49
 - interrupt routines for character devices 10-43
 - line discipline routines 10-42
 - line printer 11-2
 - line printer routines 11-7
 - line printers 10-48
 - magnetic tape 10-49
 - memory mapped screens 11-52
 - modem routines 11-23
 - naming
 - conventions 10-33
 - overview 10-1
 - routines 10-33

sample code 11-1
scheduling 10-17
terminal 11-10
terminals 10-45
warnings 10-76
writing 10-5
writing installable
drivers 10-61
device drivers 10-1
device models
block devices 10-2
character devices 10-2
disambiguating rule 8-1
disk drive
device drivers 11-38
DOS libraries 2-7
DOS object files 2-5
DOS source file 2-3
dscralloc() routine 10-20

E

error message
file creation 1-6
printing 10-42
errprint built-in 9-20
executing a program 6-20

F

file
See also SCCS
archives 1-5
block counting 1-6
check sum
computation 1-6
error message file
See error message
octal dump 1-6

relocation bits
removal 1-6
removal
See SCCS
symbol removal 1-6
text search, print 1-6
FORTRAN
conversion
program 7-36
freeing descriptors 10-21

G

GDT descriptors
device drivers 10-20
getc() routine 10-44
getcb() routine 10-39
getcfc() routine 10-40

H

-h option
lint 3-13
hard disk routines
hdintr() 11-49
hddread() 11-51
hdstart() 11-47
hdstrategy() 11-45
hdwrite() 11-51
hdintr
See hard disk routines
hddread
See hard disk routines
hdstart
See hard disk routines
hdstrategy
See hard disk routines
hdwrite

See hard disk routines
hexadecimal dump 1-6

I

in() routine 10-12
inb() routine 10-12
initializing descriptors
 device drivers 10-21
installable device
 drivers 10-61, 10-70
interrupt routines
 character device 10-33
 rules 10-11
interrupt service
 routine 10-9
interrupt-time
 processing 10-9
interrupt vectors
 determining
 numbers 10-73
 sharing 10-74
interrupts
 acknowledgement 10-14
 character device
 drivers 10-43
 disable
 interrupts 10-14
 enable interrupts 10-14
 no
 acknowledgement 10-14
ioctl() interface 10-26
iomove() 10-27

K

kernel functions 10-6
kernel routines
 console display
 routine 10-29
 data transfer 10-25
 miscellaneous
 functions 10-30
 tty routines 10-23

L

LALR 8-1
ld
 link editor 1-4
lex
 action
 default 7-14
 description 7-5
 repetition 7-14
 specification 7-14
 alternation 7-11
 ambiguous source
 rules 7-19
 angle brackets
 operator
 character 7-7, 7-33
 start condition
 referencing 7-24
 arbitrary character
 match 7-10
 array size change 7-32
 asterisk
 operator
 character 7-7, 7-33
 repeated expression
 specification 7-11
 automaton interpreter

initial condition
 resetting 7-24

backslash
 C escapes 7-8
 operator
 character 7-7, 7-33
 operator character
 escape 7-8, 7-10

BEGIN
 start condition
 entry 7-24

blank character
 quoting 7-8
 rule ending 7-8

blank, line
 beginning 7-26

braces
 expression
 repetition 7-13
 operator
 character 7-7, 7-33

brackets
 character class
 specification 7-9
 character class
 use 7-2
 operator
 character 7-7, 7-33
 operator character
 escape 7-9

buffer overflow 7-20

C escapes 7-8

caret
 character class
 inclusion 7-9
 context
 sensitivity 7-12
 operator
 character 7-7, 7-33
 string
 complement 7-9

caret operator
 left context
 recognizing 7-23

character
 internal use 7-29
 set table 7-29, 7-32
 translation table See
 set table 7-29

character class
 notation 7-2
 specification 7-9

character set
 specification 7-29

context
 sensitivity 7-12

copy classes 7-26

dash
 character class
 inclusion 7-9
 operator
 character 7-7, 7-33
 range indicator 7-9

definition
 character set
 table 7-29
 contents 7-27, 7-31
 expansion 7-13
 format 7-26, 7-31
 location 7-26
 placement 7-13
 specification 7-26

delimiter
 discard 7-26
 rule beginning
 marking 7-2
 source format 7-5
 third delimiter,
 copy 7-26

description 1-4

dollar sign
 context
 sensitivity 7-12

- end of line
 - notation 7-2
- operator
 - character 7-7, 7-33
- dollar sign operator
 - right context
 - recognizing 7-23
- dot operator See
 - period 7-20
- double precision
 - constant change 7-37
- ECHO
 - format argument,
 - data printing 7-15
- end-of-file
 - yywrap()
 - routine 7-18
 - 0 handling 7-18
- environment
 - change 7-23
- expression
 - new line illegal 7-8
 - repetition 7-13
- external character
 - array 7-14
- flag
 - environment
 - change 7-23
- FORTRAN conversion
 - program 7-36
- grouping 7-11
- I/O library See
 - library 7-4
- I/O routine
 - access 7-17
 - consistency 7-17
- input
 - description 7-1
 - end-of-file, zero
 - notation 7-17
 - ignoring 7-14
 - manipulation
 - restriction 7-22
- input routine
 - character I/O
 - handling 7-29
- input() routine 7-17
- invocation 7-4
- left context 7-12
 - caret operator 7-23
 - sensitivity 7-23
- lex.yy.c file 7-4
- lexical analyzer
 - environment
 - change 7-23
- library
 - access 7-4
 - avoidance 7-4
 - backup
 - limitation 7-18
 - loading 7-28
- line beginning
 - match 7-12
- line end match 7-12
- ll flag
 - library access 7-4
- loader flag See -ll
 - flag 7-4
- lookahead
 - characteristic 7-15,
 - 7-17
- match count 7-15
- matching
 - occurrence
 - counting 7-21
- newline
 - escape 7-29
- octal escape 7-10
- operator characters
 - designated 7-33
 - escape 7-8, 7-9, 7-10
 - listing 7-7
 - literal meaning 7-8
 - operand types
 - balancing 3-7

- quoting 7-8
- optional expression
 - specification 7-10
- output (c) routine 7-17
- output routine
 - character I/O
 - handling 7-29
- parentheses
 - grouping 7-11
 - operator
 - character 7-7, 7-33
- parser generator
 - analysis phase 7-2
- percentage sign
 - delimiter
 - notation 7-2
 - operator
 - character 7-7
 - remainder
 - operator 7-34
 - source segment
 - separator 7-13
- period
 - arbitrary character
 - match 7-10
 - newline no
 - match 7-20
 - operator
 - character 7-7
- period operator
 - designated 7-33
- plus sign
 - operator
 - character 7-7, 7-33
 - repeated expression
 - specification 7-11
- preprocessor statement
 - entry 7-26
- question mark
 - operator
 - character 7-7, 7-33
- optional expression
 - specification 7-10
- quotation marks, double
- operator
 - character 7-33
 - operator character
 - escape 7-8
- real numbers rule 7-27
- regular expression
 - See also lex,
 - operator characters
 - description 7-7
 - end indication 7-5
 - rule component 7-5
- REJECT 7-21, 7-22
- repeated expression
 - specification 7-11
- right context
 - dollar sign
 - operator 7-23
- rules
 - active 7-25
 - components 7-5
 - format 7-32
 - real number 7-27
- semicolon
 - null statement 7-14
- slash
 - operator
 - character 7-7, 7-33
 - trailing text 7-12
- source
 - copy into generated
 - program 7-26
 - description 7-1
 - format 7-5, 7-26
 - interception
 - failure 7-26
 - segment
 - separator 7-13
- source definitions
 - specification 7-26

- source file
 - format 7-31
- source program
 - compilation 7-4
- spacing character
 - ignoring 7-14
- start
 - abbreviation 7-24
- start condition 7-12
- start conditions
 - entry 7-24
 - environment
 - change 7-23
 - format 7-31
 - location 7-31
- statistics
 - gathering 7-35
- string
 - printing 7-5
- substitution string
 - See lex, definition
- tab, line
 - beginning 7-26
- text character
 - quoting 7-8
- trailing text 7-12
- unput
 - REJECT
 - noncompatible 7-22
- unput (c) routine 7-17
- unput routine
 - character I/O
 - handling 7-29
- unreachable
 - statement 3-5
- vertical bar
 - action
 - repetition 7-14
 - alternation 7-11
 - operator
 - character 7-7, 7-33
- yacc
 - interface 7-2
 - library loading 7-28
- yacc interface
 - tokens 7-28
 - yylex() 7-28
- yy leng variable 7-15
- yyless
 - text
 - reprocessing 7-16
- yyless(n) 7-15
- yy lex() program
 - contents 7-1
 - yacc interface 7-28
- yymore() 7-15
- yytext
 - external character
 - array 7-14
- yywrap()
 - yywrap() routine 7-18, 7-35
- 0, end of file
 - notation 7-18
- lex, description 7-1
- lex, program
 - generator 7-1
- library
 - conversion 1-5
 - maintenance 1-5
 - ordering relation 1-5
 - sort 1-5
- line discipline routines
 - device driver 10-42
 - L_close() 10-42
 - L_input() 10-42
 - L_ioctl() 10-42
 - L_lmdmint() 10-42
 - L_open() 10-42
 - L_output() 10-42
 - L_read() 10-42
 - L_write() 10-42
- line printer
 - device driver 10-48, 11-2

interrupt routines 11-9
 line printer routines
 lpclose() 11-7
 lpintr() 11-9
 lpopen() 11-7
 lpstart() 11-8
 lpwrite() 11-7
 linkable device
 drivers 10-69
 linking object files 2-5
 lint
 -a option 3-11
 ARGSUSED
 directive 3-17, 3-18
 argument number
 comments
 turnoff 3-17
 assignment
 See also lint, type
 check
 assignment operator
 new form 3-14
 of long to int,
 check 3-11
 old form, check 3-14
 operand type
 balancing 3-7
 -b option 3-5
 binary operator, type
 check 3-7
 break statement
 See lint, unreachable
 break
 C language check 1-3
 -c option 3-9
 C program check 3-1
 C syntax, old form,
 check 3-14
 cast
 See lint, type cast
 conditional operator,
 operand type
 balancing 3-7
 constant in conditional
 context 3-13
 construction
 check 3-1, 3-12
 degenerate unsigned
 comparison 3-12
 description 3-1
 directive
 defined 3-17
 embedding 3-17
 enumeration, type
 check 3-7
 error message, function
 name 3-6
 expression, order 3-16
 extern statement 3-3
 external declaration,
 report suppression 3-3
 file
 library declaration
 file
 identification 3-17
 function
 See also lint, unused
 function
 error message 3-6
 return value
 check 3-6
 type check 3-7
 -h option 3-13
 initialization, old style
 check 3-14
 library
 compatibility
 check 3-18
 compatibility check
 suppression 3-18
 directive
 acceptance 3-18

- file processing 3-18
- LINTLIBRARY
 - directive 3-17, 3-18
- loop check
- ly directive 3-18
- n option 3-18
- nonportable character
 - check 3-10
- nonportable expression
 - evaluation order
 - check 3-16
- NOTREACHED
 - directive 3-17
- output turnoff 3-17
- p option 3-18
- pointer
 - agreement 3-7
 - alignment
 - check 3-14
- program flow
 - control 3-5
- relational operator,
 - operand type
 - balancing 3-7
- scalar variable
 - check 3-16
- source file, library
 - compatibility
 - check 3-18
- statement, unlabeled
 - report 3-5
- structure selection
 - operator, type
 - check 3-7
- syntax 3-2
- type cast
 - check 3-9
 - comment printing
 - control 3-9
- type check
 - description 3-7
 - implied
 - assignment 3-7
 - turnoff 3-17
- u option 3-4
- unreachable break
 - report
 - suppression 3-5
- unused argument
 - report
 - suppression 3-3
- unused function
 - check 3-3
- unused variable
 - check 3-3
 - report
 - suppression 3-3
- v option 3-3
 - turn on 3-17
- VARARGS
 - directive 3-17, 3-18
- variable
 - See also lint, unused
 - variable
 - external variable
 - initialization 3-4
 - inner/outer block
 - conflict 3-13
 - set/used
 - information 3-4
 - static variable
 - initialization 3-4
 - x option 3-3
- lint, program checker 3-1
- loader
 - See ld
- lorder command
 - description 1-5
- lpclose
 - See line printer
 - routines
- lpintr
 - See line printer
 - routines
- lpstart

-
- See line printer routines
 - lpwrite
 - See line printer routines

 - M**

 - macros
 - preprocessing 1-4
 - magnetic tape
 - drivers 10-49
 - maintainer
 - See make
 - make
 - .c suffix 4-13
 - .DEFAULT 4-6
 - .f suffix 4-13
 - .IGNORE 4-6
 - .l suffix 4-13
 - .o suffix 4-13
 - .PRECIOUS 4-6
 - .r suffix 4-13
 - .s suffix 4-13
 - .SILENT 4-6
 - .y suffix 4-13
 - .yr suffix 4-13
 - argument quoting 4-8
 - command
 - form 4-2
 - location 4-2
 - print without execution 4-20
 - command argument
 - macro definition 4-7
 - command string
 - hyphen (-) start 4-6
 - command string substitution 4-7
 - d option 4-20
 - dependency line
 - form 4-2
 - dependency line substitution 4-7
 - description file
 - comment
 - convention 4-2
 - macro definition 4-7
 - description filename
 - argument 4-4
 - dollar sign
 - macro invocation 4-7
 - equal sign
 - macro definition 4-7
 - file
 - time, date
 - printing 4-20
 - updating 4-20
 - file generation 4-6
 - file update 4-1
 - hyphen
 - command string start 4-6
 - macro
 - definition 4-7
 - definition
 - override 4-8
 - invocation 4-7
 - substitution 4-6, 4-7
 - value
 - assignment 4-7
 - macro definition
 - analysis 4-8
 - argument 4-4
 - description 4-7
 - medium sized
 - projects 4-1
 - metacharacter 4-2
 - n option 4-20
 - number sign

- description file
 - comment 4-2
- object file
 - suffix 4-13
- option argument
 - use 4-4
- parentheses
 - macro enclosure 4-7
- program
 - maintainer 1-5
- program
 - maintenance 4-1
- semicolon
 - command
 - introduction 4-2
- source file
 - suffixes 4-13
- source grammar
 - suffixes 4-13
- suffixes
 - list 4-13
 - table 4-13
- t option 4-20
- target file
 - pseudo-target
 - files 4-6
 - update 4-20
- target filename
 - argument 4-4
- target name
 - omission 4-4
- touch option 4-20
- transformation rules
 - table 4-13
- troubleshooting 4-20
- make command
 - arguments 4-4
 - syntax 4-4
- make, program
 - maintainer 4-1
- memory mapping
 - device drivers 11-52

- dscalloc() 10-22
- mmudescr() 10-22
- modem interrupts 11-31
- modem routines 11-23
- modes of operation 10-6
- m4
 - description 1-4
- m4, macro processor 9-1

N

- n option
 - lint 3-18
- naming conventions
 - device driver
 - routines 10-33
- notational
 - conventions 1-2

O

- object files 2-5
- operation modes
 - system mode 10-6
 - user mode 10-6
- out() routine 10-12
- outb() routine 10-12

P

- p option
 - lint 3-18
- panic() routine 10-30
- PC XENIX operating
 - system 1-2
- PC XENIX to DOS 2-1

- assembler language
 - files 2-4
 - compiling DOS file 2-3
 - creating libraries 2-7
 - creating source files 2-2
 - debugging DOS program 2-5
 - linking 2-5
 - transferring programs 2-6
- physio() routine 10-57
- pipng
 - See SCCS
- precedence 8-27
- printf() routine 10-29
- printing error messages 10-42
- processes
 - system 10-6
 - u_ area 10-7
 - user 10-6
- program development 1-2
- program file 6-2
- program maintainer
 - See make
- putc() routine 10-44
- putcb() routine 10-39
- putchar() routine 10-29

Q

- quoting arguments 9-5

R

- ranlib
 - description 1-5
- registers
 - accessing 10-12
 - controlling 10-12
 - defining 11-11
- rm command
 - See SCCS

S

- sample device
 - drivers 11-1
 - lpopen() 11-7
- SCCS
 - See also SCCS, z-file
 - %M% keyword > g-file
 - line precedence 5-41
 - @(#) string
 - file information, search 5-43
 - a option
 - login name addition use 5-32
 - admin command
 - file
 - administration 5-35
 - file checking use 5-35
 - file creation 5-7
 - use
 - authorization 5-8
 - administrator
 - description 5-6
 - argument
 - minus sign use 5-5
 - types designated 5-5

-
- branch delta
 - retrieval 5-15
 - branch number
 - description 5-3
 - cdc command
 - commentary
 - change 5-24
 - ceiling flag
 - protection 5-33, 5-34
 - checksum
 - file corruption
 - determination 5-35
 - command
 - See also SCCS,
 - argument
 - execution
 - control 5-5
 - explanation 5-37
 - comments
 - change
 - procedure 5-24
 - omission, effect 5-38
 - corrupted file
 - determination 5-35
 - processing
 - restrictions 5-35
 - restoration 5-36
 - d-file
 - temporary g-file 5-5
 - d flag
 - default
 - specification 5-21
 - flags deletion 5-22
 - d option
 - data specification
 - provision 5-28
 - flag removal 5-22
 - data keyword
 - data specification
 - component 5-28
 - replacement 5-28
 - data specification
 - description 5-28
 - delta
 - branch delta 5-15
 - defined 5-1, 5-2
 - exclusion 5-39
 - inclusion 5-39
 - interference 5-39
 - latest release
 - retrieval 5-16
 - level number 5-2
 - name 5-2
 - printing 5-29, 5-41
 - range printing 5-29
 - release number 5-2
 - removal 5-42
 - delta command
 - comments
 - prompt 5-11
 - file change
 - procedure 5-11
 - g-file removal 5-17
 - p-file reading 5-11
 - delta table
 - delta removal,
 - effect 5-42
 - description 5-23
 - description 1-5
 - descriptive text
 - adding 5-26
 - modification 5-26
 - removal 5-27
 - diagnostic output
 - p option effect 5-16
 - diagnostics
 - code as help
 - argument 5-17
 - form 5-17
 - directory
 - file argument
 - application 5-5
 - x-file location 5-4
 - directory use 5-2
 - e option

delta range
 printing 5-29
file editing use 5-9
login name
 removal 5-33
error message
 code use 5-17
 form 5-17
exclamation point
 MR deletion
 use 5-26
-f option
 flag setting 5-20
 flag, value
 setting 5-21
file
 See also SCCS,
 descriptive text
 See also SCCS, g-file
 See also SCCS, p-file
 See also SCCS, x-file
 administration 5-35
 change
 identification 5-41
 change
 procedure 5-11
 change, major 5-14
 changes 5-2
 checking
 procedure 5-35
 comparison 5-43
 composition 5-2,
 5-23
 corrupted file 5-35
 creation 5-7
 data keyword 5-28
 descriptive text
 description 5-23
 editing, -e option
 use 5-9
 grouping 5-2
 identifying
 information 5-43
 link 5-2
 lock file 5-4
 modification 5-26
 multiple concurrent
 edits 5-30
 name 5-2
 name arbitrary 5-16
 name, s use 5-7
 printing 5-28
 protection
 methods 5-32
 removal 5-7
 retrieval 5-8
 versions 5-2
file argument
 description 5-5
 processing 5-5
file creation
 comment line
 generation 5-38
 commentary 5-38
 comments omission,
 effect 5-38
 level number 5-37
 release number 5-37
file protection 5-32
flags
 deletion 5-22
 modification 5-21
 setting 5-20
 setting, value
 setting 5-21
 use 5-22
floor flag
 protection 5-33
g-file
 creation 5-4
 creation date, time
 recording 5-18
 description 5-4

line
 identification 5-41
line, %M% keyword
 value 5-41
ownership 5-4
regeneration 5-36
removal, delta
 command use 5-17
temporary 5-5

-g option
 output
 suppression 5-42
 p-file
 regeneration 5-36

get command
 concurrent editing,
 directory use 5-30
 delta inclusion,
 exclusion
 check 5-39
 -e option use 5-9
 file retrieval 5-8
 filename
 creation 5-8
 g-file creation 5-4
 message 5-8
 release number
 change 5-14

-h option
 file audit use 5-35

help command
 argument 5-17
 code use 5-17
 use 5-37

i flag
 file creation,
 effect 5-20
 keyword message,
 error
 treatment 5-21

-i option
 delta inclusion list
 use 5-39

ID keyword 5-18

identification
 string 5-2

j flag
 multiple concurrent
 edits
 specification 5-30

-k option
 g-file
 regeneration 5-36

keyword
 data 5-28
 format 5-18
 missing 5-21
 use 5-18

l-file
 creation 5-40

-l option
 delta range
 printing 5-29
 l-file creation 5-40

level number
 delta component 5-2
 new file 5-37
 omission, file
 retrieval,
 effect 5-13

link
 number
 restriction 5-2

lock flag
 edit protection 5-34

-m option
 effective when 5-25
 file change
 identification 5-41
 new file
 creation 5-38

minus sign
 argument use 5-5
 option argument
 use 5-5

mode
 g-file 5-4

MR
 commentary
 supply 5-23
 deletion 5-26
 new file
 creation 5-38

multiple users 5-6

-n option
 %M% keyword
 value use 5-41

 g-file
 preservation 5-17
 pipeline use 5-41

option argument
 description 5-5
 processing order 5-5

output
 data
 specification 5-28
 suppression, -g
 option 5-42
 suppression, -s
 option 5-38
 write to standard
 output 5-16

p-file
 contents 5-4, 5-11
 creation 5-4
 delta command
 reading 5-11
 naming 5-4
 ownership 5-4
 permissions 5-4
 regeneration 5-36
 update 5-4
 updating 5-5

-p option
 delta printing 5-41
 output effect 5-16

percentage sign

keyword
 enclosure 5-18

pipng 5-38
 -n option use 5-41

prs command
 file printing 5-28

purpose 5-1

q-file
 use 5-5

R
 delta removal
 check 5-42

-r option
 delta creation
 use 5-31
 delta printing
 use 5-29
 file retrieval 5-12
 release number
 specification 5-37

release
 protection 5-33

release number
 change 5-3
 change
 procedure 5-14
 delta component 5-2
 new file 5-37

-r option,
 specification 5-37

rm command
 file removal 5-7

rm del command
 delta removal 5-42

-s option
 output
 suppression 5-38

sccsdiff command
 file comparison 5-43

sequence number
 description 5-3

SIDs

- components 5-2
- delta printing
 - use 5-29
- t option
 - delta retrieval 5-16
 - file
 - initialization 5-27
 - file
 - modification 5-27
- tab character
 - n option,
 - designation 5-41
- user list
 - empty by
 - default 5-32
 - login name
 - addition 5-32
 - login name
 - removal 5-33
 - protection
 - feature 5-32
- user name
 - list 5-32
- v flag
 - new file use 5-21
- what command
 - file
 - information 5-43
- write permission
 - delta removal 5-42
- x-file
 - directory,
 - location 5-4
 - naming
 - procedure 5-4
 - permissions 5-4
 - temporary file
 - copy 5-4
 - use 5-4
- x option
 - delta exclusion list
 - use 5-39
- XENIX command
 - use precaution 5-36
 - y option
 - comments prompt
 - response 5-23
 - new file
 - creation 5-38
 - z-file
 - lock file use 5-4
 - ownership 5-4
 - permissions 5-4
 - z key
 - file audit use 5-36
- SCCS,Source Code Control
 - System 5-1
- sharing interrupt
 - vectors 10-74
- signal 6-23
- signal() routine 10-30
- sleep() routine 10-16
- software development
 - described 1-2
- Source Code Control
 - System
 - See SCCS
- source files 2-2
 - creating 1-3
 - DOS source files 2-3
- spl routines 10-14
- stack
 - u_ area 10-7
- strings 9-16
- strip
 - description 1-6
- sum
 - description 1-6
- suser() routine 10-30
- suspending
 - processing 10-16
- symbol
 - name list 1-6
 - removal 1-6
- sync

description 1-6
syscmd 9-13
system calls
 ioctl() routine 10-26
system
 configuration 10-68
system mode stack 10-7
system processes 10-6

T

tags file
 creation 1-6
task-time processing 10-8
tdclose
 See terminal device routines
tdintr
 See terminal device routines
tdioctl
 See terminal device routines
tdmint
 See terminal device routines
tdmodem
 See terminal device routines
tdopen
 See terminal device routines
tdparam
 See terminal device routines
tdproc
 See terminal device routines

tread
 See terminal device routines
tdrprint
 See terminal device routines
tdwrite
 See terminal device routines
tdxint
 See terminal device routines
terminal
 device driver
 sample 11-10
terminal device routines
 tdclose() 11-19
 tdintr() 11-25
 tdioctl() 11-31
 tdmint() 11-31
 tdmodem() 11-23
 tdopen() 11-17
 tdparam() 11-17, 11-21
 tdproc() 11-33
 tread() 11-19
 tdrprint() 11-29
 tdwrite() 11-19
 tdxint() 11-27
text editor
 creating programs 1-3
timeout() routine 10-17
token, input
 See yacc, token
touch option 4-20
tsort
 description 1-5
ttinit() routine 11-17
tty routines 10-23

U

-u option
 lint 3-4
u_ area 10-7
user processes 10-6

V

-v option
 lint 3-3, 3-17
vi, the screen-oriented text
 editor 1-3

W

wakeup() routine 10-16
wrap up
 See lex, yywrap()

X

-x option
 lint 3-3
XENIX file
 identifying
 information 5-43

Y

yacc
 See also yacc, desk
 calculator
 See also yacc, parser
 %prec keyword 8-28
 accept simulation 8-40
 action
 See also yacc, parser
 conflict source 8-23
 defined 8-9
 error rules 8-31
 form 8-62
 global flag
 setting 8-39
 input style 8-36
 invocation 8-2
 location 8-10
 nonterminating 8-10
 return value 8-42
 statement 8-9, 8-11
 value in enclosing
 rules, access 8-41
 0, negative
 number 8-41
 ampersand
 and operator 8-44
 arithmetic expression
 See also yacc,
 precedence
 desk calculator 8-44
 parsing 8-27
 associativity
 arithmetic
 expression
 parsing 8-27
 grammar rule
 association 8-29
 recordation 8-29

- token
 - attachment 8-27
- asterisk
 - multiplication operator 8-44
- backslash
 - escape character 8-6
 - percentage sign substitution 8-62
- binary operator
 - precedence 8-28
- blank character
 - restrictions 8-5
- braces
 - action 8-11
 - action statement enclosure 8-9
 - action, dropping 8-62
 - header file enclosure 8-43
- colon
 - identifier, effect 8-48
 - punctuation 8-6
- comments
 - location 8-5
- conflict
 - See also yacc, associativity
 - See also yacc, precedence
 - disambiguating rules 8-22, 8-23
 - message 8-25
 - reduce conflict 8-29
 - reduce reduce conflict 8-29
 - reduce/reduce conflict 8-22
 - resolution, not counted 8-29
 - shift reduce conflict 8-29
 - shift/reduce conflict 8-22, 8-24
 - source 8-23
- declaration
 - specification file component 8-5
- declaration section
 - header file 8-43
- description 1-4
- desk calculator
 - advanced features 8-52
 - error recovery 8-52
 - floating-point interval 8-52
 - scalar conversion 8-53
- desk calculator specifications 8-44
- dflag 8-39
- disambiguating rule 8-22
- disambiguating rules 8-23
- dollar sign
 - action significance 8-9
- empty rule 8-38
- enclosing rules, access 8-41
- endmarker
 - lookahead token 8-17
 - parser input end 8-8
 - representation 8-8
 - token number 8-13
- environment 8-34
- error

- handling 8-31
- nonassociating
 - implication 8-29
- parser restart 8-31
- simulation 8-40
- yerror
 - statement 8-32
- error token
 - parser restart 8-31
- escape characters 8-6
- external integer
 - variable 8-34
- flag
 - global flag 8-39
- global flag
 - lexical analysis 8-39
- grammar rules 8-2
 - advanced
 - features 8-52
 - ambiguity 8-21
 - associativity
 - association 8-29
 - C code
 - location 8-62
 - empty rule 8-38
 - error token 8-31
 - format 8-6
 - input style 8-36
 - left recursion 8-37
 - left side
 - repetition 8-7
 - names 8-6
 - numbers 8-25
 - precedence
 - association 8-29
 - reduce action 8-15
 - reduction 8-16
 - rewrite 8-23
 - right recursion 8-37
 - specification file
 - component 8-5
 - value 8-10
 - zero character
 - avoidance 8-6
- header file, union
 - declaration 8-43
- historical features 8-62
- identifier
 - input syntax 8-48
- if-else rule 8-23
- if-then-else
 - construction 8-23
- input
 - style 8-36
 - syntax 8-48
- input error
 - detection 8-3
- key endmarker token
 - marker 8-13
- keyword 8-27
 - reservation 8-40
 - union member name
 - association 8-43
- left association 8-21
- left associative
 - reduce
 - implication 8-29
- left keyword 8-27
 - union member name
 - association 8-43
- left recursion 8-37
 - value type 8-43
- left token
 - synonym 8-62
- lex
 - interface 7-2
 - lexical analyzer
 - construction 8-13
- lexical analyzer
 - context
 - dependency 8-39
 - defined 8-1, 8-12
 - endmarker
 - return 8-8

floating-point
 constants 8-53
function 8-2
global flag
 examination 8-39
identifier
 analysis 8-48
 lex 8-13
 return value 8-42
 scope 8-11
 specification file
 component 8-5
 terminal symbol 8-2
 token number
 agreement 8-12
lexical tie-in 8-39
library 8-34
literal
 defined 8-6
 delimiting 8-62
 length 8-62
lookahead token 8-14
 clearing 8-33
 error rules 8-31
LR grammar 8-48
ly argument, library
 access 8-34
main program 8-34
minus sign
 minus operator 8-44
names
 composition 8-6
 length 8-6
 reference 8-5
 token name 8-7
newline character
 restrictions 8-5
nonassoc keyword 8-27
 union member name
 association 8-43
nonassoc token
 synonyms 8-62
nonassociating
 error
 implication 8-29
nonterminal
 union member name
 association 8-43
nonterminal name
 input style 8-36
 representation 8-6
nonterminal
 symbol 8-2
 empty string
 match 8-7
 location 8-8
 name 8-5
 start symbol 8-8
octal interger
 beginning 8-44
option
 output file 8-17
parser
 See also yacc,
 conflict
 See also yacc, error
 accept action 8-17
 accept
 simulation 8-40
 actions 8-14
 arithmetic
 expression 8-27
 creation 8-27
 defined 8-1
 description 8-14
 error action 8-17
 goto action 8-16
 initial state 8-19
 input end 8-8
 lookahead
 token 8-15
 movement 8-14
 names, yy
 prefix 8-11

- nonterminal
 - symbol 8-2
- production
 - failure 8-3
- reduce action 8-15
- restart 8-31
- shift action 8-15
- start symbol
 - recognition 8-8
- token number
 - agreement 8-12
- percentage sign
 - action 8-11
 - header file
 - enclosure 8-43
 - mod operator 8-44
- precedence
 - keyword 8-27
- specification file
 - section
 - separator 8-5
 - substitution 8-62
- plus sign
 - + operator 8-44
- prec
 - synonym 8-62
- precedence
 - binary
 - operator 8-28
 - change 8-28
 - grammar rule
 - association 8-29
 - keyword 8-27
 - parsing
 - function 8-27
 - recordation 8-29
 - token
 - attachment 8-27
 - unary operator 8-28
- program
 - specification file
 - component 8-5
- punctuation 8-6
- quotation marks, double
 - literal
 - delimiting 8-62
- quotation marks, single
 - literal enclosure 8-6
- reduce command
 - number
 - reference 8-25
- reduce conflict 8-29
- reduce reduce
 - conflict 8-29
- reduce/reduce
 - conflict 8-22
- reduction conflict 8-22
- reserved words 8-40
- right association 8-21
- right associative
 - shift
 - implication 8-29
- right keyword 8-27
- union member name
 - association 8-43
- right recursion 8-37
- right token
 - synonym 8-62
- semicolon
 - input style 8-36
 - punctuation 8-6
- shift command
 - number
 - reference 8-25
- shift reduce
 - conflict 8-29
- shift/reduce
 - conflict 8-22, 8-24
- simple-if rule 8-23
- slash
 - division
 - operator 8-44
- specification file
 - contents 8-5

- lexical analyzer
 - inclusion 8-5
 - sections
 - separator 8-5
- specification files 8-3
- start symbol
 - description 8-8
 - location 8-8
- symbol synonyms 8-62
- tab character
 - restrictions 8-5
- terminal symbol 8-2
- token
 - See also yacc, error token
 - associativity 8-27
 - defined 8-1
 - names 8-5
 - organization 8-2
 - precedence 8-27
 - synonym 8-62
- token keyword
 - union member name
 - association 8-43
- token name
 - declaration 8-7
 - input style 8-36
- token names 8-13
- token number 8-12
 - agreement 8-12
 - assignment 8-13
 - endmarker 8-13
- type keyword 8-43
- unary operator
 - precedence 8-28
- underscore sign
 - parser 8-19
- union
 - copy 8-42
 - declaration 8-42
 - header file 8-42
 - name
 - association 8-43
- unreachable
 - statement 3-5
- value
 - typing 8-42
 - union 8-42
- value stack 8-42
 - declaration 8-42
 - floating-point scalars, integers 8-52
- vertical bar
 - grammar rule
 - repetition 8-7
 - input style 8-36
 - or operator 8-44
- y.output file 8-17
- y.tab.c file 8-34
- y.tab.h file 8-42
- YYACCEPT 8-40
- yychar 8-34
- yyclearin
 - statement 8-33
- yydebug 8-34
- yyerror statement 8-32
- yyerror 8-34, 8-52
- yylex 8-34
- yyparse 8-34
 - YYACCEPT
 - effect 8-40
 - YYSTYPE 8-43
- 0 character
 - grammar rules, avoidance 8-6
- yacc, program
 - generator 8-1

Notes:

Notes:



The Personal
Computer
Programming
Family

Reader's Comment Form

**XENIX Operating
System
Application
Development
Guide**

SV21-8078

Your comments assist us in improving our publication; they are an important part of the input used for revisions.

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for questions regarding setup, operation, or program support or for requests for additional publications. Instead, contact your authorized IBM Personal Computer dealer in your area.

Comments:

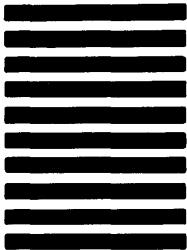


NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM CORPORATION
DEPARTMENT 997
11400 BURNET ROAD
AUSTIN, TEXAS 78758



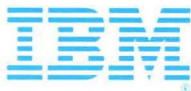
.....
Fold here

© IBM Corporation 1986
All rights reserved.

International Business
Machines Corporation
P.O. Box 1328-W
Boca Raton,
Florida 33429-1328

Printed in the
United States of America

59X8634

The IBM logo, consisting of the letters "IBM" in a stylized, blue, eight-striped font. A small registered trademark symbol (®) is located at the bottom right of the logo.

Software required:

IBM Personal Computer
XENIX™ Operating System
Version 2.00

Software included:

Three 1.2MB diskettes

System requirements:

IBM Monochrome, Color,
Enhanced Graphics, or
Professional Graphics
Display or equivalent
(with appropriate adapter)

IBM Personal Computer AT®
512KB RAM memory
IBM 20MB or 30MB fixed disk
IBM 1.2MB diskette drive

Note:
XENIX is a trademark of Microsoft
Corporation

© IBM Corporation 1986
All rights reserved.

International Business
Machines Corporation
P.O. Box 1328-W
Boca Raton,
Florida 33429-1328



Printed in the
United States of America

You should carefully read the following terms and conditions before opening this package. Opening this package indicates your acceptance of these terms and conditions. If you do not agree with them, you should promptly return the package unopened and your money will be refunded.

IBM provides this program and licenses its use in the United States and Puerto Rico. Title to the media on which this copy of the program is recorded and to the enclosed copy of the documentation is transferred to you, but title to the copy of the program is retained by IBM or its supplier, as applicable. You assume responsibility for the selection of the program to achieve your intended results, and for the installation, use and results obtained from the program.

License

You may:

- a. use the program on only one machine at any one time except as otherwise specified by IBM in the enclosed Program Specifications (available for your inspection prior to your acceptance of this Agreement);
 - b. copy the program into machine-readable or printed form for backup or modification purposes only in support of such use. (Certain programs, however, may include mechanisms to limit or inhibit copying. They are marked "copy protected");
 - c. modify the program and/or merge it into another program for your use on the single machine. (Any portion of this program merged into another program will continue to be subject to the terms and conditions of this Agreement.); and,
 - d. transfer the program with a copy of this Agreement to another party only if the other party agrees to accept from IBM the terms and conditions of this Agreement. If you transfer the program, you must at the same time either transfer all copies whether in printed or machine-readable form to the same party or destroy any copies not transferred; this includes all modifications and portions of the program contained or merged into other programs. IBM will grant a license to such other party under this Agreement and the other party will accept such license by its initial use of the program. If you transfer possession of any copy, modification or merged portion of the program, in whole or in part, to another party, your license is automatically terminated.
- You must reproduce and include the copyright notice on any copy, modification, or portion merged into another program.

You may not reverse assemble or reverse compile the program without IBM's prior written consent.

You may not use, copy, modify, or transfer the program, or any copy, modification or merged portion, in whole or in part, except as expressly provided for in this Agreement.

You may not sublicense, assign, rent or lease this program.

Term

The license is effective until terminated. You may terminate it at any other time by destroying the program together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy the program together with all copies, modifications and merged portions in any form.

Limited Warranty and Disclaimer of Warranty

IBM warrants the media on which the program is furnished to be free from defects in materials and workmanship under normal use for a period of 90 days from the date of IBM's delivery to you as evidenced by a copy of your receipt.

IBM warrants that each program which is designated by IBM as warranted in its Program Specifications, supplied with the program, will conform to such specifications provided that the program is properly used on the IBM machine for which it was designed. If you believe that there is a defect in a warranted program such that it does not meet its specifications, you must notify IBM within the warranty period set forth in the Program Specifications.

All other programs are provided "as is" without warranty of any kind, either express or implied. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you (and not IBM or an IBM authorized representative) assume the entire cost of all necessary servicing, repair or correction.

IBM does not warrant that the functions contained in any program will meet your requirements or that the operation of the program will be uninterrupted or error free or that all program defects will be corrected.

The foregoing warranties are in lieu of all other warranties, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Some states do not allow the exclusion of implied warranties, so the above exclusion may not apply to you. This warranty gives you specific legal rights and you may also have other rights which vary from state to state.

Limitations of Remedies

IBM's entire liability and your exclusive remedy shall be as follows:

1. With respect to defective media during the warranty period:

- a. IBM will replace media not meeting IBM's "Limited Warranty" which is returned to IBM or an IBM authorized representative with a copy of your receipt.
- b. In the alternative, if IBM or such IBM authorized representative is unable to deliver replacement media which is free of defects in materials and workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

2. With respect to warranted programs, in all situations involving performance or nonperformance during the warranty period, your remedy is (a) the correction by IBM of program defects, or (b) if, after repeated efforts, IBM is unable to make the program operate as warranted, you shall be entitled to a refund of the money paid or to recover actual damages to the limits set forth in this section.

For any other claim concerning performance or nonperformance by IBM pursuant to, or in any other way related to, the warranted programs under this Agreement, you shall be entitled to recover actual damages to the limits set forth in this section.

IBM's liability to you for actual damages for any cause whatsoever, and regardless of the form of action, shall be limited to the greater of \$5,000 or the money paid for the program that caused the damages or that is the subject matter of, or is directly related to, the cause of action.

In no event will IBM be liable to you for any lost profits, lost savings or other incidental or consequential damages arising out of the use of or inability to use such program even if IBM or an IBM authorized representative has been advised of the possibility of such damages, or for any claim by any other party.

Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to you.

Service

Service from IBM, if any, will be described in Program Specifications or in the statement of service, supplied with the program, if there are no Program Specifications.

IBM may also offer separate services under separate agreement for a fee.

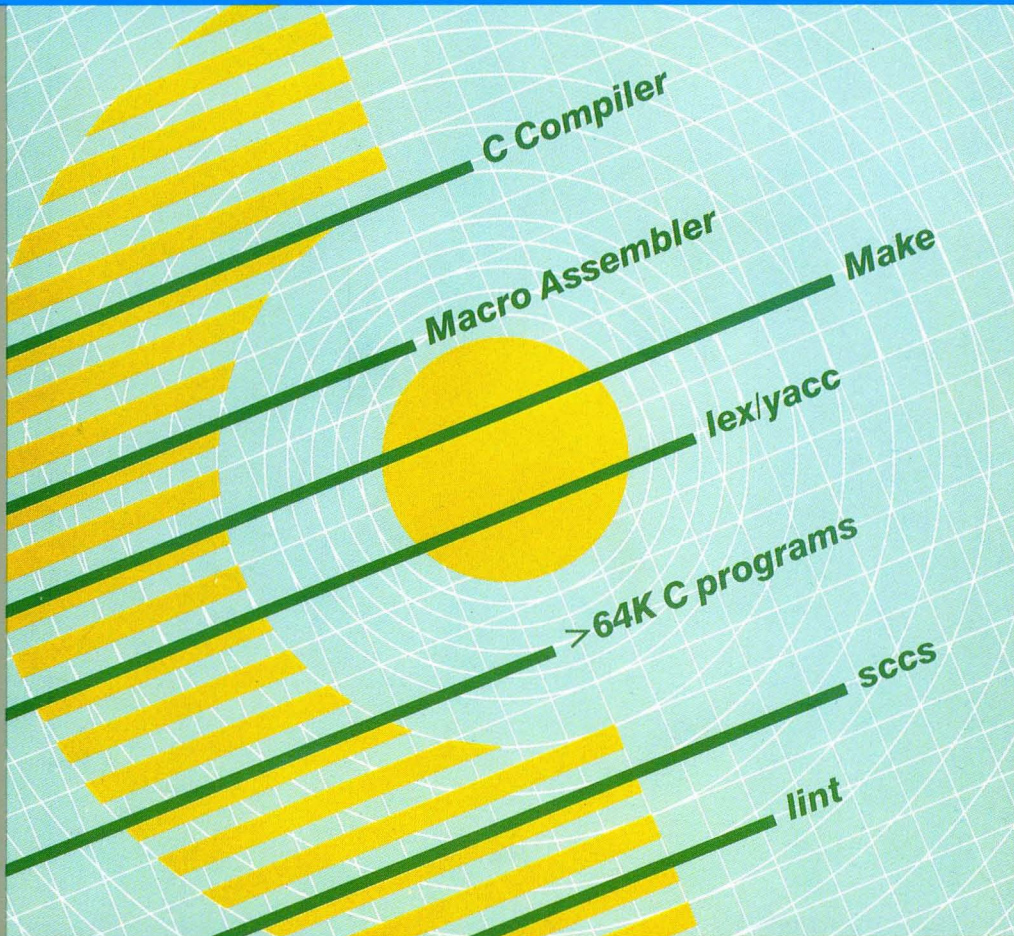
General

Any attempt to sublicense, assign, rent or lease, or, except as expressly provided for in this Agreement, to transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be construed under the Uniform Commercial Code of the State of New York.

IBM Personal Computer XENIX™ Software Development System Version 2.00

Programming Family



IBM

Personal
Computer
Software

Software development tools, including language translators, source code management tools, a C compiler, Macro Assembler, a debug facility, and a linker for combining modules into finished programs. The C compiler generates code for DOS or the IBM Personal Computer XENIX™ Operating System.