Software Development Guide

IBM Personal Computer XENIX™ Software Development System

Programming Family

IBM

Personal
Computer
Software

6138694

Software Development Guide

IBM Personal Computer XENIX™ Software Development System

Programming Family

Personal Computer Software

First Edition (December 1984)

IBM Personal Computer XENIX Library Overview

The XENIX[1] System has three available products. They are the:

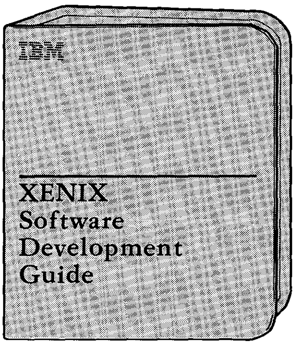
- Operating System
- Software Development System
- Text Formatting System

The following pages outline the XENIX Software Development System library.

[1] XENIX is a trademark of Microsoft Corporation.

XENIX Software Development System

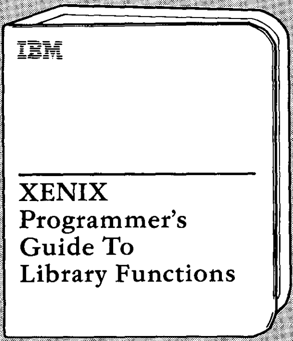
For C Language Users



- Creating language programs
- Invoking the C Compiler
- Program checkers, maintainers, and debuggers
- Using S-Files
- The C-Shell

A guide to the available programming tools in the XENIX environment.

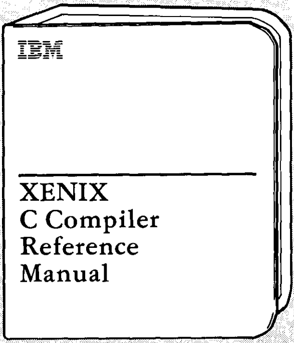
For All Users



- Using stream functions
- Screen processing
- Process controls
- Creating and using pipes
- Using signals and system resources

A reference to system calls, subroutines, and file formats. Use with the XENIX Software Command Reference.

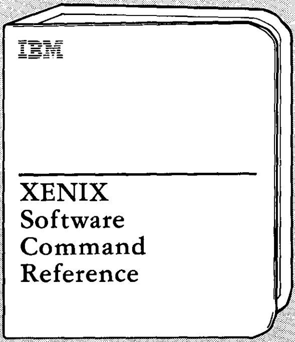
For Experienced Language Users



- Elements of the C programming language
- Preprocessor Directives
- Declarations
- Expressions and Assignments
- Description of functions and statements

A reference to the C programming language. Notational conventions are described throughout the manual.

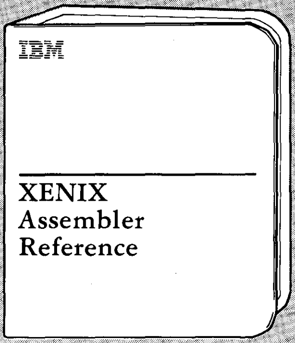
For All Users



- Software Development commands (CP)
- Command definition and syntax
- System calls and subroutines (S)
- System call and library function cross reference

A reference to Software Development System commands. Describes system services in the Operating System kernel.

For Assembler Users



- Assembly Language format
- Operands and Operators
- Directives
- Segment usage
- Machine instructions
- Assembler messages

A reference for programmers who use the IBM Personal Computer XENIX Assembler.

About This Book

This guide explains how to use the XENIX Software Development system to create and maintain C language and assembly language programs. This guide is intended for programmers who are familiar with the C programming language and with the XENIX system.

C language programmers should read Chapters 2, 3, 4, and 7 for an explanation of how to compile and debug C language programs. Programmers who wish to automate the compilation process of their programs should read Chapter 5 for an explanation of the **make** program. Programmers who wish to organize and maintain multiple versions of their programs should read Chapter 6 for an explanation of the SCCS commands.

Special project programmers who need a convenient way to produce lexical analyzers and parsers should read Chapters 8 and 9 for explanations of the **lex** and **yacc** program generators.

This book is organized as follows:

Chapter 1. Introduction
Introduces the XENIX software development programs provided with this package.

Chapter 2. CC: A C Compiler
Explains how to compile C language programs using the **cc** command.

Chapter 3. XENIX to DOS: A Cross Development System
Provides information on how to create programs that run under DOS. The DOS cross development system lets you create, compile, and link DOS programs on the XENIX system and transfer these programs to the DOS system.

Chapter 4. The lint Program: A C Program Checker
Explains how to check C language programs for syntactical and semantical correctness using the C program checker **lint**.

Chapter 5. A Program Maintainer: make
Explains how to automate the development of a program or other project using the **make** program.

Chapter 6. SCCS: A Source Code Control System
Explains how to control and maintain all versions of a project's source files using the Source Code Control System (SCCS) commands.

Chapter 7. The adb Program Debugger
Explains how to debug C and assembly language programs using the XENIX debugger **adb**.

Chapter 8. The lex Program: A Lexical Analyzer
Explains how to create lexical analyzers using the program generator **lex**.

Chapter 9. The yacc Program Generator: A Compiler–Compiler
Explains how to create parsers using the program generator **yacc**.

Chapter 10. The C Shell
Explains how to use the C shell, a command interpreter that provides greater flexibility and more power than the standard XENIX shell, **sh**.

Appendix A. C Language Portability
Explains how to write C language programs that can be compiled on other XENIX systems.

Appendix B. M4: A Macro Processor
Explains how to use to create and process macros using the **m4** macro processor.

Appendix C. XENIX Device Driver
Explains how to write device drivers for XENIX systems.

Appendix D. Linker Error Messages
Lists all linker error messages alphabetically. Explains each error message.

Related XENIX Personal Computer Publications

- IBM Personal Computer *XENIX Programmer's Guide to Library Functions*
- IBM Personal Computer *XENIX C Compiler Reference Manual*
- IBM Personal Computer *XENIX Software Command Reference*
- IBM Personal Computer *XENIX Assembler Reference*
- IBM Personal Computer *XENIX Installation Guide*
- IBM Personal Computer *XENIX Visual Shell*
- IBM Personal Computer *XENIX System Administration*
- IBM Personal Computer *XENIX Basic Operations Guide*
- IBM Personal Computer *XENIX Command Reference*

Contents

Chapter 1. Introduction

Contents

Overview

The IBM Personal Computer XENIX Software Development System provides a broad spectrum of programs and commands to help you design and develop applications and system software. These programs and commands enable you to create C and assembly language programs for execution on the XENIX system. They also let you debug these programs, "automate" their creation, and maintain different versions of the programs you develop. The **make** program saves many steps because you don't have to do each one manually.

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

[]	Brackets indicate an optional command argument.
...	Ellipses indicate that the preceding argument can be repeated one or more times.
bold	Boldface characters indicate a command or program name.
italics	Italic characters indicate a placeholder for a command argument. When typing a command, a placeholder must be replaced with an appropriate filename, number, or option.

The following sections introduce the programs and commands of the IBM XENIX Software Development System, and explain the steps you can take to develop programs for the IBM XENIX system. Most of the programs and commands in these introductory sections are fully explained later in this guide. Some commands mentioned here are part of the IBM Personal Computer XENIX Timesharing System. These are explained in the IBM Personal Computer *XENIX Basic Operations Guide* and the IBM Personal Computer *XENIX System Administration.*

Creating C Language Programs

All C language programs start as a collection of C program statements in a source file. The XENIX system provides a number of text editors that let you create source files easily and efficiently. The most convenient editor is the screen-oriented editor **vi**. The **vi** screen editor provides many editing commands that let you easily insert, replace, move, and search for text. All commands can be invoked from command keys or from a command line. The **vi** editor also has a variety of options that let you modify its operation.

Once a C language program has been written to a source file, you can create an executable program by using the **cc** command. The **cc** command invokes the XENIX C compiler which compiles the source file. This command also invokes other XENIX programs to prepare the compiled program for execution such as the linker and the assembler.

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

If you wish to check a program before compiling it, you can use **lint**, the XENIX C program checker. The **lint** program checks the content and construction of C language programs for syntactical and logical errors. It also enforces a strict set of guidelines for proper C programming style. The **lint** program is normally used in the early stages of program development to check for illegal and improper usage of the C language.

Another way to check a program is with **cb**, the XENIX C program beautifier. The **cb** program improves readability of C programs, making detection of logical errors easier.

Creating Other Programs

The C programming language can meet the needs of most programming projects. In cases where finer control of execution is required, you can create assembly language programs using the XENIX assembler **as**. The **as** program assembles source files and produces relocatable object files that can be linked to your C language programs with **ld**. The **ld** program is the XENIX linker. It links relocatable object files created by the C compiler or assembler to produce executable programs. The **cc** command automatically invokes the linker and the assembler, so use of either **as** or **ld** is optional.

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

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

Creating and Maintaining Libraries

You can create libraries of useful C and assembly language functions and programs using the **ar** and **ranlib** programs. The

XENIX archiver, **ar**, creates libraries of relocatable object files. The XENIX random library generator, **ranlib**, converts archive libraries to random libraries and places a table of contents at the front of each library.

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

Maintaining Program Source Files

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

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

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

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

Other commands let you examine object and executable binary files. The **nm** command prints the list of symbol names in a program. The **hd** command performs a hexadecimal dump of given files, printing files in a variety of formats, one of which is hexadecimal. The **size** command reports the size of an object file. The **strings** command finds and prints readable text (strings) in an object or other binary file. The **strip** command removes symbols and relocation bits from executable files. 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. The **xstr** command extracts strings from C programs to implement shared strings.

Creating Programs with Shell Commands

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

The **csh** command invokes the C-shell, a XENIX command interpreter. The C-shell interprets and executes commands taken from the keyboard or from a command file. It has a C-like syntax that makes programming in this command language easy. It also has an "aliasing" facility and a command history mechanism. For a discussion of aliasing, refer to the section “Using Aliases” in “Chapter 10. The C Shell.”

Chapter 2. CC: The C Compiler

Contents

Introduction

This chapter explains how to use the **cc** command. In particular, it explains how to:

- Compile C language source files
- Choose a memory model for a program
- Use object files and libraries with a program
- Create smaller and faster programs
- Prepare C programs for debugging
- Control the C preprocessor

It also describes the error and warning messages generated by the C compiler, and explains how to use advanced features of the **cc** command to make customized programs.

This chapter assumes that you are familiar with the C programming language, and that you can create C program source files using a IBM Personal Computer XENIX text editor. For a description of the C language, see the IBM Personal Computer *XENIX Software Command Reference* or the *C Compiler Reference.*

Invoking the C Compiler

The **cc** command has the form

```
cc [ option ] . . .  filename  . . .
```

where *option* is a command option, and *filename* is the name of a C language source file, an assembly language source file, an object file, or an archive library. You can give more than one option or filename, if desired, but must separate each item with one or more spaces.

The **cc** command options let you control and modify the tasks performed by the command. For example, you can direct **cc** to perform optimization or create an assembly listing file. The options also let you specify additional information about the compilation, such as which program libraries to examine and what the name of the executable file should be. Many options are described in the following sections. For a complete description of all options, see *cc (CP)* in the IBM Personal Computer *XENIX Software Command Reference.*

Creating Programs From C Source Files

The **cc** command creates executable programs from C language source files. A file's contents are identified by the filename extension. C source files must have the extension ".c".

The **cc** command can create executable programs only from source files that make up a complete C program. In XENIX system, a complete program must have one (and only one) function named "main". This function becomes the entry point for program execution. The "main" function can call other functions as long as they are defined within the program or are part of the C standard library. The standard C library is described in the IBM Personal Computer *XENIX Programmer's Guide to Library Functions.*

Compiling a C Source File

You can compile a C source file by giving the name of the file when you invoke the **cc** command. The command compiles the statements in the file, then copies the executable program to the default output file *a.out.*

To compile a source program, type:

```
cc filename
```

where *filename* is the name of the file containing the program. The program must be complete, that is, it must contain a “main” program function. It can also contain calls to functions explicitly defined by the program or by the standard C library.

For example, assume that the following program is stored in the file named *main.c*.

```
#include <stdio.h>

main ()
{
        int x,y;

        scanf("%d  %d", &x, &y);
        printf("%d\n", x+y);
}
```

To compile this program, type:

```
cc main.c
```

The command first invokes the C preprocessor, which adds the statements in the file */usr/include/stdio.h* to the beginning of the program. It then compiles these statements and the rest of the program statements. Next, the command links the program with the standard C library, which contains the object files for the *scanf* and *printf* functions. Finally, it copies the program to the file *a.out*.

You can execute the new program by typing

```
a.out
```

The program waits until you enter two numbers, then prints their sum. For example, if you type “3 5” the program displays “8”.

Compiling Several Source Files

Large source programs are often split into several files to make them easier to understand, update and edit. You can compile such a program by giving the names of all the files belonging to the program when you invoke the **cc** command. The command reads and compiles each file in turn, then links all object files together, and copies the new program to the file *a.out.*

To compile several source files, type:

```
cc filename  . . .
```

where each *filename* is separated from the next by at least one space. One of these files (and only one) must contain a "main" function. The others can contain functions called by this "main" function or by other functions in the program. The files must not contain calls to functions not explicitly defined by the program or by the standard C library.

For example, suppose the following main program function is stored in the file *main.*

```
#include <stdio.h>
extern int add();

main ()
{
        int x,y,z;

        scanf ("%d  %d", &x, &y);
        z = add (x, y);
        printf ("%d \n", z);
}
```

Assume that the following function is stored in the file *add.c.*

```
add (a, b)
int a, b;
{
        return (a + b);
}
```

You can compile these files and create an executable program by typing:

```
cc main.c add.c
```

The command compiles the statements in *main.c*, then compiles the statements in *add.c*. Finally, it links the two together (along with the standard C library) and copies the program to *a.out.* This program, like the program in the previous section, waits for two numbers, then prints their sum.

Since the **cc** command cannot keep track of more than one compiled file at a time, when several source files are compiled at a time, the command creates object files to hold the binary code generated for each source file. These object files are then linked to create an executable program. The object files have the same basename as the source files, but are given the ".o" file extension. For example, when you compile the two source files above, the compiler produces the object files *main.o* and *add.o.* These files are permanent files, that is, the command does not delete them after completing its operation. Note that the command also creates an object file if only one source file is compiled. :

Naming the Output File

You can give the executable program file any valid filename by using the **-o** (for output) option. The option has the form

```
-o filename
```

where *filename* is a valid filename or pathname.

Refer to "Chapter 10. The C Shell" in the section "Using Shell Variables" for more information about pathname. Also refer to the IBM Personal Computer *XENIX Basic Operations Guide.* If a filename is given, the program file is stored in the current directory. If a full pathname is given, the file is stored in the given directory. If that file already exists, its contents are replaced with the new executable program.

For example, the command:

```
cc  -o addem  main.c  add.o
```

causes the compiler to create an executable program file *addem* from the source file *main.c* and object file *add.o.* You can execute this program by typing:

```
addem
```

Note that the **-o** option does not affect the existing *a.out* file. This means that the **cc** command does not change the current contents of *a.out* if the **-o** option has been given.

Creating Small, Middle, and Large Programs

C programs in memory consist of the actual machine instructions created from the program's source statements, and the several bytes of binary data storage created for the program's variables. The data storage also contains the stack used by the program for temporary storage during execution. The XENIX system stores the instructions and data in one or more segments of physical memory. Each segment is 64K bytes long. Thus, the maximum allowable size for any program depends on how many segments are allocated for it when compiled.

The **cc** command lets you create programs of a variety of sizes and purposes using the **-Ms**, **-Mm**, **-Ml**, and **-i** options. These options define the size of a given program by defining the number of segments in the memory to be allocated for your program's use. They also determine how the system loads the program for execution.

The **cc** command allows the creation of programs in four different memory models: impure-text small model, pure-text small model, middle model, and large model. Each model defines a different type of program structure and storage.

Following is a brief description of each of these options.

The -Ms Option

You can create a small model program by using the **-Ms** option. Impure text small model programs are C programs that are short or have a limited purpose. These programs must not exceed 64K bytes. See the section "Creating Small Model Programs."

The -Ms and -i Options

You can combine the **-Ms** and **-i** options to create a pure-text small model program. Pure-text small model programs are typically short programs intended to be invoked by many users. Pure-text programs can occupy up to 128K bytes, but no more than 64K bytes each is permitted for either instructions or data. Unlike impure-text small model programs, the system loads only one copy of a pure-text program's instructions into memory, no matter how many times it has been invoked. As long as this copy stays in memory, the system simply loads a new copy of the data for each new invocation of the program. It then keeps each copy of data separate, while sharing the instructions among the different invocations. Pure-text programs save valuable memory space that would otherwise be wasted by small model programs. See the section "Creating Pure-Text Small Model Programs" later in this chapter.

The -Mm Option

You can create a middle model program by using the **-Mm** option. Middle model programs are typically C programs, that have a large number of program statements but a relatively small amount of data. Program instructions can be any size, but program data must not exceed 64K bytes. Refer to "Creating Middle Model Programs."

The -Ml Option

You can create a large model program by using the **-Ml** option. Large model programs are very large C programs that use a large amount of data storage during normal processing. Program instructions and data can have any size, except that the program must not contain a single array or structure exceeding 64K bytes. For an example refer to the "Creating Large Model Programs" section.

The following sections describe how to use the **-M** and **-i** options to create programs with a specific number of segments. They also describe how to create pure-text programs for execution by multiple users.

Creating Small Model Programs

You can create a small model program by using the **-Ms** option. This option directs **cc** to create a program that occupies a single segment when loaded into physical memory. To create a small model program, type:

```
cc -Ms filename
```

where *filename* is the name of the program you wish to compile.

The **cc** command creates small model programs by default when you do not otherwise specify a program model. Thus, the **-Ms** option is not required.

Creating Pure-Text Small Model Programs

You can create a pure-text small model program by combining the **-i** and **-Ms** options. The **-i** option directs **cc** to create separate memory segments for the instructions and data of a small model program. To create a pure-text program, type:

```
cc -Ms -i filename
```

where *filename* is the name of the file source program to be compiled. Since **cc** creates small model programs by default, only the **-i** option is required.

Creating Middle Model Programs

You can create a middle model program by using the **-Mm** option. This option creates one segment for the data of the program, and one or more segments for the instructions. To create a middle model program, type

```
cc -Mm filename  . . .
```

where *filename* is the name of the source file to be compiled. When creating a program, the compiler attempts to fit as many instructions into a segment (up to 64K bytes) as possible.

Middle model programs are pure in the sense that the system never loads more than one copy of the program's instructions into memory at one time. This means the **-i** option, used with pure-text small model programs, is not required for middle model programs.

Creating Large Model Programs

You can create large model programs by using the **-Ml** option. This option directs **cc** to create multiple segments for both instructions and data. To create a large model program, type

```
cc -Ml filename
```

where *filename* is the name of a source file to be compiled.

As with middle model programs, the compiler attempts to fit as many instructions into a segment as possible.

Like middle model programs, large model programs are considered pure.

Using Object Files and Libraries

The **cc** command lets you save useful functions as object files, and use these object files to create programs at a later time. Object files contain the compiled or assembled instructions of your source file, so they save you the time and trouble of recompiling the functions each time you need them. All object files created by **cc** have the file extension ".o".

The **cc** command also lets you use functions found in XENIX system libraries, such as the standard C library or the screen processing library *curses*. To use these functions, you simply supply the name of the library containing them. In some cases, such as for the standard C library, **cc** accesses the library automatically and no explicit naming is required.

For convenience, you can create your own libraries with the **ar** and **ranlib** commands. These commands, described in section CP of the IBM Personal Computer *XENIX Software Command Reference*, copy your useful object files to a library file, and prepare the file for use by the **cc** command. You can access the library like any other library in the system if you copy it to the */lib* directory.

Creating Object Files

You can create a linkable object file for each source file by using the **-c** (for compile) option. The **-c** option does not link these files. No executable program is created. This option directs **cc** to compile the source file without creating a final program. The option has the form:

```
cc -c filename . . .
```

where *filename* is the name of the source file. You can give more than one filename if you wish. Make sure each name is separated from the next by a space.

To make object files for the source files *add.c* and *mult.c*, type:

```
cc -c add.c mult.c
```

This command compiles each file and copies the compiled source files to the object files *add.o* and *mult.o*. It does not link these files; no executable program is created.

The **-c** option saves useful functions for programs developed later. Once a function is in an object file it can be used as is, or saved in a library file and accessed like other library functions, as described in the following sections.

The **cc** command automatically creates object files for each source file in the command line. Unless the **-c** option is given, however, it also attempts to link these files, even if they do not form a complete program.

> **Note:** For more information about the **compile** command, see IBM Personal Computer *XENIX Software Command Reference* (cc) (CP).

Creating Programs from Object Files

You can use the **cc** command to create executable programs from one or more object files, or from a combination of object files and C source files. The command compiles the source files (if any), then links the compiled source files with the object files to create an executable program.

To create a program, give the names of the object and source files you wish to use. For example, if the source file *main.c* contains calls to the functions *add* and *mult* (saved in the object files *add.o* and *mult.o),* you can create a program by typing:

```
cc main.c add.o mult.o
```

In this case, *main.c* is compiled, then linked with *add.o* and *mult.o* to create the executable file *a.out.*

Linking a Program to Functions In Libraries

You can link a program to functions in a library by using the **-l** (for library) option. The option directs **cc** to search the given library for the functions called in the source file. If the functions are found, the command links them to the program file.

The option has the form:

```
cc -lname
```

where *name* is a shortened version of the library's actual filename (see *Intro (S)* in the IBM Personal Computer *XENIX Software Command Reference*for a list of names). Spaces between the name and option are optional. The linker searches the */lib* directory for the library. If not found, it searches the */usr/lib* directory.

For example, the command:

```
cc main.c -lcurses
```

links the library */lib/libcurses.a* to the source file *main.c.*

A library is a convenient way to store a large collection of object files. The XENIX system provides several libraries, the most common of which is the standard C library. Functions in this library are automatically linked to your program whenever you invoke the compiler. Other libraries, such as *libcurses.a*, must be explicitly linked using the **-l** option. The XENIX libraries and their functions are described in detail in the IBM Personal Computer *XENIX Programmer's Guide to Library Functions.*

In general, the **cc** command does not search a library until the **-l** option is encountered, so the placement of the option is important. The option must follow the names of any source files containing calls to functions in the given library. In general, place all library options at the end of the command line, after all source and object files.

Creating Smaller and Faster Programs

You can create smaller and faster C programs by using the optimizing options available with the **cc** command. These options reduce the size of a compiled program by removing unnecessary or redundant instructions or unnecessary symbol information. Smaller programs usually run faster and save valuable space.

Creating Optimized Object Files

You can create an optimized object file or an optimized program from a given source file by using the **-O** (for optimize) option. This option reduces the size of the object file or program by removing unnecessary instructions. For example, the command:

```
cc -O main.c
```

creates an optimized program from the source file *main.c*. The resulting object file or program is smaller (in bytes) than if the source had been compiled without the option. A smaller object file usually means faster execution.

The **-O** option applies to source files only; existing object files are ignored if included with this option. The option must appear before the names of the files you wish to optimize. For example, the command:

```
cc -O add.c main.c
```

optimizes *main.c* and *add.c*.

You can combine the **-O** and **-c** options to compile and optimize source files without linking the resulting object files. For example, the command:

```
cc -O -c main.c add.c
```

creates separate optimized object files from the source files *main.c* and *add.c*.

Although optimization is very useful for large programs, it takes more time than regular compilation. In general, it should be used in the last stage of program development, after the program has been debugged.

Stripping the Symbol Table

To reduce the size of a program's executable file use the **-s** and **-x** options. These options direct **cc** to remove items from the symbol table. The symbol table contains information about code relocation and program symbols and is used by the XENIX debugger *adb* to allow symbolic references to variables and functions when debugging. The information in this table is not required for normal execution. Remove it when the program has been completely debugged.

The **-s** option strips the entire table, leaving machine instructions only. For example, the command:

```
cc -s main.c add.c
```

creates an executable program that contains no symbol table. It also creates the object files *main.o* and *add.o*, which contain no symbol tables.

The **-x** option strips all nonglobal symbols from the file including the names of local functions and variables, but excluding externally declared items. The command:

```
cc -x main.o add.o
```

creates an executable program with global symbols, but only if the object files *main.o* and *add.o* have symbol tables.

The **-s** option can be combined with the **-O** option to create an optimized and stripped program.

Also, the **-x** option can be combined with the **-O** option to create an optimized and stripped program. You can also strip a program with the XENIX command **strip**. See the IBM Personal Computer *XENIX Software Command Reference* for details.

Removing Stack Probes from a Program

You can reduce the size of a program slightly by using the **-K** option to remove all stack probes. A stack probe is a short routine called by a function to check the program stack for available space. The probes are not needed if the program makes very few function calls or has unlimited stack space.

To remove the stack probes from the program *main.c*, type

```
cc -K main.c
```

Although this option, when combined with the **-O** option, makes the smallest possible program, it should be used with great care. Removing stack probes from a program whose stack use is not well known can cause execution errors.

Preparing Programs for Debugging

The **cc** command provides a variety of options to prepare a program that is under development for debugging. These options range from creating an assembly language listing of the program, for use with the XENIX debugger **adb**, to adding routines for profiling the execution of a program.

Producing an Assembly Language Listing

You can direct the compiler to generate an assembly language listing of your compiled source file by using the **-S** and **-L** options. The **-S** option creates an assembly source listing of the compiled C source file and copies this listing to the file whose basename is the same as the source, but whose extension is ".s". This file is not suitable for assembly. This option provides code for reading only. The **-L** option creates a listing that shows assembled code, as well as instructions. The file created by **-L** is given the file extension ".L".

Assembly language listing files are used by programmers to debug their program with **adb**, since **adb** recognizes machine instructions instead of the actual source statements in your program. A programmer needs an assembly language listing for accurate debugging.

To create an assembly language listing, give the name of the desired source file. For example, the command:

```
cc -S add.c
```

creates an assembly language listing file named *add.s* and the command

```
cc -L mult.c
```

creates a listing file named *mult.L.* Both the **-S** and **-L** commands suppress subsequent compilation of the source file; they imply the **-c** option. Thus, no program file is created and no linking is performed.

Another use of the **-S** option is to create an assembly language source file. Although this method can be useful, optimizing should be left to the compiler whenever possible.

The **-S** and **-L** options apply to source files only; the compiler cannot create an assembly language listing file from an existing object file. Furthermore, the option in the command line must appear before the names of the files for which the assembly listing is to be saved.

Profiling a Program

You can examine the flow of execution of a program by adding "profiling" code to the program with the **-p** option. The profiling code automatically keeps a record of the number of times program functions are called during execution of the program. This record is written to the *mon.out* file and can be examined with the **prof** command.

For example, the command:

```
cc -p main.c
```

adds profiling code to the program created from the source file *main.c*. The profiling code automatically calls the *monitor* function, which creates the *mon.out* file at normal termination of the program. The **prof** command and *monitor* function are described in detail in *prof (CP)* and *monitor (S)* in the IBM Personal Computer *XENIX Software Command Reference.*

The **-p** option must be given in any command line that references object files that contain profiling code. For example, if the command

```
cc -c -p f1.c f2.c
```

was used to create the object files *f1.o* and *f2.o*, then the command

```
cc -p f1.o f2.o
```

must be used to create an executable program from these files.

Controlling the C Preprocessor

The **cc** command provides a number of options that let you control the operation of the C preprocessor. These options let you define macros, create new search paths for include files, and suppress subsequent compilation of the source file.

Defining a Macro

You may define the value or meaning of a macro used in a source file by using the **-D** (for define) option. The option lets you assign a value to a macro when you invoke the compiler, and is useful if you have used **if**, **ifdef**, and **ifndef** directives in your source files.

The option has the form:

```
-Dname[=string]
```

where *name* is the name of the macro and *string* is its value or meaning. If no *string* is given, the macro is assumed to be defined and its value is set to 1. For example, the command:

```
cc -DNEED=2 main.c
```

sets the macro NEED to the value 2. This is the same as having the directive:

```
#define NEED 2
```

in the source file. The command compiles the source file *main.c*, replacing every occurrence of NEED with 2.

The **-D** option is especially useful with the **ifdef** directive. You may use the option to determine which statements in the source are to be compiled. For example, suppose a source file, *main.c*, contains the directive

```
#ifdef NEED
```

but does not contain an explicit **define** directive for the macro NEED. Then all statements following the **ifdef** directive are compiled only if you define NEED using the **-D** option. For example, the command:

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

is sufficient to compile all statements following the **ifdef** directive, while the command

```
cc main.c
```

causes all those statements to be ignored.

You may use **-D** to define up to 11 macros on a command line. However, you may not redefine a macro once it has been defined. If a file uses a macro, you must place the **-D** option before that file's name on the command line. For example, in the command

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

the macro NEED is defined for *add.c* but not defined for *main.c.*

Defining Include Directories

You may explicitly define the directories containing "include" files by using the **-I** (for include) option. This option adds the given directory to a list of directories to be searched for include files. The directories in the list are searched whenever an include directive is encountered in the source file. The option has the form:

```
-Idirectoryname
```

where *directoryname* is a valid pathname to a directory containing include files. For example, the command:

```
cc -I/usr/joe/include main.c
```

causes the compiler to search the directory */usr/joe/include* for include files requested by the source file *main.c.*

The directories are searched in the order they are listed and only until the given include file is found. The */usr/include* directory is the default include directory and is always searched after directories given with **-I** .

Ignoring the Default Include Directories

You may prevent the C preprocessor from searching the default include directories by using the **-X** option. This option is generally used with the **-I** option to define the location of include files that have the same names as those found in the default directories, but which contain different definitions. For example, the command:

```
cc -X -I/usr/joe/include main.c add.c
```

causes **cc** to look for all include files only in the directory */usr/joe/include.*

Saving a Preprocessed Source File

You may save a copy of the preprocessed source file by using the **-P** and **-E** options. The file is identical to the original source file except that all C preprocessor directives have been expanded or replaced. The **-P** option copies the result to the file named *filename .i*, where *filename* is the same name as the source file without the .c extension. The **-E** option copies the result to the standard output, and places a **#line** directive at the beginning and end of this output. You may save this output by redirecting it. For example, the command:

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

creates a preprocessed file *main.i* from the source file *main.c*, and the command:

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

creates a preprocessed file from the source file *add.c*. The output is redirected to the file *add.i*.

Please observe that **-P** and **-E** suppress compilation of the source file. Thus, no object file or program is created.

Refer to "Chapter 10. The C Shell" in the section "Redirecting Input and Output" for an explanation of the redirection > symbol. Also, see the IBM Personal Computer *XENIX Basic Operations Guide.*

Error Messages

The C compiler generates a broad range of error and warning messages to help you locate errors and potential problems in programs. In addition to compiler messages, the **cc** command also displays error messages generated by the XENIX C preprocessor and the XENIX assembler and linker programs. (Refer to Appendix D for an alphabetical list of the linker error messages.) The following sections describe the form and meaning of the compiler error messages and warning messages you may encounter while using the **cc** command.

C Compiler Messages

The C compiler displays messages about syntactical and semantic errors in a source file, such as misplaced punctuation, illegal use of operators, and undeclared variables. It also displays warning messages about statements containing potential problems caused by data conversions or the mismatch of types. Error and warning messages have the form:

```
filename ( linenumber ) : message
```

where *filename* is the name of the source file being compiled, *linenumber* is the number of the line in the source file containing the error, and *message* is a self-explanatory description of the error or warning.

If an error is severe, the compiler displays a message and terminates the compilation. Otherwise, the compiler continues looking for other errors, but does not create an object file. If only warning messages are displayed, the compiler completes compilation and creates an object file.

You may avoid many C compiler errors by using the XENIX C program checker **lint** before compiling your C source files. The **lint** program performs detailed error checking on a source file, and provides a list of actual errors and possible problems that can affect execution of the program. For a description of **lint**, see "Chapter 4. The lint Program-a C Program Checker."

Setting the Level of Warnings

You may set the level of warning messages produced by the compiler by using the **-W** option. This option directs the compiler to display messages about statements that may not be compiled as the programmer intends. Warnings indicate potential problems rather than actual errors. The option has the form:

```
-W number
```

where *number* is a number in the range 0 to 3 giving the level of warnings. The levels are:

Level	**Warning**
0	Suppresses all warning messages. Only syntactical messages or semantic errors are displayed.
1	Warns about potentially missing statements, non-reachable statements, and other structural problems. Also, warns about overt type mismatches.
2	Warns about all type mismatches (strong typing).
3	Warns on all automatic data conversions.

If the option is not used, the default is level 1.

The higher option levels are especially useful in the earlier stages of program development when messages about potential problems are most helpful. The lower levels are best for compiling programs whose questionable statements are intentionally designed. For example, the command:

```
cc -W 3 main.c
```

directs the compiler to perform the highest level of checking, and produces the greatest number warning messages. The command:

```
cc -W 0 main.c
```

produces no warning messages. The **-w** option has the same effect as **-W 0**.

Using Advanced Options

The **cc** command provides a number of advanced programming options that give greater control over the compilation process and the final form of the executable program. The following sections describe a number of these options.

Creating Programs from Assembly Language Source Files

You may use the **cc** command to create executable programs from a combination of C source files and 8086/286 assembly language source files. Assembly language source files must contain 8086/286 instructions and must have the extension ".s".

When assembly language source files are given, the **cc** command invokes the XENIX assembler, **as**, to assemble the instructions and create an object file. The object file may then be linked with object files created by the compiler. For example, the command:

```
cc  main.c  add.s
```

compiles the C source file *main.c*, but assembles the assemble language source file *add.s*. The resulting object files, *main.o* and *add.o*, are linked to form a single program.

When using assembly language routines with C programs, you must be sure to provide the correct interface for calls to and from C language functions. C functions require a specific calling and return sequence. Assembly language functions which fail to provide this interface will cause errors. See Appendix A, "Assembly Language Interface," in the IBM Personal Computer *XENIX Programmer's Guide to Library Functions.*

Using the near and far keywords

The **near** and **far** keywords are special type modifiers that define the length and meaning of the address of a given variable. The **near** keyword defines an object with a 16-bit address. The **far** keyword defines an object with a full 32-bit segmented address. Any data item or function may be accessed.

The **near** and **far** keywords override the normal address length generated by the compiler for variables and functions. Therefore, you need to enable these keywords at compile time (with the -Me option). In small model programs, **far** lets you access data and functions in segments outside of the program. In middle and large model programs, **near** lets you access data with just an offset.

The examples in the following table illustrate the **far** and **near** keywords as used in declarations in a small model program.

Uses of near and far Keywords

Declaration	Address Size	Item Size
char c;	near(16 bits)	8 bits (data)
char far d;	far(32 bits)	8 bits (data)
char *p;	near(16 bits)	16 bits (near pointer)
char far *q;	near(16 bits)	32 bits (far pointer)
char * far r;	far(32 bits)	16 bits (near pointer) [1]
char far * far s;	far(32 bits)	32 bits (far pointer) [2]
int foo();	near(16 bits)	function returning 16 bits
int far foo();	far(32 bits)	function returning 16 bits[3]

Notes:

1. This example has no meaning; it is shown for syntactic completeness only.

2. This is similar to accessing data in a long model program.

3. This example leads to trouble in most environments. The far call changes the CS register, and makes run time support unavailable.

The following example is from a middle model compilation:

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

This does a near call in an otherwise far (calling) program.

Since there is no type checking between items in separate source files, use the **near** and **far** keywords with great care.

Setting the Stack Size

You can set the size of the program stack by using the **-F** option. This option has the form

```
-F num
```

where *num* is the size (in bytes) of the program stack. The program stack stores function parameters and automatic variables. If the option is not used, a default stack size (usually 4K bytes) is set.

You can determine the stack requirements of a given program by using the **stackuse** program. This program analyzes C source files and computes the minimum stack requirement for all functions in the program. The program displays a warning if recursive functions are encountered; stack use requirements for recursive functions must be determined by the programmer. The **stackuse** program is described in *stackuse (CP)* in the IBM Personal Computer *XENIX Software Command Reference.*

All programs created by **cc** have fixed stacks. This means the stack size cannot be increased during execution of the program. Therefore, a sufficient stack size must be given when compiling the program.

Using Modules, Segments, and Groups

"Module" is another name for the object file created by the C compiler. Every module has a name, and the **cc** command uses this name in error messages if problems are encountered during linking. The module name is usually the same as the source file's name (without the ".c" or ".s" extension). You can change this name using the **-NM** option. The option has the form:

```
-NM name
```

where *name* can be any combination of letters and digits.

Changing a module's name is useful if the source file to be compiled is actually the output of a program preprocessor and generator, such as **lex** or **yacc**.

A "segment" is a contiguous block of binary code produced by the C compiler. Every module has two segments: a text segment containing the program instructions, and a data segment containing the program data. Each segment in every module has a name. This name is used by **cc** to define the order in which the segments of the program appear in memory when loaded for execution. Text segments having the same name are loaded as a contiguous block of code. Data segments of the same name are also loaded as contiguous blocks.

Text and data segment names are normally created by the C compiler. These default names depend on the memory model chosen for the program. For example, in small model programs the text segment is named "_TEXT" and the data segment is named "_DATA". These names are the same for all small model modules, so all segments from all modules of a small model program are loaded as a contiguous block. In middle model programs, each text segment has a different name. In large model programs, each text and data segment has a different name. The default text and data segment names for middle and large model programs are given in the section "Segment and Module Names" given at the end of this chapter.

You can override the default names used by the C compiler (and override the default loading order) by using the **-NT** and **-ND** options. These options set the names of the text and data segments, in each module being compiled, to a given name. The options have the form:

```
-NT name
```

and

```
-ND name
```

where *name* is any combination of letters and digits. These options are useful in middle and large model programs that have no specific loading order. In these programs, you can guarantee contiguous loading for two or more segments by giving them the same name.

All text and data segments, whether or not they are loaded as contiguous blocks, are eventually loaded into one or more physical segments of memory. All segments in a physical segment are collectively called a "group".

All programs have at least two groups: a text group and a data group. Each group has a name. The text group is named IGROUP and the data group is named DGROUP. The C compiler automatically applies these names to the text and data segments in each module. Thus, when the modules are eventually linked, all text segments belong to the same group, and all data segments belong to the same group.

Creating Programs for DOS

You can create IBM Personal Computer Disk Operating System (DOS-executable) C programs from existing XENIX C language source files by using the -dos option. The option directs **cc** to compile and link C language source files for execution on DOS systems. The **cc** uses a special DOS linker, include files, and libraries to produce an executable program file that can be copied to and run on any DOS system.

For a complete description of the -dos option and the cross development tools available under the IBM Personal Computer XENIX system, see "Chapter 3. XENIX to DOS: A Cross Development System."

Compiler Summary

The following sections summarize **cc** options and memory models.

CC Options

The following is a complete list of **cc** options:

-c	Creates a linkable object file for each source file.
-C	Preserves comments when preprocessing a file (only when **-P** or **-E**). Refer to IBM Personal Computer *XENIX Software Command Reference cc(CP).*
-dos	Creates an executable program for DOS. (Program requires an .exe extension to run under DOS. Refer to Chapter 3.)
-D name [= string]	Defines *name* to the preprocessor. The value is *string* or 1.

-E	Preprocesses each source file, copying the result to the standard output.
-F num	Sets the size of the program stack.
-i	Creates separate instruction and data spaces for small model programs.
-I pathname	Adds *pathname* to the list of directories to be searched for #include files.
-K	Removes stack probes from a program.
-l name	Search library *name* for unresolved function names.
-L	Creates an assembler listing file containing assembled code and assembly source instructions.
-M string	Sets the program configuration. The *string* can be any combination of "s" (small model), "m" (middle model), "l" (large model), "e" (enable far and near keywords), "2" (enables 286 code generation)-- this is the default, and, "t" (sets data threshold for largest item in a segment). The "s," "m," and "l" are mutually exclusive.
-nl num	Sets the maximum length of external symbols. Refer to IBM Personal Computer *XENIX Software Command Reference (cc)(CP).*
-ND name	Sets the data segment name.

-NM name	Sets the module name.
-NT name	Sets the text segment name.
-o filename	Makes *filename* the name of the final executable program.
-O	Invokes the object code optimizer.
-p	Adds code for program profiling.
-P	Preprocesses source files and sends output to files with the extension ".i".
-s	The -s option strips the entire table, leaving machine instructions only.
-S	Creates an assembly source listing.
-V string	Copies *string* to the object file.
-w	Suppresses compiler warning messages.
-W num	Sets the output level for compiler warning messages.
-x	The -x option strips all nonglobal symbols from the file.
-X	Removes the standard directories from the list of directories to be searched for #include files.

Memory Models

The following table defines the number of text and data segments for the four different program memory models. This table also lists the segment register values.

Model	Text	Data	Segment Registers
Small	1*	1*	CS=DS=SS
Middle	1 per module	1	DS=SS
Large	1 per module	1 per module	

*In impure-text small module programs, text and data occupy the same segment. In pure-text programs, they occupy different segments.

Pointer and Integer Sizes

The following table defines the sizes (in bits) of integers (**int** type), and text and data pointers, in each program memory model.

Model	Data Pointer	Text Pointer	Integer
Small	16	16	16
Middle	16	32	16
Large	32	32	16

Segment and Module Names

The following table lists the default text and data segment names, and the default module name, for each object file.

Model	Text	Data	Module
Small	_TEXT	_DATA	*filename*
Middle	*module*_TEXT	_DATA	*filename*
Large	*module*_TEXT	*module*_DATA	*filename*

Chapter 3. XENIX to DOS: A Cross Development System

Contents

Introduction

The IBM Personal Computer XENIX system provides a variety of tools to create programs that can be executed under control of the IBM Disk Operating System (DOS). The DOS cross development system lets you create, compile, and link DOS programs on the XENIX system and transfer these programs to a DOS system for execution and debugging.

The complete development system consists of

- The C program compiler **cc**
- The 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. The command provides a special **-dos** option which directs the compiler to create code for execution under DOS. When **-dos** is given, **cc** uses the special DOS include files and libraries to create a program. The resulting program file has the correct format for execution on any DOS system.

The **cc** command uses the **dosld** commands to carry out the last part of the compilation process, the creation of the executable program file. **Cc** only invokes the **as** command when 8086 assembly language source files are given in the command line. In most cases, **cc** invokes **as** and **dosld** automatically. You can also invoke them directly when you need to perform special tasks.

The last important step in the cross development process is to transfer the executable program files to a DOS system. Since DOS programs cannot be executed or debugged on the XENIX system, you must copy the resulting programs to DOS file systems before attempting execution. You can do this using the XENIX *dos(C)* commands. For example, the **doscp** command lets you copy files back and forth between XENIX and DOS disks. This means you can transfer program files from the XENIX system to a DOS system, or copy source files from a DOS system to a XENIX system.

Creating Source Files

You can create program source files using either XENIX or DOS text editors. The most convenient way is to use a XENIX editor, such as **vi**, since this means you do not have to transfer the source files from the DOS system to XENIX system each time you make changes to the files.

When creating source files, you should follow these simple rules:

- Use the standard C language format for your source files. DOS source files have the same format as XENIX source files. In fact, many DOS programs, if compiled without the **-dos** option, can be executed on the XENIX system.

- Use the DOS naming conventions when giving file and directory names within a program, for example, use \ instead of / for the pathname separator. Since the compiler does not check names, failure to follow the conventions will cause errors when the program is executed.

- Use only the DOS include files and library functions. Most DOS include files and functions are identical to their XENIX counterparts. Others have only slight differences. For a complete list of the available DOS include files and functions, and a description of the differences between them and the corresponding XENIX files and functions, see Appendix C of the IBM Personal Computer *XENIX Programmer's Guide to Library 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 XENIX **cc** command. The command line has the form

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

where *options* are other **cc** command options (as described in Chapter 2), and *filename* is the name of the source file you wish to compile. You can give more than one source file if desired. Each source filename must end with the ".c" extension.

The **cc** command compiles each source file separately, creating an object file for each file, then links all object files together with the appropriate C libraries. The object files created by the **cc** command have the same base name as the corresponding source file, but end with the ".o" extension instead of the ".c" extension. The resulting program file also has the name *a.out* if no name is explicitly given.

For example, the command

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

compiles the source file *test.c* and creates the object file *test.o* . It then calls **dosld** which links the object file with functions from the DOS libraries. The resulting program file is named, *a.out* .

You can use any number of **cc** options in the command line. The way the options work is described in Chapter 2 of this book. For example, you can use the **-o** option to explicitly name the resulting program file, or the **-c** option to create object files without creating a program file. In some cases, the default values for an option are different than when compiling for a XENIX system. In particular, the default directory for library files given with the **-l** option is */usr/lib/dos* . Note that the **-p** (for profiling) option cannot be used.

Using Assembly Language Source Files

You can direct **cc** to assemble 8086 assembly language source files by including the files in the **cc** command line. Like C source files, assembly language source files may only contain calls to functions in the DOS libraries. Furthermore, the source files must follow the C calling conventions described in Appendix A of the IBM Personal Computer *XENIX Programmer's Guide to Library Functions*. The filename of an assembly language source file must end with the ".s" extension.

When an assembly language source file is given, **cc** automatically invokes **as** the 8086 assembler. The assembler creates an object file that can be linked with any other object file created by **cc**.

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

Creating Linking Object Files

You can link DOS object files previously created by **cc** or **as** by giving the names of the files in the **cc** command line. The object files must have been created using **as** or with **cc** using the **-dos** option. Object files created without using the **-dos** cannot be linked to DOS programs. The object filenames must end with the ".o" extension.

When an object file is given, **cc** automatically invokes **dosld** the DOS linker which links the given object files with the appropriate C libraries. If there are no errors, **dosld** creates an executable program file named *a.out* .

You can invoke the linker directly by using the **dosld** command. The command creates a DOS program file just as the **cc** command, but does not accept source files. For a description of the command and its options, see *dosld (CP)* in the IBM Personal Computer *XENIX Software Command Reference.*

> **Note:** DOS programs created by **cc** and **dosld** are completely compatible with the DOS system and can be executed on any such system. DOS programs cannot be executed on the XENIX system.

Running and Debugging a DOS Program

You can debug a DOS program by transferring the program file to a DOS system and using the DOS debugger, **Debug**, to load and execute the program. The following section explains how to transfer program files between systems. For a description of the **Debug** program, see your IBM Personal Computer *Disk Operating System* manual.

Transferring Programs Between Systems

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

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

where **-r** is the required "raw" option, *file-1* is the name of the DOS program file you wish to transfer, *dev* is the full pathname of a XENIX system diskette drive, and *file-2* is the full pathname of the new program file on the DOS disk. The new filename must have the .exe extension. The **-r** option ensures that the program file is copied byte for byte.

To transfer a program file to a DOS system, follow these steps:

1. Insert a formatted DOS diskette into a XENIX system diskette drive.

2. Use the **doscp** command to copy the program file to the disk. For example, to copy the program file *a.out* to the file *test.exe* on the DOS disk into the disk drive */dev/fd0* , type:

   ```
   doscp -r a.out  /dev/fd0:/test.exe
   ```

3. Remove the diskette from the drive.

You can now insert the diskette into the diskette drive of the DOS system and invoke the program just as you would any other DOS program.

> **Note:** DOS program files that do not end with the .EXE or .COM extension cannot be loaded for execution under DOS. When transferring program files from a XENIX system to DOS, you must make sure you rename *a.out* files to an appropriate .exe or .com file.

Creating DOS Libraries

You can create a library of your own DOS object files by using the XENIX **ar** command. The command copies object files created by the compiler to a given archive file. The command has the form:

```
ar archive filename  . . .
```

where *archive* is the name of an archive file, and *filename* is the name of the DOS object file you wish to add to the library.

> **Note:** DOS libraries created on the 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 4. The lint Program-a C Program Checker

Contents

Introduction

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

In particular, **lint** checks for:

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

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

Invoking Lint

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

```
lint [ option] . . . filename . . . lib . . .
```

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

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

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

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

```
filename ( num ): description
```

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

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

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

Checking for Unused Variables and Functions

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

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

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

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

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

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

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

Some programming styles require functions that perform closely related tasks to have the same number and type of arguments, regardless of whether these arguments are used. Under normal operation, **lint** reports any argument not used as an unused variable, but you can direct **lint** to ignore unused arguments by using the **-v** option. The **-v** option causes **lint** to ignore all unused function arguments except for those declared with **register** storage class. The program considers unused arguments of this class to be a preventable waste of the register resources of the computer.

You can direct **lint** to ignore all unused variables and functions by using the **-u** (for unused) option. This option prevents **lint** from reporting variables and functions it considers unused. The **-u** option checks a source file that contains just a portion of a large program. Such source files usually contain declarations of variables and functions intended to be used in other source files and are not explicitly used within the file. Since **lint** can only check the given file, it assumes that such variables or functions are unused and reports them as errors whenever the **-u** option is not given.

Checking Local Variables

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

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

```
char c;

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

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

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

```
int i,total;

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

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

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

Checking for Unreachable Statements

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

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

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

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

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

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

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

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

Checking for Infinite Loops

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

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

and

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

are both considered infinite loops. While the statements:

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

and

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

are never executed.

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

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, the appearance of both:

```
return (expr);
```

and

```
return ;
```

statements is cause for alarm. In this case, **lint** produces 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. This is demonstrated with a simple example.

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

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

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. This is considered a serious error.

Checking Types

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

1. Across certain binary operators and implied assignments

2. At the structure selection operators

3. Between the definition and uses of functions

4. In the use of enumerations

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

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

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

The **lint** program 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 in C was introduced largely as an aid to producing more portable programs. Consider the assignment:

```
p = 1 ;
```

where "p" is a character pointer. The **lint** program reports this as suspect. But consider the assignment

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

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

Checking for Nonportable Character Use

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

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

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

```
nonportable character comparison
```

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

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

Checking for Assignment of longs to ints

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

Checking for Strange Constructions

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

```
*p++ ;
```

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

```
null effect
```

The program fragment:

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

is also strange since the test will never succeed.

Similarly, the test:

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

is equivalent to:

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

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

```
degenerate unsigned comparison
```

If you use

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

then **lint** reports:

```
constant in conditional context
```

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

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

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

or

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

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

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

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

Checking for Use of Older C Syntax

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

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

```
a =-1 ;
```

which could be taken as either

```
a =-  1 ;
```

or

```
a  =  -1 ;
```

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

A similar issue arises with initialization. The older language allowed

```
int  x  1 ;
```

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

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

looks somewhat like the beginning of a function declaration

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

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

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

This form is free of any possible syntactic ambiguity.

Checking Pointer Alignment

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

```
possible pointer alignment problem
```

results from this situation.

Checking Expression Evaluation Order

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

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

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

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

```
/* NOTREACHED */
```

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

```
/* NOSTRICT */
```

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

```
/* ARGSUSED */
```

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

```
/* VARARGS */
```

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

```
/* VARARGS2 */
```

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

```
/* LINTLIBRARY */
```

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

Checking for Library Compatibility

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

```
-ly
```

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

```
/* LINTLIBRARY */
```

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

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

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

Chapter 5. A Program Maintainer: make

Contents

Introduction

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

This chapter explains how to use **make** to automate medium-sized programming projects. It explains how to create makefiles for each project, and how to invoke **make** for creating programs and updating files. For more details about the program, see **make (CP)** in the IBM Personal Computer *XENIX Software Command 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 are required to bring a file up to date. A dependency line has the form:

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

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

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

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

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

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

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

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

```
#include "defs"
```

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

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

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

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

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

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

Invoking make

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

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

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

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

```
make
```

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

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

```
make x.o
```

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

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

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

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

```
-f filename
```

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

```
make -f makeprog
```

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

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 XENIX commands.
-k	Abandons work on the current entry, but continues on other branches that do not depend on that entry.
-s	Executes commands without displaying them.
-r	Ignores the built-in rules.
-n	Displays commands but does not execute them. 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, care must be taken with certain commands (for example, **cd** and shell control commands) that have meaning only within a single shell process; the results are forgotten before the next line is executed. If an error occurs, **make** normally stops the command.

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 given object files when the pseudo-target name "cleanup" is given in the invocation of **make**.

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

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

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, allowing **make** to continue after an error. This is the same as the **-i** option. Make also ignores errors for a given command if the command string begins with a hyphen (-).

The pseudo-target name .DEFAULT defines the commands to be executed either when no built-in rule or 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 using the Interrupt or Quit key, and 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 filename or command option. The macros can be defined when you invoke **make** or in the makefile itself.

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

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

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

A macro is invoked by preceding the macro name with a dollar sign; macro names longer than one character must be placed in parentheses.

The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations.

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

The last two invocations are identical.

Macros are typically used as placeholders for values that may change from time to time. For example, the following makefile uses a 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 this makefile is invoked with the command:

```
make
```

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 spaces are in the definition, use double quotation marks to enclose the definition. Macros in a command line override corresponding definitions found in the makefile. For example, the command:

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

loads assigns the library options **-lln** and **-lm** to "LIBES."

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 to 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 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* . It can be used in dependency lines that redefine the built-in rules.
$@	Contains the full pathname of the current target. It can be used in dependency lines with user-defined target names.
$<	Contains the filename of the dependent that is more recent than the given target. It can be used in dependency lines with built-in target names or the **.DEFAULT** pseudo-target name.
$?	Contains the filenames of the dependents that are more recent than the given target. It can be used in dependency lines with user-defined target names.
$%	Contains the filename of a library member. It can be used with target library names (see the section "Using Libraries" later in this chapter). In this case, $@ contains the name of the library and $% contains the name of the library member.

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

As an example, let's say 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 pathname 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 filename 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 makefile, setting MAKE to make and invoking this file with the **-n** options displays all the commands used to generate the programs *prog1*, *prog2*, and *prog3* without actually executing them.

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

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

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

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

Using the Built-In Rules

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

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

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

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

The built-in rules are designed to reduce the size of your makefiles. They provide the rules for creating common files from typical dependents. Reconsider the example given in the section "Creating a Makefile ." In this example, the program *prog* depended on three object files *x.o*, *y.o*, and *z.o* . These files in turn depended on the C language source files *x.c*, *y.c*, and *z.c* . The files *x.c* and *y.c* also 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, since the built-in rules could have been used instead. The following is all that is needed to show the relationships between these files.

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

x.o y.o: defs
```

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

Changing the Built-in Rules

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

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

The 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 LOADLIBES.

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

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

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

```
suffix-rule:
        command
```

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

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

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

A tilde (~) indicates an SCCS file. A single suffix 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 wish to access by using a library name. A library name has the form:

```
lib(member-name)
```

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

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

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

You can create your own built-in rules for archive libraries by adding the *.a* suffix to the suffix list, and creating new suffix combinations. For example, the combination ".c.a" 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 than the suffix of the ultimate file. For example, the combination ".c.a" can be used for a rule that creates *.o* files, but not for one that creates *.c* files.

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

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

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

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

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

Troubleshooting

Most difficulties in using **make** arise from **make**'s specific meaning of dependency. 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 normally execute without actually executing them.

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

If a change to a file is 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
```

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

Using make: An Example

An example of the use of **make** is shown at the end of this chapter. Examine the makefile used to maintain the **make** itself. The code for **make** is spread over a number of C source files and a **yacc** grammar.

The **make** program usually prints out each command before issuing it. The following output results from 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

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

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

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

lint :  dosys.c doname.c files.c main.c misc.c vers.c gram.c
	$(LINT) dosys.c doname.c files.c main.c misc.c vers.c gram.c
	rm gram.c
arch:
	ar uv /sys/source/s2/make.a $(FILES)
```

Makefile Contents

Chapter 6. SCCS: A Source Code Control System

Contents

Introduction

The Source Code Control System (SCCS) is a collection of IBM Personal Computer XENIX 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 you wish at any time, make changes to this version, and save the changes as a new version of the file in the SCCS file.

The SCCS system is useful wherever you require a compact way to store multiple versions of the same file. The SCCS system provides an easy way to update any given version of a file and explicitly record the changes made. The commands are used to control changes to multiple versions of source programs, but can also 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 system. In particular, it describes:

1. Files and directories

2. Deltas and SIDs

3. SCCS working files

4. SCCS command arguments

5. 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 XENIX 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 should have read (examine) permission for everyone, and write permission for the user only.

You must not use the XENIX **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 corresponds to the changes needed to construct exactly one version of the file. Then combine the lists to create the desired version from the original.

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

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

An SID can also have two optional numbers. The branch number 3, the optional third number, indicates changes at a particular level, and the "sequence number," the fourth number, indicates changes at a particular branch. For example, the SIDs 1.1.1.1 and 1.1.1.2 indicate two new versions that contain slight changes to the original delta 1.1.

An s-file can contain several different releases, levels, branches, and sequences of the same file. In general, the maximum number of releases an s-file 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 system usually creates a new SID by incrementing the level number of the original version. If you wish to create a new release, you must explicitly instruct the system to do so. A change to a release number indicates a major new version of the file. How to create a new version of a file and change release numbers is described later.

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

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

s-file A permanent file that contains all versions of the given text file. The versions are stored as deltas, that is, lists of changes to be applied to the original file to create the given version. The name of an s-file is formed by placing the file prefix *s.* at the beginning of the original filename.

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

l-file	A special file containing a list of the deltas required to create a given version of a file. The l-file name is formed by placing the prefix *l.* at the beginning of the original filename.
d-file	A temporary copy of the g-file used to generate a new delta.
q-file	A temporary file used by the **delta** 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 wish 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 filenames. 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 filename indicates the file to be acted on. The syntax for SCCS filenames is like other XENIX filename syntax. Appropriate pathnames must be given if required. Some commands also allow directory names. In this case, all files in the directory are acted on. If the directory contains non-SCCS and unreadable files, these are ignored. A filename must not begin with a minus sign (-).

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

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

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

File Administrator

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

Creating and Using S-files

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

This section explains how to use the **admin**, **get**, and **delta** commands to create and use s-files. In particular, it describes how to create the first version of a file, 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 using the **-i** (for initialize) option of the **admin** command. The command has the form:

admin -i **filename s.filename**

where -ifilename gives the name of the text file from which the s-file is to be created, and *s.filename* is the name of the new s-file. The name must begin with *s.* and must be unique; no other s-file in the same directory 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 using the **rm** command, since it is no longer needed. If you wish to view the text file or make changes to it, you can retrieve the file using the **get** command described in the next section.

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

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

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

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

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

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

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

```
1.1
6 lines
```

You 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 filename before the relevant information.

Retrieving a File for Editing

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

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

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

The command retrieves the lastest version of the text file and copies it to an ordinary text file. The file has the same name as the s-file but with the *s.* removed. It has read and write file permissions. For example, suppose the s-file *s.demo.c* contains the first version of a C program. To retrieve this program, 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 filename, the command creates files for each corresponding s-file. Since the **-e** option applies to all the files, you can edit each one.

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

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

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

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

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

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

After a command has copied the new delta to the s-file, it displays a message showing the new SID and the number of lines inserted, deleted, or left unchanged in the new version.

For example, if the C program has been changed to:

```
#include <stdio.h>

main ()
{
     int i = 2;

     printf("This is version 1.%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 wish 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 you wish from an s-file by using the **-r** (for retrieve) of the **get** command. The command has the form:

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

where **-e** is the edit option, -rSID 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 filename. Separate the names with with spaces.

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

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

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

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

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

You can omit the level number of a version number if you wish, that is, just give a 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. If no version with the given release number exists, the command retrieves the most recent version in the previous release.

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 -rrel-num *s.filename* . . .

where **-e** is the required edit option, **-rrel-num** 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 may give more than one filename.

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

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

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

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

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.

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

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

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 incrementing the sequence number by one. For example, if you retrieve branch version 1.3.1.1 for editing, **get** gives 1.3.1.2 as the proposed SID.

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

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

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

retrieves version 1.3.1.4.

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* at the standard output. Similarly, the command:

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

displays version 2.1 at the standard output.

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

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

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

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 as desired, 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"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

The **help** command can be used at any time.

Using Identification Keywords

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

This section explains how keywords are treated by the various SCCS commands, and how you 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 section *get (CP)* in the IBM Personal Computer *XENIX Software Command Reference.*

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 the program is retrieved for reading using the **get** command, the keywords are replaced by their current values. For example, if the %M%, %I% and %H keywords are used in place of the module name, the SID, and the current data 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 wish 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 a fatal error, as explained later in this chapter.

Assigning Values to Keywords

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

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

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

Forcing Keywords

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

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

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

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

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 section *admin (CP)* in the IBM Personal Computer *XENIX Software Command 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 -f**flag** s.filename

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

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

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

Some s-file flags take 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

The **i** flag causes the **admin** and **delta** commands to print a fatal error message and stop, if no keywords are found in the given text file. The flag is used to prevent a version of a file, 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 -d**flag** s.filename

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

```
admin -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 contents.

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"George Wheeler"  s.demo.c
```

saves the comment "George Wheeler" 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 -r**SID** s.filename

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

```
comments?
```

You 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 explicitly set with the **-f** option. For example, the command:

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

inserts the modification request “error0” in the new file *s.def.h.*

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

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

admin -t*filename* *s.filename*

where -t filename gives the name of the file containing the descriptive text, and *s.filename* is the name of the s-file to receive the descriptive text. The file to be inserted 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 filename. 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 -d**spec** **s.filename**

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

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

For example, the command:

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

may produce the line:

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

A complete list of the data keywords is given in the section *prs (CP)* in the IBM Personal Computer *XENIX Software Command 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 -r*SID s.filename*

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

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

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

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

Printing Later and Earlier Versions

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

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

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

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

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

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

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

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

When the flag is set, the **get** command uses the next available branch SID for each new proposed SID. For example, suppose a user retrieves 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 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 -r*SID* *s.filename*

where -r**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 XENIX system file permissions to protect s-files from changes by unauthorized users. In addition to the XENIX system protections, the SCCS system provides two ways to protect the s-files: the "user list" and the "protection flags." The user list is a list of login names and group IDs of users who are allowed to access the s-file and create new versions of the file. The protection flags are three special s-file flags that define 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 -a name gives the login name of the user or the group name of a group of users to be added to the list, and *s.filename* gives the name of the s-file to receive the new users. For example, the command:

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

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

If you create an s-file without giving the **-a** option, the user list is left empty, and all users 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 -e 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, making 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 filename. If you do, the command checks each file in turn. 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 the IBM Personal Computer *XENIX Basic Operations Guide*.) If this is not possible, the file can be edited using a XENIX text editor.

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

Changing an S-file's Checksum

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

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

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

Regenerating a G-file for Editing

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

Restoring a Damaged P-file

The **-g** option of the **get** command generates a new copy of a p-file accidentally removed. For example, the command:

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

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

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 a short explanation of the command and command syntax. For example, the command:

```
help  rmdel
```

displays the message:

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

Creating a File with the Standard Input

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

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

causes **admin** to create a new s-file named *s.demo.c* 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 -rrel-num *s.filename*

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

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

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

Adding a Comment to the First Version

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

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

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

The comment 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 wish to include as well as the ones you wish to exclude when creating a g-file, by using the **-i** and **-x** options of the **get** command.

The **-i** option causes the command to apply the given deltas when constructing a version. The **-x** option causes the command to ignore the given deltas when constructing a version. Both options must be followed by one or more SIDs. If multiple SIDs are given they must be separated by commas (,). A range of SIDs 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 wish to automatically apply changes to a version while retrieving it for editing. For example, the command:

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

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

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

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

retrieves version 1.6 for editing. When the file is retrieved, the changes in delta 1.5 are automatically left out of it, making the g-file the same as if version 1.4 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 in which the problem can exist. 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* containing the deltas required to create the most recent version of *demo.c*.

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

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

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

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 XENIX **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 -r**SID** **s.filename**

where -rSID 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 refuses to remove a protected delta, that is, a delta whose release number is below the current floor value, above the current ceiling value, or equal to a current locked value (see the section "Protecting S-files" given earlier in this chapter). The command refuses to 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.

Please observe that **rmdel** changes the type indicator of the given delta from "D" to "R". A type indicator defines the type of delta. Type indicators are described in full in the section *delta (CP)* in the IBM Personal Computer *XENIX Software Command 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 -r**SID1**-r**SID2** **s.filename**

where -rSID1 and -rSID2 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. The differences are displayed in a form similar to the XENIX **diff** command. This command prints on the standard output the differences between two versions of the s-file. The command has the form:

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

where -rSID1 and -rSID2 give the SIDs of the versions to be compared, and *s.filename* is the name of the s-file containing the versions. The version SIDs must be given in the order in which they were created.

Chapter 7. The adb Program Debugger

Contents

Introduction

The **adb** program is a debugging tool for C and assembly language programs. It carefully controls the execution of a program while letting you examine and modify the program's data and text areas.

This chapter explains how to use **adb**. In particular, it explains how to:

1. Start the debugger

2. Display program instructions and data

3. Run, breakpoint, and single-step a program

4. 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 wish 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** normally 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 filename. In this case, **adb** searches for the default file *a.out* in your current working directory and prepares it for debugging. Thus, the command:

```
adb
```

is the same as typing

```
adb a.out
```

The **adb** program debugger displays an error message 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 and provide a way to determine the cause of an error.

To examine a core image file with its corresponding program, you must give the name of both the core and and the program file. The command line has the form:

```
adb programfile corefile
```

where *programfile* is the filename of the program that caused the error, and *corefile* is the filename of the core image file generated by the system. 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 regardless of whether or not the file belongs to the given program. You can prevent **adb** from opening this file by using the hyphen (-) in place of the core filename. For example, the command:

```
adb sample -
```

prevents **adb** from searching your current working directory for a core file. 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

You can make changes and corrections in a program or data file using **adb** if you open it for writing using the **-w** option. For example, the command:

```
adb -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. See the section "Patching Binary Files" later in this chapter.

Starting with the Prompt Option

You can define the prompt used by **adb** by using the **-p** option. The option has the form:

```
-p prompt
```

where *prompt* is any combination of characters. If you use spaces, enclose the *prompt* in quotes. For example, the command:

```
adb -p "Mar 10->" sample
```

sets the prompt to Mar 10->. The new prompt takes the place of the default prompt (*) when **adb** begins to prompt for commands.

Make sure there is at least one space between the **-p** and the new prompt, otherwise **adb** 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** command by pressing the Interrupt (Del) or Quit (Crtl \) keys. These keys are caught by **adb** and cause it to 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, and the meaning of each of the display commands.

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.

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 may have octal digits only. Hexadecimal numbers must begin with the prefix "0x" and may contain decimal digits and the letters "a" through "f" (in both 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 globol variables and functions defined within the program being debugged and are equal to the address of the given variable or function. Symbols are stored in the program's symbol table and are available if the symbol table has not been stripped from the program file (see *strip (CP)* in the IBM Personal Computer *XENIX Software Command 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.

`d`	`size of data`
`e`	`entry address of the program`
`m`	`execution type`
`n`	`number of segments`
`s`	`size of stack`
`t`	`size of text`

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 a .b adb variable in an expression by preceding the variable name with an less than (<) sign. For example, the current value of the base variable "b" is

```
<b
```

You can create your own variables or change the value of an existing variable by assigning a value to a variable name with the greater than (>) sign. The assignment has the form

```
expression > variable-name
```

where *expression* is the value to be assigned to the variable, and *variable-name* must be a single letter. For example, the assignment

```
0x2000>b
```

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

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 are usually the same except when implied commands, such as the newline and caret (^) characters, are used. These automatically increment and decrement (.) but leave (") unchanged.

Both the (.) and the (") 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 may 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 expression

```
2*3+4
```

is equal to 10 and

```
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 resolves 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.

Choosing Data Formats

A format is a letter or character that defines how data is to be displayed. The following are the most commonly used formats:

Letter	Format
o	1 word in octal
d	1 word in decimal
D	2 words in decimal
x	1 word in hexadecimal
X	2 words in hexadecimal
u	1 word as an unsigned integer
f	2 words in floating point
F	4 words in floating point
c	1 byte as a character
s	a null terminated character string
i	machine instruction
b	1 byte in octal
a	the current 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** to 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** type variables and **s** is for arrays of **char** 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 wish it to be repeated. For example the format:

```
4c
```

displays four ASCII characters.

It is possible to combine format requests to provide elaborate displays. For example, the command:

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

displays four octal words followed by their ASCII interpretation from the data space of the core image file. In this example, the display starts at the address <b, the base address of the program's data. The display continues until the end-of-the-file since the negative count -1 causes an indefinite execution of the command until an error condition such as the end-of-the-file occurs. 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 "*C" redisplays 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 "at" sign itself is displayed as a double at sign "@@."

Using the Equal Command

The (=) command displays a given address in a given format. The command is used primarily to display instruction and data addresses in simpler form, or to display the results of arithmetic expressions. For example, the command:

```
main=A
```

displays the absolute address of the symbol "main" (giving the segment and offset) and the command:

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

displays (in decimal) the sum of the variable "b" and the hexadecimal value "0x2000."

If a count is given, the same value is repeated that number of times. For example, the command:

```
main,2=x
```

displays the value of main twice.

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

```
.=
```

If no format is given, the previous format 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 backslash Commands

You can display the contents of a text or data segment with the **(?)** and **(/)** commands. The commands have the form:

```
[ address ][, count ] ? [ format ]

[ address ] [,count ] / [ format ]
```

where *address* is an address with the given segment, *count* is the number of items you wish to display, and *format* is the format of the items you wish to display.

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

```
main,5?ia
```

displays five instructions starting at the address, main, and the address of each instruction is displayed immediately before it. 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.

An Example: Simple Formatting

This example illustrates how to combine formats in **(?)** or **(/)** commands to display different types of values whene 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 wish to see them all in hexadecimal, type:

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

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

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

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

In this case, the command displays 4 values in hexadecimal, then 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 by typing:

```
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 assembly language listing of your C program before using **adb**, then refer to the listing as you use the debugger. To create an assembly language listing, use the **-S** option of the **cc** command (see Chapter 2, "Cc: a C Compiler").

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 wish to skip before one is taken, and *arguments* are the command line arguments, such as filenames and options, you wish to pass to the program.

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

```
:r
```

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

```
,5:r
```

causes **adb** to skip the first 5 breakpoints.

If arguments are given, they must be separated by at least one space each. The arguments are passed to the program in the same way the system shell passes command line arguments to a program. You can use the shell redirection symbols if you wish.

The **:R** command passes the command arguments through the shell before starting program execution. This means you can use shell metacharacters in the arguments to refer to multiple files or other input values. The shell expands arguments containing metacharacters before passing them on to the program.

The command is especially useful if the program expects multiple filenames. For example, the command:

```
:R [a-z]*.s
```

passes the argument "[a-z]*.s" to the shell where it is expanded to a list of the corresponding filenames before being passed to the program.

The **:r** and **:R** commands remove the contents of all registers and destroy the current stack before starting the program. This kills any previous copy of the program you may have been running.

Setting Breakpoints

You can set a breakpoint in a program by using the **:br** command. Breakpoints cause execution of the program to stop when it reaches the specified address. Control then returns to **adb**. The command has the form:

```
address [, count ] : command
```

where *address* must be a valid instruction address, *count* is a count of the number of times you wish the breakpoint to be skipped before it causes the program to stop, and *command* is the **adb** command you wish to execute when the breakpoint is taken.

Breakpoints are typically set to stop program execution at a specific place in the program, such as the beginning of a function, so that the contents of registers and memory can be examined. For example, the command:

```
main:br
```

sets a breakpoint at the start of the function named main. The breakpoint is taken just as control enters the function and before the function's stack frame is created.

A breakpoint with a count is 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.

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 wish to continue execution, *count* is the number of breakpoints you wish to ignore, and *signal* is the number of the signal to send to the program (see *signal (S)* in the IBM Personal Computer *XENIX Software Command 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.

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**, The key are especially useful for programs that have infinite loops or other program errors.

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

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

```
:co  0
```

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

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 wish to execute, and *count* is the number of times you wish to repeat the command.

If no *address* is given, **adb** uses the current address. If a *count* is given, **adb** continues to execute each successive instruction until *count* instructions have been executed. For example, the command

```
main,5:s
```

executes the first 5 instructions in the function *main*.

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.

Deleting Breakpoints

You can delete a breakpoint from a program by using the **:dl** command. The command has the form:

```
address :dl
```

where *address* is the address of the breakpoint you wish to delete.

The **:dl** command deletes breakpoints you no longer wish to use. The following command deletes the breakpoint set at the start of the function "main".

```
main:dl
```

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, as well as the address from which each function was called and the arguments passed to it.

For example, the command:

```
$c
```

displays a backtrace of the C language functions called.

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

```
  ,25$c
```

displays upto 25 calls in the current call path.

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 remedy 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
di      0x0                 ss      0x0
si      0x0                 es      0x0
sp      0x0                 sp      0x0
0:0:    addb      al,bl
```

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 to quickly summarize their values. The command displays one name on each line with the variable's value (if any) on the same line.

The display has the form:

```
fac:        0
_errno:     0
_end:       0
__sobuf:    0
_obuf:      0
__lastbu:   0406
__sibuf:    0
__stkmax:   0
Iscadr:     02
__iob:      01664
_edata:     0
```

An Example: Tracing Multiple Functions

The following example illustrates how to execute a program under **adb** control. In particular, it shows how to set breakpoints, start the program, and examine registers and memory. The program to be examined has the following source statements.

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

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

f(a,b)
{
        int fi; register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++ ;
        fj:
        g(fr,fi);
}

main()
{
        f(1,1);
}
```

The program is compiled and stored in 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.:  push    di
_f+4.:  push    si
_f+5.:  call    chkstk
_f+8.:
```

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
        push    di
        push    si
        call    chkstk
```

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 commands shows that only two functions are active: "main" and "f".

```
_f (1.,1.)  from_main+6.
_main (1.,470.)  from_start+114.
```

Although the breakpoint has been set at the start of 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 debugger 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+48.
_f (1.,1.)  from_main+6.
_main (1.,470.) from_start+114.
```

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

```
fcnt/d
```

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

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

```
,10:co
```

The **adb** program debugger starts the program and displays the running message again. It does not stop the program until exactly ten 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+46:
_h (10., 9.)     from_g+48:
_g (11., 20.)    from_f+48:
_f (2., 9.)      from_h+46:
_h (8., 7.)      from_g+48:
_g (9., 16.)     from_f+48:
_f (2., 7.)      from_h+46:
_h (6., 5.)      from_g+48:
_g (7., 12.)     from_f+48:
_f (2., 5.)      from_h+46:
_h (4., 3.)      from_g+48:
_g (5., 8.)      from_f+48:
_f (2., 3.)      from_h+46:
_h (2., 1.)      from_g+48:
```

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

If you have executed the program, the command 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 than the size of the segment in the file and often changes as you execute the program. This is due to expansion of the stack or allocation of additional memory during program execution. The filenames to the right always name program file. The file position value is ignored.

If you give a segment number with the command, **adb** displays information only about that segment. For example, the command:

```
$m 63
```

displays a map for segment 63 only. The display has the form:

```
Segment #= 63.
Type= Text
File position= 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* gives the number of the segment map you wish to change, *file-position* gives the offset in the file to the beginning of the given address, and *size* gives the segment size in bytes. The **?m** assigns values to a text segment entry; **/m** to a data segment entry.

For example, the following command changes the file position for segment 63 in the text map to 0x2000:

```
?m 63 0x2000
```

The command

```
/m 39 0x0
```

changes the file position for segment 39 in the data map to 0.

Creating New Map Entries

You can create new segment maps and add them to your memory map by using the **?M** and **/M** commands. Unlike **?m** and **/m**, these commands create a new map instead of changing an existing one. These commands have the form:

```
?M segment-number file-position size
```

and

```
/M segment-number file-position size
```

where *segment-number* gives the number of the segment map you wish to create, *file-position* gives the offset in the file to the beginning of the given address, and *size* gives the segment size in bytes. The **?M** command creates a text segment entry; **/M** creates a data segment entry. The segment number must be unique. You cannot create a new map entry that has the same number as an existing one.

The **?M** and **/M** commands are especially useful for accessing segments that are allocated to your program. For example, the command:

```
?M 71 0 2504
```

creates a text segment entry for segment 71 whose size is 2504 bytes.

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 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
```

search for and display 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 filename. For example, to read commands from the file *script*, type:

```
adb sample <script
```

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

Reading commands from a script file is very convenient when you wish to use the same set of commands on several different object files. Scripts 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
=3"C Stack Backtrace"
$C
=3"C External Variables"
$e
=3"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 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 accomodate 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 wish to work with a combination of decimal, octal, and hexadecimal numbers. For example, if you use:

```
$x
```

you may give addresses in hexadecimal without prepending each address with "0x". Furthermore, **adb** displays all numbers in hexadecimal 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 XENIX Commands

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

```
! command
```

where *command* is the XENIX command you wish to execute. The command must have any required arguments. 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 = command. The command directs **adb** to display the value of an expression in a given format.

The command converts1 numbers in one base to another, to double check the arithmetic performed by a program, and to display complex addresses in easier form. For example, the command:

```
0x2a=d
```

displays the hexadecimal number 0x2a as the decimal number 42 but:

```
0x2a=c
```

displays it as the ASCII character (*). Expressions in a command may have any combination of symbols and operators. For example, the command:

```
<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 XENIX file system. The directory file is assumed to be named *dir* and contains a variety of files. The XENIX file system is assumed to be associated with the device file */dev/src* and has the necessary permissions to be read by the user.

To display a directory file, you must create an appropriate script, then start **adb** with the name of the directory, redirecting its input to the script.

First, you can create a script file named *script.* A directory file normally contains one or more entries. Each entry consists of an unsigned “inumber” and a 14-character filename. You can display this information by adding the command:

```
0,-1?ut14cn
```

to the script file. This command displays one entry for each line, separating the number and filename with a tab. The display continues to the end of the file. If you place the command

```
="number"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, 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 filename of the device associated with the file system (for example, the fixed disk drive).

The inode table of a file system has a very complex structure. Each entry contains: a word value for the file's status flags; a byte value for the number links; two byte values for the user and group IDs; a byte and word value for the size; eight word values for the location on disk of the file's blocks; and two word values for the creation and modification dates. The inode table starts at 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 hypen (-) 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.

The

```
?l
```

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

```
?l 'Th'
```

```
?W 'The'
```

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

Chapter 8. The lex Program: A Lexical Analyzer

Contents

Introduction

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

The user supplies the additional code needed to complete the tasks, including code written by other generators. The program that recognizes the expressions is generated in the from the user's C program fragments. The **lex** program is not a complete language, but rather 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 *yylex* program recognizes expressions in a stream (called input in this chapter) and performs the specified actions for each expression as it is detected.

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

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

are all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression that matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C-language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates one or more of the previous item; and the dollar sign ($) indicates the end of the line. No action is specified, so the program generated by **lex** 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, observes at the termination of the string of blanks or tabs whether or not there is a newline character, and then executes the desired rule's action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule matches all remaining strings of blanks or tabs.

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**. 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 control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

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

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" later in this chapter.

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:

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

If any of these 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 by quoting every nonalphanumeric character used as a text character, you need not memorize the above list of current operator characters.

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

```
xyz\+\+
```

that is another, less readable, equivalent of the above expressions. 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; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Invoking lex

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

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

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

The resulting program is placed on the usual file *a.out* for later execution. To use **lex** with **yacc** see the section, "The Programs lex and yacc" later in this chapter. Also, refer to Chapter 9. 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.

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: these are the backslash (\), the dash (-), and the caret (^). The dash character indicates ranges. For example:

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

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

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

matches all the digits and the plus and minus signs.

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

```
[^abc]
```

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

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

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

Specifying an Arbitrary Character

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

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

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

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 for recognizing identifiers in computer languages.

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 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 can never conflict with the other meaning of the caret, complementation of character classes, since complementation only applies within brackets. If the very last character is 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 the section "Specifying Left Context Sensitivity". If a rule is only to be executed when the **lex** automaton interpreter is in start condition *x*, the rule should be enclosed in angle brackets:

```
<x>
```

If the beginning of a line starts condition ONE, then the caret (^) operator is equivalent to:

```
<ONE>
```

Start conditions are explained more in the section, "Specifying Left Context Sensitivity."

Specifying Expression Repetition

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

```
{digit}
```

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

Specifying Definitions

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

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

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

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

Specifying Actions

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

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

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

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

Another easy way to avoid writing actions is to use the repeat action character; | this indicates that the action for this rule is the action for the next rule. The previous example is also written as:

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

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

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

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

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

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

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

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

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

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

is needed. This is explained further below.

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

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

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

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

Occasionally, a **lex** action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore ()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string 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 provides the same sort of lookahead offered by the slash (/) operator, but in a different form.

For example consider a language that defines a string as a set of characters between quotation marks ("), and 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[yyleng-1] == '\\')
             yymore();
         else
             ... normal user processing
         }
```

and, when faced with a string such as:

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

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 reprocess text in various circumstances. Consider the problem in the older C syntax of distinguishing the ambiguity of =-*a*. Suppose it is desired to treat this as =- *a* and to print a message. A rule might be:

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

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

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

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

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:

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

a still better rule.

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

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

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

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

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They can be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end-of-file; and the relationship between *unput* and *input* must be retained or the lookahead 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 lookahead:

```
+   *   ?   $
```

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

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

This routine is also a convenient place to print 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, since 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 can match 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:

```
 '.*'
```

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 defeat this 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 once and only once. For example, suppose it is desired 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. 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 the user would like to override this choice. The action REJECT means go do the next alternative. It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of *he*:

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

These rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression are then be counted. In this example, of course, the user could note that *she* includes *he*, but not vice versa, and omit the REJECT action on *he*; in other cases, 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, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word *the* is considered to contain both *th* and *he* . Assuming a two-dimensional array named *digram* 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 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, recognizing immediately preceding left context just as the dollar sign ($) recognizes immediately following right context. Adjacent left context could be extended to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments:

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

- The use of start conditions with rules

- The use multiple lexical analyzers running together

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

Consider the following problem:

- Copy the input to the output.
- Change the word *magic* to *first* on every line that begins with the letter *a*.
- Change *magic* to *second* on every line that begins with the letter *b*.
- Change *magic* to *third* on every line that begins with the letter *c*.
- Leave all other words and all other lines unchanged.

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

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

should be adequate.

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

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

where the conditions 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

```
BEGIN 0;
```

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

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

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

The same example as before can 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, though, to define variables for use in your program and for use by **lex**. These can go either in the definitions section or in the rules section.

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

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

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

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

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

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

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least 1 blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

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

The first two rules for real numbers are require that have a decimal point and contain an optional exponent field, but the first requires at least 1 digit before the decimal point and the second requires at least 1 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:

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

could be used 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 the section "Source Format" later in this chapter.

The Programs lex and yacc

To use **lex** with **yacc** note that what **lex** writes is a program named *yylex*(), the name required by **yacc** for its analyzer. Normally, the default main program on the **lex** library calls this routine, but if **yacc** is loaded, and its main program is used, **yacc** calls *yylex*(). In this case, each **lex** rule should end with:

```
return(token);
```

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

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

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

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

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

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

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

The rule [0-9]+ recognizes strings of digits; *atoi*() 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, as here:

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

Numerical strings containing a decimal point or preceded by a letter 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 statistics gathering, 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[yyleng]++;
.       |
\n      ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
     if (lengs[i] > 0)
          printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

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

As a larger example, here are some parts of a program written to convert double-precision FORTRAN to single precision FORTRAN. Because FORTRAN does not distinguish between 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:

```
^"     "[^ 0]   ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Two different meanings of the caret (^) are used here. There follow some rules to change double-precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}&
1br.0-9]+         |
[0-9]+{W}"."{W}{d}{W}[-]?{W}[0-9]+      |
"."{W}[0-9]+{W}{d}{W}[-]?{W}[0-9]+      {
     /* convert constants */
     for(p=yytext; *p != 0; p++)
          {
          if (*p == 'd' || *p == 'D')
               *p+= 'e'- 'd';
                ECHO;
          }
```

After the floating-point constant is recognized, it is scanned by the **for** loop to find the letter "d" or "D". The program then adds "'e'-'d'" and then converts it 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}1{m}{a}{c}{h}          {
        yytext[0] += 'r'  - 'd';
        ECHO;
}
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+                  |
\n                      |
.       ECHO;
```

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

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 giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only *%T*. The table contains lines of the form:

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

that indicate the value associated with each character. For example:

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

This table maps the lowercase and uppercase letters together into the integers 1 through 26, 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.

Source Format

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

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

The definitions section contains a combination of:

1. Definitions, in the form "name space translation"

2. Included code, in the form "space code"

3. Included code, in the form

```
%{
code
%}
```

4. Start conditions, given in the form

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

5. Character set tables, in the form

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

6. Changes to internal array sizes, in the form

```
%x  nnn
```

where *nnn* is a decimal integer representing an array size. and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	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:

x	The character "x"
"x"	An "x", even if x is an operator.
\x	An "x", even if x is an operator.

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

Chapter 9. The yacc Program Generator: A Compiler-Compiler

Contents

Introduction

Computer program input generally has some structure; every computer program that does input can be thought of as defining an input language that it accepts. An input language can be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often lax about checking their inputs for validity.

The **yacc** program generator provides a general tool for describing the input to a computer program. The name **yacc** itself stands for "yet another compiler-compiler". The **yacc** user specifies the structures of his input, together with code to be invoked as each such structure is recognized. The **yacc** program generator turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

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

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

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 includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do 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. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

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

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be:

```
date : month_name day  ','  year   ;
```

Here, *date*, *month__name*, *day*, and *year* represent structures of interest in the input process; presumably, *month__name*, *day*, and *year* are defined elsewhere. The comma (,) is enclosed in single quotation marks; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input:

```
July  4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. A structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules:

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

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond **yacc 's** ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

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

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

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

allowing

```
7/4/1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be slipped in to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. For example, the specifications can be self contradictory, or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions that are difficult for **yacc** to handle are also frequently difficult for human beings 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
- Various reasons why **yacc** may be unable to produce a parser from a specification, and what to do about it.
- A simple mechanism for handling operator precedences (order of arithmetic operation) in arthmetic operations.
- Error detection and recovery.
- The operating environment and special features of the parsers **yacc** produces.
- Some suggestions that should 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. In addition, for reasons discussed later, it is often desirable to include the lexical analyzer as part of the specification file. It can be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent %% marks. The percent sign (%) is generally used in **yacc** specifications as an escape character.

In other words, a full specification file looks like

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

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark can be omitted also; thus, the smallest legal **yacc** specification is:

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multicharacter reserved symbols. Comments appear wherever a name is legal; they are enclosed in /* . . . */, as in C.

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

```
A : BODY ;
```

A represents a nonterminal name, and *BODY* represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names are of arbitrary length, and may be made up of letters, dot (.), the underscore (__), and noninitial 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 ('). As in C, the backslash (\) is an escape character within literals, and all the C escapes are recognized.

Thus:

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

For a number of technical reasons, never use the ASCII NUL character ('\0' or 0) in grammar rules.

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

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

can be given to **yacc** as:

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

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty : ;
```

Names representing tokens must be declared; this is most simply done by writing

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

in the declarations section. (Following sections discuss this in more detail.) Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start symbol
```

The end of the input to the parser is signaled by a special token, called the endmarker. If the tokens up to, but not including, the endmarker form a structure matching the start symbol, the parser function returns to its caller after the endmarker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see the section, "Actions" next in this chapter. Usually the endmarker represents some reasonably obvious I/O status, such as the end of the file or end of the record.

Actions

With each grammar rule, the user associates actions performed each time the rule is recognized in the input process. These actions may 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 an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces { and }. For example:

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

and

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

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The dollar sign ($) is used as a signal to **yacc** in this context.

To return a value, the action normally sets the pseudo-variable $$ to some value. For example, an action that does nothing but return the value 1 is:

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

To obtain the values returned by previous actions and the lexical analyzer, the action uses the pseudo-variables $1, $2, . . . , that 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 ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule:

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

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr : '(' expr ')' { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form:

```
A : B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. The **yacc** program generator permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual $ mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule:

```
A : B
    {  $$ = 1;  }
    C
    {   x = $2;    y = $3;  }
    ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by **yacc** by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. The **yacc** program generator actually treats the above example as if it had been written:

```
$ACT : /* empty */
                    {  $$ = 1;  }
     ;

A    : B  $ACT  C
                    {   x = $2;   y = $3;  }
     ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call:

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr : expr '+' expr
                      {  $$ = node( '+', $1, $3 );  }
```

in the specification.

The user can define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks *%{* and *%}*. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{ int variable = 0; %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The **yacc** parser uses only names beginning in *yy*, therefore, the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in a later section.

Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, called the token number, representing the kind of token read. If a value is associated with that token, it should be assigned to the external variable *yylval.*

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers are chosen either by **yacc**, or the user. In either case, the # *define* mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name *DIGIT* has been defined in the declarations section of the **yacc** specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
         . . .
        c = getchar();
         . . .
        switch( c ) {
                 . . .
        case '0':
        case '1':
         . . .
        case '9':
                yylval = c-'0';
                return( DIGIT );
                 . . .
                }
         . . .
```

The intent is to return a token number of *DIGIT*, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the 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; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should not be used naively.

As mentioned above, the token numbers 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), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user. Hence, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is **lex**, discussed in a previous section. These lexical analyzers are designed to work in close harmony with **yacc** parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. 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 crafted by hand.

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 (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by **yacc** consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and 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 decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This 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 is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF     shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. You may have to consult the lookahead token to decide whether to reduce, but usually it is not; in fact 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 hand side (three in this case). To reduce, first pop off the top three states from the stack (in general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered that was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a "goto" action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A       goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action turns back the clock in the parse, popping the states off the stack to go back to the state where the right-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off of the stack; the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After 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 other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing. Error recovery (as opposed to the detection of error) is discussed later in this chapter.

Consider the following example:

```
%token  DING DONG DELL
%%
rhyme : sound  place
      ;
sound : DING  DONG
      ;
place : DELL
      ;
```

When **yacc** is invoked with the **-v** option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
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 seen, and what is yet to come, in each rule. Suppose the input is:

```
DING DONG DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is *shift 3*, so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is *shift 6*, so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound : DING  DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
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 it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by *$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 probably repay you when problems arise in more complicated contexts.

Ambiguity and Conflicts

A set of grammar rules is ambiguous if some 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. It is instructive to consider the problem that confronts the parser when it is given an input such as:

```
expr - expr - expr
```

When the parser has read the second expr, the input that it has seen:

```
expr - expr
```

matches the right side of the grammar rule above. 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, when the parser has seen:

```
expr - expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen:

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving:

```
expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read:

```
expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a shift/reduce conflict. It may also possible for the parser to have a choice of two legal reductions; 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 the user rather 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 action 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 is somewhat unnatural, and produces slower parsers; thus, **yacc** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an if-then-else construction:

```
stat : IF '(' cond ')' stat
     | IF '(' cond ')' stat ELSE stat
     ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the simple-if rule, and the second the if-else rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
          IF ( C2 ) S1
          }
ELSE S2
```

or

```
IF ( C1 ) {
          IF ( C2 ) S1
          ELSE S2
          }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last *IF* immediately preceding the *ELSE*. In this example, consider the situation where the parser has seen:

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get,

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce:

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to 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 - 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 ".", 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 release 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 really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references might be consulted; the services of a local expert might also be appropriate.

Precedence

One common situation where the rules given above for resolving conflicts are not sufficient is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form:

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules, and 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 star and slash, that are also left associative. The keyword %right describes right associative operators, and the keyword %nonassoc describes operators, like the operator .LT. in FORTRAN, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in FORTRAN, and such an operator would be described with the keyword %nonassoc in **yacc**. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr  : expr '=' expr
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | NAME
      ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; 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, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/'

%%

expr  : expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | '-' expr %prec '*'
      | NAME
      ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %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 nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by **yacc**. This means that mistakes in the specification of precedences 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 was 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. This leads to the problem of getting the parser restarted after an error. A general class of algorithms to perform this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, **yacc** provides a simple, but reasonably general feature. The token name *error* is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token *error* is legal. It then behaves as if the token *error* were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

To prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is deleted.

As an example, a rule of the form

```
stat : error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement where the error was seen. More precisely, the parser scans ahead, looking for three tokens that legally follow a statement, and starts processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it can make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions can be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as:

```
stat : error ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule 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 admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, a mechanism can be used to force the parser to believe that an error has been fully recovered from. The statement:

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written:

```
input : error '\n'
                   { yyerrok;
                     printf( "Reenter last line: " ); }
                   input
                   { $$ = $4; }
                   ;
```

As mentioned above, the token seen immediately after the *error* symbol is the input token where the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like:

```
stat : error
              { resynch();
                yyerrok ;
                yyclearin ; }
              ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

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 it 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.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **yacc**, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a **-ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to *yyerror* is a string containing an error message, usually the string *syntax error.* The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error 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 in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser outputs a verbose description of its actions, including a discussion of the input symbols that have been read and what the parser actions are. Depending on the operating environment, you can possibly set this variable by using a debugging system.

Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file.

1. Use uppercase letters for token names, lowercase letters for nonterminal names. This rule helps you to know where to place the blame when things go wrong.

2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

3. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

4. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

5. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The examples in the text of this section follow this style (where space permits). The user must decide about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the **yacc** parser encourages so called left recursive grammar rules: rules of the form

```
name : name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list : item
     | list ',' item
     ;
```

and

```
seq : item
    | seq item
    ;
```

In each of these cases, the first rule 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 a bit bigger, and the items are seen, and reduced, from right to left. More seriously, an internal stack in the parser is in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq : /* empty */
    | seq item
    ;
```

Once again, the first rule 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 asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names are entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
  int dflag;
%}
   . . .   other declarations  . . .

%%

prog     : decls  stats
         ;

decls    : /* empty */
                {        dflag = 1;   }
         | decls declaration
         ;

stats    : /* empty */
                {        dflag = 0;   }
         | stats statement
         ;

     . . .   other rules  . . .
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, except for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of back door approach can be overdone. Nevertheless, it represents a way of doing some things that are difficult to do otherwise.

Handling Reserved Words

Some programming languages permit the user to use words like *if*, which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc** ; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable." The user can make a stab at it, but it is difficult. It is best that keywords be reserved; that is, be forbidden for use as variable names.

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*. *YYACCEPT* causes *yyparse* to return the value 0; *YYERROR* causes the parser to behave as if the current input symbol had been a syntax error; *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 simply the same as with ordinary actions, 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*. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism saves a great deal of trouble, especially 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 know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where **yacc** cannot easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
       body of union  . . .
       }
```

This declares the **yacc** value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If **yacc** was invoked with the **-d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union 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, by use of %{ and %}.

Once *YYSTYPE* is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, 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, similarly to associates union member names with nonterminals. Thus, one might say:

```
%type  <nodetype> expr stat
```

A couple of cases remain where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no predefined type. Similarly, reference to left context values (such as $0 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 ); }
     ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in a later section. The facilities in this subsection are not triggered until used: in particular, the use of %type turns on 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 was true historically.

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 has a description of 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 sticky part comes when an identifier is seen in a rule immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token

C_IDENTIFIER. Otherwise, it returns *IDENTIFIER*. Literals (quoted strings) are also returned as *IDENTIFIER*, but never as part of *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 using some of the advanced features discussed in earlier sections. The desk calculator example is modified to provide a desk calculator that does floating-point interval arithmetic. The calculator understands floating-point constants, the arithmetic operations +, -, *, /, unary -, and = (assignment), and has 26 floating-point variables, *a* through *z*. Moreover it also understands intervals, written:

```
( 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 endpoint 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.

It is also worth noting the use of *YYERROR* to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of **yacc** throws away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (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 when the grammar is run through **yacc**: 18 Shift/Reduce and 26 Reduce/Reduce. 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 thence error recovery.

```
%{

#  include  <stdio.h>
#  include  <ctype.h>

typedef  struct  interval  {
        double  lo,  hi;
        }  INTERVAL;

INTERVAL  vmul(),  vdiv();

double    atof();

double    dreg[ 26 ];
INTERVAL  vreg[ 26 ];

%}

%start    lines

%union  {
        int  ival;
        double  dval;
        INTERVAL  vval;
        }

%token  <ival>  DREG  VREG   /* indices into dreg, vreg arrays*/

%token  <dval>  CONST        /* 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   : /*  empty  */
        | lines  line
        ;

line    : dexp '\n'
                { printf( "%15.8f\n", $1 ); }
        | vexp  '\n'
                { printf( "(%15.8f, %15.8f )\n",
                                $1.lo, $1.hi ); }
        | DREG  '=' dexp '\n'
                { dreg[$1] = $3; }
        | VREG  '='  vexp  '\n'
                { vreg[$1] = $3; }
        | error '\n'
                { yyerrok; }
        ;

dexp    : CONST
        | DREG
                { $$ = dreg[$1]; }
        | dexp '+' dexp
                { $$ = $1 + $3; }
        | dexp '-' dexp
                { $$ = $1 - $3; }
        | dexp '*' dexp
                { $$ = $1 * $3; }
        | dexp '/' dexp
                { $$ = $1 / $3; }
        | '-' dexp %prec UMINUS
                { $$ = - $2; }
        | '(' dexp ')'
                { $$ = $2; }
        ;
```

```
vexp  : dexp
              { $$.hi = $$.lo = $1; }
      | '(' dexp ',' dexp ')'
              {
                $$.lo  =  $2;
                $$.hi  =  $4;
                if(  $$.lo  >  $$.hi  ){
                        printf("interval out oforder \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;
                    $$.lo. = $1 - $3.;}
      | vexp '*' vexp
              { $$ = vmul( $1.lo, $1.hi, $3 ); }
      | dexp '*' vexp
              { $$ = vmul( $1, $1, $3 ); }
      | vexp '/' vexp
              { if ( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1.lo, $1.hi, $3 ); }
```

```
        | dexp '/' vexp
                { if ( dcheck( $3 ) ) YYERROR;
                  $$ = vdiv( $1, $1, $3 ); }
        | '-' vexp  %prec UMINUS
                { $$.hi = -$2.lo.; $$.lo. = -$2.hi; }
        | '(' vexp ')'
                {      $$  =  $2;  }
        ;
%%

#  define  BSZ  50   /*  buffer  size  for  fp numbers  */

        /*  lexical  analysis  */

yylex(){
        register  c;
        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  ungetc( c, stdin );
               /* above pushes back last char read */
        yylval.dval = atof ( buf );
        return( CONST );
        }
  return( c );
  }
```

```
INTERVAL  hilo( a, b, c, d )  double  a, b, c, d; {
        /* returns the smallest interval containing a, b, c, and  d */
        /* used by *, / routines */
        INTERVAL v;

        if( a>b ) { v.hi = a; v.lo = b; }
        else { v.hi = b; v.lo = a; }
        if( c>d ) {
                if ( c>v.hi ) v.hi = c;
                if ( d<v.lo ) v.lo = d;
                }
        else  {
                if ( d>v.hi) v.hi = d;
                if ( c<v.lo ) v.lo = c;
                }
        return(  v  );
        }

INTERVAL vmul( a, b, v ) double a, b;  INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) ) ;
        }


dcheck( v ) INTERVAL v; {
        if( v.hi >= 0. && v.lo <= 0. ){
                printf( "divisor interval contains 0.\n" );
                return(1);
                }
        return(0);
        }

INTERVAL  vdiv( a, b, v )  double a, b;  INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi,/v.lo ) );
}
```

Old Features

This section mentions synonyms and features supported for historical continuity, but, for various reasons, not encouraged.

1. Literals can also be delimited by double quotation marks (").

2. Literals can be more than 1 character long. If all the characters are alphabetic, numeric, or underscore, the type number of the literal is defined, just as if the literal did not have the quotation marks around it. Otherwise, it is difficult to find the value for such literals. The use of multicharacter literals is likely to mislead those unfamiliar with **yacc**, since it suggests that **yacc** is doing a job that must actually be done by the lexical analyzer.

3. Most places where percent (%) is legal, backslash (\) can be used. In particular, the double backslash (\\) is the same as *%%*, \left the same as *%left*, etc.

4. There are a number of other synonyms:

   ```
   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec
   ```

5. Actions also have the form:

   ```
   ={  . . .  }
   ```

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Chapter 10. The C Shell

Contents

Introduction

The C shell program, *csh*, is a command language interpreter for XENIX system users. The C shell, like the standard XENIX shell *sh*, is an interface between you and the XENIX commands and programs. It translates command lines typed at a terminal into corresponding system actions, gives access to information, such as your login name, home directory, and mailbox, and lets you construct shell procedures for automating system tasks.

This chapter explains how to use the C shell. It also explains the syntax and function of C shell commands and features, and shows how to use these features to create shell procedures. The C shell is fully described in *csh (CP)* in the IBM Personal Computer XENIX *Software Command Reference.*

Invoking the C Shell

To invoke the C shell from another shell, use the **csh** command. Type:

```
csh
```

at the standard shell's command line. To direct the system to invoke the C shell when you log in, give the C shell as your login shell in your */etc/passwd* file entry, and the system automatically starts the shell when you log in.

After the system starts the C shell, the shell the home directory for the command files *.cshrc* and *.login* . If the shell finds the files, it executes the commands contained in them, then displays the C shell prompt.

The *cshrc* file contains the commands you wish to execute each time you start a C shell, and the *.login* file contains the commands you wish to execute after logging in to the system. For example, the following is the contents of a typical *.login* file:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
set time=15
set history=10
mail
```

This file contains several **set** commands. The **set** command is executed directly by the C shell; there is no corresponding XENIX program for this command. The **set** command sets the C shell variable "ignoreeof" that shields the C shell from logging out if Ctrl-D is hit. Instead of Ctrl-D , the **logout** command is used to log out of the system. By setting the "mail" variable, the C shell is notified that it is to watch for incoming mail and notify you if new mail arrives.

Next the C shell variable "time" is set to 15 causing the C shell to automatically print out statistics lines for commands that execute for at least 15 seconds of CPU time. The variable "history" is set to 10 indicating that the C shell will remember the last 10 commands typed in its history list, (described later).

Finally, the XENIX *mail* program is invoked.

When the C shell finishes processing the *.login* file, it begins reading commands from the terminal, prompting for each with:

```
%
```

When you log out (by giving the **logout** command) the C shell prints

```
logout
```

and executes commands from the file *logout* if it exists in the home directory. After that, the C shell terminates and the XENIX system logs you off.

Using Shell Variables

The C shell maintains a set of variables. For example, in the above discussion, the variables "history" and "time" had the values 10 and 15. Each C shell variable has as its value an array of zero or more strings. C shell variables are assigned values by the **set** command, which has several forms. The most useful is:

```
set name=value
```

C shell variables store values that are used later in commands through a substitution mechanism. The C shell variables most commonly referenced are, however, those that the C shell itself refers to. By changing the values of these variables you can directly affect the behavior of the C shell.

One of the most important variables is "path." This variable contains a list of directory names. When you type a command name at your terminal, the C shell examines each named directory in turn, until it finds an executable file whose name corresponds to the name you typed. The **set** command with no arguments displays the values of all variables currently defined in the C shell. The following example shows a typical default values:

```
argv    ()
home    /usr/bill
path    (. /bin /usr/bin)
prompt  %
shell   /bin/csh
status  0
```

This output indicates that the variable "path" begins with the current directory indicated by dot (.), then */bin*, and */usr/bin.* Your own local commands can be in the current directory. Normal XENIX commands reside in */bin* and */usr/bin.*

Sometimes a number of locally developed programs reside in the directory */usr/local.* If you want all C shells that you invoke to have access to these new programs, place the command

```
set path=(. /bin /usr/bin /usr/local)
```

in the *.cshrc* file in the home directory. Try doing this, then logging out and back in. Type:

```
set
```

to see that the value assigned to "path" has changed.

When you log in the C shell examines each directory that you insert into the path and determines which commands are contained there, except for the current directory which the C shell treats specially. This means that if commands are added to a directory in your search path after you have started the C shell, they will not necessarily be found. To use a command added after you have logged in, you should give the command:

```
rehash
```

to the C shell. The **rehash** command causes the shell to recompute its internal table of command locations, so that it will find the newly added command. Since the C shell has to look in the current directory on each command anyway, placing it at the end of the path specification usually works best and reduces overhead.

Other useful built in variables are "home" that shows the home directory, and "ignoreeof" that can be set in your *.login* file to tell the C shell not to exit when it receives an end-of-file from a terminal. The variable "ignoreeof" is one of several variables whose value the C shell does not care about; the C shell is only concerned with whether these variables are set or unset. Thus, to set "ignoreeof" simply type:

```
set ignoreeof
```

and to unset it type

```
unset ignoreeof
```

Some other useful built-in C shell variables are "noclobber" and "mail". The syntax:

```
>filename
```

that redirects the standard output of a command just as in the regular shell, overwrites and destroys the previous contents of the named file. In this way, you may accidentally overwrite a valuable file. To avoid having the C shell overwrite files in this way, you can:

```
set noclobber
```

in your *.login* file. Then typing:

```
date > now
```

causes an error message if the file *now* already exists. You can type:

```
date >! now
```

to overwrite the contents of *now*. The >! is a special syntax indicating that overwriting or "clobbering" the file is all right. The space between the exclamation point (!) and the word "now" is critical here, as "!now" would be an invocation of the history mechanism, described below, and have a totally different effect.

Using the C Shell History List

The C shell can maintain a history list into which it places the text of previous commands. It is possible to use a notation that reuses commands, or words from commands, in forming new commands. This mechanism repeats previous commands or corrects minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the C shell. Boldface indicates user input:

```
 % cat bug.c
main()
{
	printf("hello);
}
```

```
%cc !$
cc bug.c
bug.c(4) : warning : Newline in string constant
bug.c(4) : syntax error: '}'
% ed !$
ed bug.c
29
3s/);/"&/p
     printf("hello");
w
30
q
% !c
cc bug.c
bug.c
% a.out
hello % !e
ed bug.c
30
3s/lo/lo\\n/p
     printf("hello\n");
w
32
q
% !c -o bug
cc bug.c -o bug
bug.c
% size a.out bug
a.out: 4176 + 496 + 1072 = 5744 = 0x1670
bug.c 4176 + 496 + 1072 = 5744 = 0x1670
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill     5898 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill     5898 Dec 19 09:42 bug
% bug
hello
 % pr bug.c | lpt
lpt: Command not found.
%^lpt^lpr
pr bug.c | lpr
%
```

This example demonstrates a very simple C program with a bug or two in the file *bug.c*, that we **cat** out on our terminal. We then try to run the C compiler on it, referring to the file again as "!$", meaning the last argument to the previous command. Here the exclamation mark (!) is the history mechanism invocation metacharacter, and the dollar sign ($) stands for the last argument, by analogy to the dollar sign in the editor that stands for the end-of-line. The C shell echoed the command, as it would have been typed without use of the history mechanism, and then executed the command. The compilation yielded error diagnostics, so we now edit the file we are trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as "!c", which repeats the last command that started with the letter "c". If there were other commands beginning with the letter "c" executed recently, we could have said "!cc" or even "!cc:p" which prints the last command starting with "cc" without executing it, so that you can check to see whether you really want to execute a given command.

After this recompilation, we ran the resulting *a.out* file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra "-o bug" telling the compiler to place the resultant binary in the file *bug* rather than *a.out.* In general, the history mechanisms are used anywhere in the formation of new commands, and other characters are placed before and after the substituted commands.

We then ran the **size** command to see how large the binary program images we have created were, and then we ran an "ls -l" command with the same argument list, denoting the argument list:

!*

Finally, we ran the program *bug* to see that its output is indeed correct.

To make a listing of the program, we ran the **pr** command on the file *bug.c*. To print the listing at a line printer, we piped the output to **lpr** , but misspelled it as "lpt". To correct this we used a C shell substitute, placing the old text and new text between caret (^) characters. This is similar to the substitute command in the editor.

Other mechanisms are available for repeating commands. The **history** command prints out a numbered list of previous commands. You can then refer to these commands by number. There is a way to refer to a previous command by searching for a string that appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in *csh (CP)* the IBM Personal Computer XENIX *Software Command Reference.*

Using Aliases

The C shell has an alias mechanism that can make transformations on commands immediately after you input them. This mechanism:

- Simplifies the commands you type
- Supplies default arguments to commands

  or

- Performs transformations on commands and their arguments.

The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using C shell command files, but these take place in another instance of the C shell and cannot directly affect the current C shell's environment or involve commands such as **cd** which must be done in the current C shell.

For example, suppose a new version of the mail program is on the system called *newmail,* and you wish to use this instead of the standard mail program *mail.* If you place the C shell command:

```
alias mail newmail
```

in your *.cshrc* file, the C shell transforms an input line of the form:

```
mail bill
```

into a call on *newmail.* Suppose you wish the command **ls** to always show sizes of files, that is, to always use the **-s** option. In this case, use the **alias** command to do:

```
alias ls ls -s
```

or even:

```
alias dir ls -s
```

creating a new command named **dir**. Then type:

```
dir ~bill
```

the C shell translates this to:

```
ls -s /usr/bill
```

The tilde (~) is a special C shell symbol that represents the user's home directory.

Thus the **alias** command can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases that contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition:

```
alias cd 'cd \!* ; ls '
```

specifies an **ls** command after each **cd** command. The entire alias definition is enclosed in single quotation marks (') to prevent most substitutions from occurring and to prevent the semicolon (;) from being recognized as a metacharacter. The exclamation mark (!) is escaped with a backslash (\) to prevent it from being interpreted when the alias command is typed in. The "\!*" here substitutes the entire argument list to the prealiasing **cd** command; no error is given if there are no arguments. The semicolon separating commands indicates one command is to be done and then the next is to be done. Similarly the following example defines a command that looks up its first argument in the password file.

```
alias whois 'grep \!^ /etc/passwd'
```

The C shell currently reads the *.cshrc* file each time it starts up. Try to limit the number of aliases you have; 10 or 15 is a reasonable number. Too many aliases cause system delays and sluggishness when you execute commands from within an editor or other programs. Also, C shells tend to start slowly.

Redirecting Input and Output

In addition to the standard output, commands also have a diagnostic output normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally useful to direct the diagnostic output along with the standard output. For instance, if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can type:

```
command >& file
```

The ">&" here tells the C shell to route both the diagnostic output and the standard output into *file*. Similarly you can give the command:

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the lineprinter. The form:

```
command >&! file
```

is used when "noclobber" is set and *file* already exists.

Finally, use the form:

```
command >> file
```

to append output to the end of an existing file. If "noclobber" is set, then an error results if *file* does not exist, otherwise the C shell appends the output to *file*. The form:

```
command >>! file
```

lets you append to a file even if it does not exist and “noclobber” is set.

Creating Background and Foreground Jobs

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single job is created by the C shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the C shell creates a job. Each of the following lines creates a job:

```
sort < data
ls -s | sort -n | head -5
mail harold
```

If the ampersand metacharacter (&) is typed at the end of the commands, then the job is started as a background job. This means that the C shell does not wait for the job to finish, but instead, immediately prompts for another command. The job runs in the background at the same time that normal jobs, called foreground jobs, continue to be read and executed by the C shell. Thus

```
du > usage &
```

runs the *du* program, that:

- Reports on the disk usage of your working directory
- Puts the output into the file *usage*

  and
- Returns immediately with a prompt for the next command without waiting for *du* to finish.

The *du* program continues executing in the background until it finishes, even though you can type and execute more commands in the meantime. Background jobs are unaffected by any signals from the keyboard such as the Interrupt (Del) or Quit (Ctrl \) signals.

The **kill** command terminates a background job immediately. Normally, this is done by specifying the process number of the job you want killed. Process numbers can be found with the **ps** command.

Using Built-In Commands

This section explains how to use some of the built-in C shell commands.

The **alias** command described above assigns new aliases and displays existing aliases. If given no arguments, **alias** prints the list of current aliases. You can also give it one argument, to show the current alias for a given string of characters. For example:

```
alias ls
```

prints the current alias for the string "ls".

The **history** command displays the contents of the history list. The numbers given with the history events reference previous events that are difficult to reference contextually. There is also a C shell variable named "prompt". By placing an exclamation point (!) in its value the C shell will substitute the number of the current command in the history list. You can use this number to refer to a command in a history substitution. For example, you could type:

```
set prompt='\! % '
```

The exclamation mark (!) had to be escaped here even within quotes.

The **logout** command terminates a login C shell that has "ignoreeof" set.

The **rehash** command causes the C shell to recompute a table of command locations. This is necessary if you add a command to a directory in the current C shell's search path and want the C shell to find it, since otherwise the hashing algorithm may tell the C shell that the command wasn't in that directory when the hash table was computed.

The **repeat** command is used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could type:

```
repeat 5 cat one >> five
```

The **setenv** command sets variables in the environment. Thus:

```
setenv TERM adm3a
```

sets the value of the environment variable "TERM" to "adm3a." The program *env* exists to print out the environment. For example, its output might look like this:

```
HOME=/usr/bill
SHELL=/bin/csh
PATH=/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
```

The **source** command forces the current C shell to read commands from a file. Thus:

```
source .cshrc
```

can be used after editing in a change to the *.cshrc* file that you wish to take effect before the next time you login.

The **time** command causes command to be timed no matter how much CPU time it takes. Thus:

```
time cp /etc/rc /usr/bill/rc
```

displays:

```
0.0u 0.1s 0:01 8%
```

Similarly

```
time wc /etc/rc /usr/bill/rc
```

displays:

```
     52     178    1347 /etc/rc
     52     178    1347 /usr/bill/rc
    104     356    2694 total
0.1u 0.1s 0:00 13%
```

This indicates that the **cp** command used a negligible amount of user time (u) and about 1/10th of a second system time (s); the elapsed time was 1 second (0:01). The word count command **wc** used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The "13%" indicates that over the period when it was active the **wc** command used an average of 13 percent of the available CPU cycles of the machine.

The **unalias** and **unset** commands remove aliases and variable definitions from the C shell. The command **unsetenv** removes variables from the environment.

Creating Command Scripts

It is possible to place commands in files and to cause C shells to be invoked to read and execute commands from these files, called C shell scripts. This section describes the C shell features useful for creating C shell scripts.

Using the argv Variable

A **csh** command script can be interpreted by saying:

```
csh script argument  . . .
```

where *script* is the name of the file containing a group of C shell commands and *argument* is a sequence of command arguments. The C shell places these arguments in the variable "argv" and then begins to read commands from *script*. These parameters are then available through the same mechanisms used to reference any other C shell variables.

If you make the file *script* executable by doing:

```
chmod 755 script
```

or:

```
chmod +x script
```

and then place a C shell comment at the beginning of the C shell script (that is, begin the file with a number sign (#)) then */bin/csh* will automatically be invoked to execute *script* when you type:

```
script
```

If the file does not begin with a number sign (#) then the standard shell */bin/sh* is used to execute it.

Substituting Shell Variables

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed, a mechanism known as variable substitution is performed on these words. Keyed by the dollar sign ($), this substitution replaces the names of variables by their values. Thus:

```
echo $argv
```

when placed in a command script causes the current value of the variable "argv" to be echoed to the output of the C shell script. It is an error for "argv" to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation:

```
$?name
```

expands to 1 if *name* is set or to 0 if *name* is not set. It is the fundamental mechanism for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation:

```
$#name
```

expands to the number of elements in the variable "name." To illustrate, examine the following terminal session (input is in boldface):

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
argv: Undefined variable.
%
```

It is also possible to access the components of a variable that has several values. Thus:

```
$argv[1]
```

gives the first component of "argv" or in the example above "a." Similarly:

```
$argv[$#argv]
```

would give "c", and

```
$argv[1-2]
```

would give:

```
a b
```

Other notations useful in C shell scripts are:

```
$n
```

where *n* is an integer. This is shorthand for:

```
$argv[n]
```

the *n*'th parameter and:

```
$*
```

which is a shorthand for:

```
$argv
```

The form:

```
$$
```

expands to the process number of the current C shell. Since this process number is unique in the system, it is often used in the generation of unique temporary filenames.

One minor difference exists between "$**n**"and "$argv[**n**]". The form "$argv[**n**]" yields an error if *n* is not in the range 1-$#argv while "$n" never yields an out-of-range subscript error. This is for compatibility with the way older shells handle parameters.

It is never an error to give a subrange of the form "n-"; if there are less than "n" components of the given variable then no words are substituted. A range of the form "m-n" likewise returns an empty vector without giving an error when "m" exceeds the number of elements of the given variable, provided the subscript "n" is in range.

Using Expressions

To construct useful C shell scripts, the C shell must be able to evaluate expressions based on the values of variables. In fact, all the arithmetic operations of the C language are available in the C shell with the same precedence that they have in C. In particular, the operations == and != compare strings and the operators && and | | implement the logical AND and OR operations. The special operators =~ and !~ are similar to == and != except that the string on the right side can have pattern matching characters (like *, ? or [and]). These operators test whether the string on the left matches the pattern on the right.

The C shell also allows file enquiries of the form:

```
-? filename
```

where question mark (?) is replaced by a number of single characters. For example, the expression primitive:

```
-e filename
```

tells whether *filename* exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or if it has nonzero length.

It is possible to test whether a command terminates normally, by using a primitive of the form:

```
{command}
```

which returns 1 if the command exits normally with exit status 0, or 0 if the command terminates abnormally or with exit status nonzero. If more detailed information about the execution status of a command is required, it can be executed and the "status" variable examined in the next command. Since "$status" is set by every command, its value is always changing.

For the full list of expression components, see **csh (CP)** in the IBM Personal Computer *XENIX Software Command Reference.*

Using the C Shell: A Sample Script

A sample C shell script follows that uses the expression mechanism of the C shell and some of its control structures:

```
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i !~ *.c) continue  # not a .c file so do not

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup . . .  not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script uses the **foreach** command. The command executes the other commands between the **foreach** and the matching **end.** For each of the values given between parentheses with the named variable "i" set to successive values in the list. Within this loop use the command **break** to stop executing the loop and **continue** to prematurely terminate one iteration and begin the next. After the **foreach** loop the iteration variable (**i** in this case) has the value at the last iteration.

The " noglob" variable is set to prevent filename expansion of the members of " argv". This is a good idea if the arguments to a C shell script are filenames already expanded or if the arguments contain filename expansion metacharacters. It is also possible to quote each use of a "$" variable expansion, but this is harder and less reliable.

The other control construct is a statement of the form:

```
if ( expression ) then
        command
          . . .
endif
```

The placement of the keywords in this statement is not flexible due to the current implementation of the C shell. The following two formats are not acceptable to the C shell:

```
if (expression) # Won't work!
then
        command
          . . .
endif
```

and

```
if (expression) then command endif # Won't work
```

The C shell does have another form of the if statement:

```
if ( expression ) command
```

that can be written:

```
if ( expression ) \
        command
```

Here we have escaped the newline for the sake of appearance. The command must not involve |, & or ; and must not be another control command. The second form requires the final backslash (\) to immediately precede the end-of-line.

The more general **if** statements above also admit a sequence of **else-if** pairs followed by a single **else** and an **endif**, for example:

```
if ( expression ) then
        commands
else if (expression ) then
        commands
 . . .

else
        commands
endif
```

Another important mechanism used in C shell scripts is the colon (:) modifier. We can use the modifier **:r** here to extract the root of a filename or **:e** to extract the extension. Thus if the variable "i" has the value */mnt/foo.bar* then:

```
echo $i $i:r $i:e
```

produces:

```
/mnt/foo.bar /mnt/foo bar
```

This example shows how the **:r** modifier strips off the trailing .bar and the **:e** modifier leaves only the bar. Other modifiers take off the last component of a pathname leaving the head **:h** or all but the last component of a pathname leaving the tail **:t**. These modifiers are fully described in the **csh (CP)** entry in the IBM Personal Computer *XENIX Software Command Reference*. It is also possible to use the command substitution mechanism to perform modifications on strings to then reenter the C shell environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the colon (:) modification mechanism. It is also important to note that the current implementation of the C shell limits the number of colon modifiers on a "$" substitution to 1. Thus:

```
% echo $i $i:h:t
```

produces:

```
/a/b/c /a/b:t
```

and does not do what you might expect.

The number sign character (#) lexically introduces a C shell comment in C shell scripts (but not from the terminal). All subsequent characters on the input line after a number sign are discarded by the C shell. This character can be quoted using (′) or (\) to place it in an argument word.

Using Other Control Structures

The C shell also has control structures **while** and **switch** similar to those of C. These take the forms:

```
while ( expression )
        commands
end
```

and:

```
switch ( word )

case str1:
        commands
        breaksw

 . . .

case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

For details see the XENIX *Software Command Reference* **csh (CP)**. C-programmers should use **breaksw** to exit from a **switch** while **break** exits a **while** or **foreach** loop. A common mistake to make in C shell scripts is to use **break** rather than **breaksw** in switches.

Finally, the C shell allows a **goto** statement, with labels looking like they do in C:

```
loop:
        commands
        goto loop
```

Supplying Input to Commands

Commands that are run from C shell scripts receive by default the standard input of the C shell running the script. It allows C shell scripts to fully participate in pipelines, but mandates extra notation for commands that are to take inline data.

Thus we need a metanotation for supplying inline data to commands in C shell scripts. For example, consider this script that runs the editor to delete leading blanks from the lines in each argument file:

```
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
```

The notation

```
<< 'EOF'
```

means that the standard input for the **ed** command comes from the text in the C shell script file up to the next line consisting of exactly EOF. The fact that the EOF is enclosed in single quotation marks ('), that is, it is quoted, causes the C shell to not perform variable substitution on the intervening lines. If any part of the word following the "<<" used by the C shell to terminate the text to be given to the command is quoted, then these substitutions are not performed. In this case since we used the form "1,$" in our editor script we needed to insure that this dollar sign was not variable substituted. Another way to ensure this is preceding the dollar sign ($) with a backslash (\), that is:

```
1,\$s/^[ ]*//
```

Quoting the EOF terminator is a more reliable way of achieving the same thing.

Catching Interrupts

If the C shell script creates temporary files, you may wish to catch interruptions of the C shell script so that you can clean up these files. You can then do:

```
onintr label
```

where *label* is a label in our program. If an interrupt is received the C shell does "goto label". Remove the temporary files, and do an **exit** command (built in to the C shell) to exit from the C shell script. To exit with nonzero status, write:

```
exit (1)
```

to exit with status 1.

Using Other Features

Other features of the C shell are useful to writers of C shell procedures. The **verbose** and **echo** options and the related **-v** and **-x** command line options help to trace the actions of the C shell. The **-n** option causes the C shell to read commands only.

Unless they begin with the number sign (#), the C shell does not execute C shell scripts. That is, C shell scripts that do not begin with a comment.

There is also another quotation mechanism using the double quotation mark ("), that allows only some of the expansion mechanisms discussed so far to occur on the quoted string and serves to make this string into a single word as the single quote (') does.

Starting a Loop at a Terminal

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For instance, if three shells are in use on a particular system, */bin/sh*, */bin/nsh*, and */bin/csh*, you can count the number of persons using each shell with the following commands:

```
grep -c csh$ /etc/passwd
grep -c nsh$ /etc/passwd
grep -c -v sh$ /etc/passwd
```

Since these commands are very similar, use **foreach** to simplify them:

```
 $  foreach i ('sh$' 'csh$' '-v sh$')
 ?  grep -c $i /etc/passwd
 ?  end
```

The C shell prompts for input with "?" when reading the body of the loop. This occurs only when the **foreach** command is entered interactively.

Also useful with loops are variables that contain lists of filenames or other words. For example, examine the following terminal session:

```
%  set a=(`ls`)
%  echo $a
csh.n csh.rm
%  ls
csh.n
csh.rm
%  echo $#a
2
```

The **set** command here gave the variable “a” a list of all the filenames in the current directory as value. You can then iterate over these names to perform any chosen function.

The C shell converts the output of a command enclosed in accent symbols (`) to a list of words. You can also place the quoted string within double quotation marks (") to take each (nonempty) line as a component of the variable. This prevents the lines from being split into words at blanks and tabs. Use a modifier **:x** later to expand each component of the variable into another variable by splitting the original variable into separate words at embedded blanks and tabs.

Using Braces with Arguments

Another form of filename expansion involves the characters, { and }. These characters specify that the contained strings, separated by commas (,) are to be consecutively substituted into the containing characters and the results expanded left to right. Thus:

```
A{str1,str2, . . . strn}B
```

expands to:

```
Astr1B Astr2B  . . .  AstrnB
```

This expansion occurs before the other filename expansions, and can be applied recursively (that is, nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. Use mechanism to generate arguments that are not filenames, but have common parts.

A typical use would be:

```
mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories *hdrs*, *retrofit* and *csh* in the home directory. This mechanism is most useful when the common prefix is longer than in this example:

```
chown root /usr/demo/{file1,file2, . . . }
```

Substituting Commands

A command enclosed in accent symbols (`) is replaced, just before filenames are expanded, by the output from that command. Thus, it is possible to do:

```
set pwd=`pwd`
```

to save the current directory in the variable "pwd" or to do

```
vi `grep -l TRACE  *.c`
```

to run the editor **vi** supplying as arguments those files whose names end in *.c* that have the string TRACE in them. Command expansion also occurs in input redirected with "<<" and within quotation marks ("). Refer to **csh (CP)** in the IBM Personal Computer *XENIX Software Command Reference* for more information.

Special Characters

The following table lists the special characters of **csh** and the XENIX system. A number of these characters also have special meaning in expressions. See the **csh** in the IBM Personal Computer *XENIX Software Command Reference* for a complete list.

Syntactic Metacharacter

;	Separates commands to be executed sequentially

|	Separates commands in a pipeline
( )	Brackets expressions and variable values
&	Follows commands to be executed without waiting for completion

Filename Metacharacters

/	Separates components of a file's pathname
.	Separates root parts of a filename from extensions
?	Expansion character matching any single character
*	Expansion character matching any sequence of characters
[ ]	Expansion sequence matching any single character from a set of characters
~	Used at the beginning of a filename to indicate home directories
{ }	Used to specify groups of arguments with common parts

Quotation Metacharacters

\	Prevents meta-meaning of following single character
'	Prevents meta-meaning of a group of characters
"	Like ', but allows variable and command expansion

Input/Output Metacharacters

<	Indicates redirected input
>	Indicates redirected output

Expansion/Substitution Metacharacters

$	Indicates variable substitution
!	Indicates history substitution
:	Precedes substitution modifiers
^	Used in special forms of history substitution
`	Indicates command substitution

Other Metacharacters

#	Begins scratch filenames; indicates C shell comments
-	Prefixes option (flag) arguments to commands

Appendixes

Contents

Appendix A. C Language Portability

Introduction

The standard definition of the C programming language leaves many details to be decided by individual implementations of the language. These unspecified features of the language detract from its portability and must be studied when attempting to write portable C code.

Most of the issues affecting C portability arise from differences in either target machine hardware or compilers. C was designed to compile to efficient code for the target machine (initially a PDP[1]-11) and so many of the language features not precisely defined are those that reflect a particular machine's hardware characteristics.

This appendix highlights the various aspects of C that may not be portable across different machines and compilers. It also briefly discusses the portability of a C program in terms of its environment, which is determined by the system calls and library routines it uses during execution, file pathnames it requires, and other items not guaranteed to be constant across different systems.

The C language has been implemented on many different computers with widely different hardware characteristics, from small 8-bit microprocessors to large mainframes. This appendix is concerned with the portability of C code in the XENIX programming environment. This is a more restricted problem to consider since all UNIX[2] systems to date run on hardware with the following basic characteristics:

1 PDP is a trademark of the Digital Equipment Corporation.
2 UNIX is a trademark of AT&T Bell Laboratories.

- ASCII character set
- 8-bit bytes
- 2-byte or 4-byte integers
- Twos complement arithmetic

These features are not formally defined for the language and may not be found in of all implementations of C. However, the remainder of this appendix is devoted to those systems where these basic assumptions hold.

The C language definition contains no specification of how input and output is performed. This is left to system calls and library routines on individual systems. Within XENIX systems there are system calls and library routines that can be considered portable. These are described briefly in a later section.

This appendix is not intended as a C-language primer. It is assumed that the reader is familiar with C, and with the basic architecture of common microprocessors.

Program Portability

A program is portable if it can be compiled and run successfully on different machines without alteration. There are many ways to write portable programs. The first is to avoid using inherently nonportable language features. The second is to isolate any nonportable interactions with the environment, such as I/O to nonstandard devices. For example programs should avoid hard-coding pathnames unless a pathname is common to all systems (for example, */etc/passwd*).

Files required at compiletime (that is, include files) can also introduce nonportability if the pathnames are not the same on all machines. In some cases include files containing machine parameters can be used to make the source code itself portable.

Machine Hardware

Differences in the hardware of the various target machines and differences in the corresponding C compilers cause the greatest number of portability problems. This section lists problems commonly encountered on UNIX systems.

Byte Length

By definition, the **char** data type in C must be large enough to hold as positive integers all members of a machine's character set. For the machines described in this appendix, the **char** size is exactly an 8-bit byte.

Word Length

In C, the size of the basic data types for a given implementation are not formally defined. Thus they often follow the most natural size for the underlying machine. It is safe to assume that **short** is no longer than **long**. Beyond that no assumptions are portable. For example on some machines **short** is the same length as **int**, whereas on others **long** is the same length as **int**.

Programs that need to know the size of a particular data type should avoid hard-coded constants where possible. Such information can usually be written in a fairly portable way. For example the maximum positive integer (on a twos complement machine) can be obtained with:

```
#define MAXPOS  ((int)(((unsigned)-1) >> 1))
```

This is preferable to something like:

```
#ifdef PDP11
#define MAXPOS 32767
#else
          . . .
#endif
```

To find the number of bytes in an **int** use "sizeof (int)" rather than 2, 4, or some other nonportable constant.

Storage Alignment

The C-language defines no particular layout for storage of data items relative to each other, or for storage of elements of structures or unions within the structure or union.

Some CPUs, such as the PDP-11 and M68000[3], require that data types longer than 1 byte be aligned on even byte address boundaries. Others, such as the 8086[4] and VAX[5] have no such hardware restriction. PDP and VAX are trademarks of the Digital Equipment Corporation. However, even with these machines, most compilers generate code that aligns words, structures, arrays, and long words on even addresses, or even long-word addresses. Thus, on the VAX-11, the following code sequence gives "8," even though the VAX hardware can access an **int** (a 4-byte word) on any physical starting address:

```
struct s_tag {
        char c;
        int  i;
};
printf("%d\n",sizeof(struct s_tag));
```

The principal implications of this variation in data storage are that data accessed as nonprimitive data types is not portable, and code that makes use of knowledge of the layout on a particular machine is not portable.

3 M6800 is a trademark of the Motorola Corporation
4 8086 is a trademark of the Intel Corporation
5 VAX is a trademark of the Digital Equipment Corporation

Unions containing structures are nonportable if the union accesses the same data in different ways. Unions are only likely to be portable only if they have different data in the same space at different times. For example, if the following union were used to obtain 4 bytes from a long word, the code would not be portable:

```
union {
        char c[4];
        long lw;
} u;
```

Always use the *sizeof* operator when reading and writing structures:

```
struct s_tag st;

    . . .

write(fd, &st, sizeof(st));
```

This ensures portability of the source code. It does not produce a portable data file. Portability of data is discussed in a later in this appendix.

Note that the *sizeof* operator returns the number of bytes an object would occupy in an array. Thus on machines with structures aligned to begin on a word boundary in memory, the *sizeof* operator includes any necessary padding for this in the return value, even if the padding occurs after all useful data in the structure. This occurs whether or not the argument is actually an array element.

Byte Order in a Word

The variation in byte order in a word affects the portability of data more than the portability of source code. However, any program that makes use of knowledge of the internal byte order in a word is not portable. For example, on some systems there is an include file *misc.h* that contains the following structure declaration:

```
/*
* structure to access an
* integer in bytes
*/
struct {
        char    lobyte;
        char    hibyte;
};
```

Certain less restrictive compilers could access the high- and low-order bytes of an integer separately, and in a completely nonportable way. The correct way is to use mask and shift operations to extract the required byte:

```
#define LOBYTE(i) (i & 0xff)
#define HIBYTE(i) ((i >> 8) & 0xff)
```

Even this operation is applicable only to machines with 2 bytes in an **int**.

One result of the byte ordering problem is that the following code sequence does not always perform as intended:

```
int c = 0;

read(fd, &c, 1);
```

On machines where the low-order byte is stored first, the value of "c" is the byte value read. On other machines the byte is read into some byte other than the low-order one, and the value of "c" is different.

Bitfields

Bitfields are not implemented in all C compilers. When they are, no field is larger than an **int**, and no field can overlap an **int** boundary. If necessary the compiler leaves gaps and moves to the next **int** boundary.

The C language makes no guarantees about whether fields are assigned left to right, or right to left in an **int**. Thus, while bitfields are useful for storing flags and other small data items, their use in unions to dissect bits from other data is definitely nonportable.

To ensure portability no individual field should exceed 16 bits.

Pointers

The C language is fairly generous in allowing manipulation of pointers, to the extent that most compilers permit nonportable pointer operations. The **lint** program is particularly useful for detecting questionable pointer assignments and comparisons.

The common nonportable use of pointers is the use of casts to assign one pointer to another pointer of a different data type. This almost always makes some assumption about the internal byte ordering and layout of the data type, and is therefore nonportable. In the following code, the byte order in the given array is not portable:

```
char c[4];
long *lp;

lp = (long *)&c[0];
*lp = 0x12345678L;
```

The **lint** program issues warning messages about such uses of pointers. Code like this is very rarely necessary or valid. It is acceptable, however, when using the *malloc* function to allocate space for variables that do not have **char** type. The routine is declared as type **char** * and the return value is cast to the type to be stored in the allocated memory. If this type is not **char** * then **lint** issues a warning concerning illegal type conversion. In addition, the *malloc* function is written to always return a starting address suitable for storing all types of data. **lint** does not know this, so it gives a warning about possible data alignment problems too. In the following example, *malloc* obtains memory for an array of 50 integers.

```
extern char *malloc();
int *ip;

ip = (int *)malloc(50 * sizeof(int));
```

This example attracts a warning message from *lint* .

The *C Reference Manual* states that a pointer can be assigned (or cast) to an integer large enough to hold it. The size of the **int** type depends on the given machine and implementation. This type is a **long** on some machines and **short** on others. Do not assume that "sizeof(char *) == sizeof(int)."

In most implementations, the null pointer value, "NULL" is defined as the integer value 0. This can lead to problems for functions that expect pointer arguments larger than integers. For portable code, always use:

```
func( (char *)NULL );
```

to pass a "NULL" value of the correct size.

Address Space

The address space available to a program running a UNIX operating system varies considerably from system to system. On a small PDP-11 there are only 64K bytes available for program and data combined. Larger PDP-11's, and some 16-bit microprocessors allow 64K bytes of data, and 64K bytes of program text. Other machines may allow considerably more text, and possibly more data as well.

Large programs, or programs that require large data areas can have portability problems on small machines.

Character Set

The C language does not require the use of the ASCII character set. In fact, the only character set requirements are all characters must fit in the **char** data type, and all characters must have positive values.

In the ASCII character set, all characters have values between zero and 127. Therefore they can all be represented in 7 bits, and on an 8-bits-per-byte machine are all positive, whether **char** is treated as signed or unsigned.

Use the set of macros defined under XENIX macros in the header file */usr/include/ctype.h* for most tests on character quantities. They provide insulation from the internal structure of the character set and in most cases their names are more meaningful than the equivalent line of code. Compare:

```
if(isupper(c))
```

to

```
if((c >= 'A') && (c <= 'Z'))
```

With some of the other macros, such as *isdigit* to test for a hex digit, the advantage is even greater. Also, the internal implementation of the macros makes them more efficient than an explicit test with an if statement.

Compiler Differences

A number of C compilers run under the UNIX system. On PDP-11 systems there is the "Ritchie" compiler. Also on the 11, and on most other systems, there is the Portable C Compiler.

Signed/Unsigned char, Sign Extension

The current state of the signed versus unsigned **char** problem is best described as unsatisfactory.

The sign extension problem is a serious barrier to writing portable C, and the best solution at present is to write defensive code that does not rely on particular implementation features.

Shift Operations

The left shift operator, "<<" shifts its operand a number of bits left, filling vacated bits with zero. This is a so-called logical shift. The right shift operator, ">>" when applied to an unsigned quantity, performs a logical shift operation. When applied to a signed quantity, the vacated bits are filled with zero (logical shift) or with sign bits (arithmetic shift). The decision is implementation dependent, and code that uses knowledge of a particular implementation is nonportable.

The PDP-11 compilers use arithmetic right shift. To avoid sign extension it is necessary to shift and mask out the appropriate number of high order bits:

```
char c;

c = (c >> 3) & 0x1f;
```

You can also avoid sign extension by using using the divide operator:

```
char c;

c = c / 8;
```

Identifier Length

Using long symbols and identifier names causes portability problems with some compilers. To avoid these problems, a program should keep the following symbols as short as possible:

- C Preprocessor Symbols
- C Local Symbols
- C External Symbols

The loader used can also place a restriction on the number of unique characters in C external symbols.

Symbols unique in the first six characters are unique to most C language processors.

On some non-UNIX C implementations, uppercase and lowercase letters are not distinct in identifiers.

Register Variables

The number and type of register variables in a function depends on the machine hardware and the compiler. Excess and invalid register declarations are treated as nonregister declarations and should not cause a portability problem. On a PDP-11, up to three register declarations are significant, and they must be of type **int**, **char**, or pointer. While other machines and compilers can support declarations such as:

```
register unsigned short
```

this should not be relied upon.

Since the compiler ignores excess variables of register type, declare the most important register type variables first. This way, if any are ignored, they will be the least important ones.

Type Conversion

The C language has some rules for implicit type conversion; it also allows explicit type conversions by type casting. The most common portability problem in implicit type conversion is unexpected sign extension. This is a potential problem whenever something of type **char** is compared with an **int**.

For example:

```
char c;

if(c == 0x80)
          . . .
```

never evaluates true on a machine that sign extends since “c” is sign extended before the comparison with 0x80, an **int**.

The only safe comparison between **char** type and an **int** is the following:

```
char c;

if(c == 'x')
          . . .
```

This is reliable because C guarantees all characters to be positive. The use of hard-coded octal constants is subject to sign extension. For example the following program prints “ff80” on a PDP-11:

```
main()
{
        printf("%x\n",'\200');
}
```

Type conversion also takes place when arguments are passed to functions. Types **char** and **short** become **int**. Machines that sign extend **char** can give surprises. For example the following program gives -128 on some machines:

```
char c = 128;
printf("%d\n",c);
```

This is because "c" is converted to **int** before passing to the function. The function itself has no knowledge of the original type of the argument, and is expecting an **int**. The correct way to handle this is to code defensively and allow for the possibility of sign extension:

```
char c = 128;
printf("%d\n", c & 0xff);
```

Functions With Variable Number of Arguments

Functions with a variable number of arguments present a particular portability problem if the type of the arguments is variable too. In such cases the code is dependent upon the size of various data types.

An include file in the XENIX command, */usr/include/varargs.h*, contains macros for use in variable argument functions to access the arguments in a portable way:

```
typedef char *va__list
#define va__dcl int va__alist;
#define va__start(list) list = (char *) &va__alist
#define va__end(list)
#define va__arg(list,mode) ((mode *)(list += sizeof(mode)))[-1]
```

The va_end() macro is not currently required. Use of the other macros is demonstrated by an example of the *fprintf* library routine. This has a first argument of type **FILE ***, and a second argument of type **char ***. Subsequent arguments are of unknown type and number at compilation time. They are determined at run time by the contents of the control string, argument 2.

The first few lines of *fprintf* to declare the arguments and find the output file and control string address could be:

```
#include <varargs.h>
#include <stdio.h>

int fprintf(va_alist)
va_dcl
{
       va_list ap;      /* pointer to arg list   */
       char *format;
       FILE *fp;

       va_start(ap);    /* initialize arg pointer */
       fp = va_arg(ap, (FILE *));
       format = va_arg(ap, (char *));

                      . . .
}
```

Just one argument is declared to *fprintf* . This argument is declared by the va__dcl macro to be type **int**, although its actual type is unknown at compile time. The argument pointer "ap" is initialized by *va__start* to the address of the first argument. Successive arguments can be picked from the stack so long as their type is known using the *va__arg* macro. This has a type as its second argument, and this controls what data is removed from the stack, and how far the argument pointer "ap" is incremented. In *fprintf*, once the control string is found, the type of subsequent arguments is known and they can be accessed sequentially by repeated calls to va__arg (). For example, arguments of type **double**, **int** *, and **short**, could be retrieved as follows:

```
double dint;
int *ip;
short s;

dint = va_arg(ap, double);
ip = va_arg(ap, (int *));
s = va_arg(ap, short);
```

The use of these macros makes the code more portable, although it does assume a certain standard method of passing arguments on the stack. In particular no holes must be left by the compiler, and types smaller than **int** (for example, **char**, and **short** on long word machines) must be declared as **int**.

Side Effects, Evaluation Order

The C language makes few guarantees about the order of evaluation of operands in an expression, or arguments to a function call. Thus:

```
func(i++, i++);
```

is extremely nonportable, and even:

```
func(i++);
```

is unwise if *func* is ever likely to be replaced by a macro, since the macro can use "i" more than once. Certain XENIX macros are common in user programs; these are all guaranteed to use their argument once, and so can safely be called with a side-effect argument. The most common examples are *getc*, *putc*, *getchar*, and *putchar* .

Operands to the following operators are guaranteed to be evaluated left to right:

```
,        &&        ||        ?        :
```

The comma operator here is a separator for two C statements. A list of items separated by commas in a declaration list is not guaranteed to be processed left to right. Thus the declaration:

```
register int a, b, c, d;
```

on a PDP-11 where only three register variables can be declared could make any three of the four variables register type, depending on the compiler. The correct declaration is to decide the order of importance of the variables being register type, and then use separate declaration statements, since the order of processing of individual declaration statements is guaranteed to be sequential:

```
register int a;
register int b;
register int c;
register int d;
```

Program Environment Differences

Most programs make system calls and use library routines for various services. This section indicates some of those routines that are not always portable, and those that particularly aid portability.

We are concerned here primarily with portability under the UNIX operating system. Many of the UNIX system calls are specific to that particular operating system environment and are not present on all other operating system implementations of C. Examples of this are *getpwent* for accessing entries in the UNIX password file, and *getenv* that is specific to the UNIX concept of a process environment.

Any program containing hard-coded pathnames to files or directories, or user IDs, login names, terminal lines or other system dependent parameters is nonportable. These types of constant should be in header files, passed as command line arguments, obtained from the environment, or obtained by using the UNIX default parameter library routines *dfopen*, and *dfread* .

Within the UNIX system, most system calls and library routines are portable across different implementations and UNIX releases. However, a few routines have changed in their user interface. The UNIX library routines are usually portable among UNIX systems.

The members of the printf family, *printf*, *fprintf*, *sprintf*, *sscanf*, and *scanf* have changed in several ways during the evolution of the UNIX system, and some features are not completely portable. The return values of these routines cannot be relied upon to have the same meaning on all systems. Some of the format conversion characters have changed their meanings, in particular those relating to uppercase and lowercase in the output of hexadecimal numbers, and the specification of **long** integers on 16-bit word machines. The reference manual page for *printf* in the IBM Personal Computer *XENIX Software Command Reference* contains the correct specification for these routines.

Portability of Data

Data files are almost always nonportable across different machine CPU architectures. As mentioned above, structures, unions, and arrays have varying internal layout and padding requirements on different machines. In addition, byte ordering within words and actual word length may differ.

The only way to achieve data file portability is to write and read data files as one-dimensional character arrays. This avoids alignment and padding problems if the data is written and read as characters, and interpreted that way. Thus ASCII text files can usually be moved between different machine types without too many problems.

The lint C Program Checker

The **lint** C program checker attempts to detect features of a collection of C source files that are nonportable or even incorrect C. One particular advantage of **lint** over any compiler checking is that **lint** checks function declaration and usage across source files. Neither compiler nor loader do this.

The **lint** program generates warning messages about nonportable pointer arithmetic, assignments, and type conversions. Passage unscathed through **lint** is not a guarantee that a program is completely portable.

Byte Ordering Summary

The following conventions are used in tables below:

a0	The lowest physically addressed byte of the data item a0 + 1, and so on.

b0 The least significant byte of the data item, 'b1' being the next least significant, and so on.

Any program that actually makes use of the following information is guaranteed to be nonportable!

Byte Ordering for Short Types

CPU	**Byte Order**	
	aØ	a1
PDP-11	bØ	b1
VAX-11	bØ	b1
8Ø86	bØ	b1
286	bØ	b1
M68ØØØ	b1	bØ
Z8ØØØ	b1	bØ

Byte Ordering for Long Types

CPU	**Byte Order**			
	aØ	a1	a2	a3
PDP-11	b2	b3	bØ	b1
VAX-11	bØ	b1	b2	b3
8Ø86	b2	b3	bØ	b1
286	b2	b3	bØ	b1
M68ØØØ	b3	b2	b1	bØ
Z8ØØØ	b3	b2	b1	bØ

Appendix B. 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 C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor-replacement of text by other text.

Besides the straightforward replacement of one string of text by another, **m4** provides:

- Macros 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-ins and user-defined macros work in exactly the same way, except that some of the built-in macros have side effects on the state of the process.

Invoking m4

The invocation syntax for **m4** is:

```
m4 [files]
```

Each filename argument is processed in order. If there are no arguments, or if an argument is a dash (-), then the standard input is read. The processed text is written to the standard output, and can be redirected as in the following example:

```
m4 file1 file2 - >outputfile
```

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. Name* must be alphanumeric and must begin with a letter (the underscore (__) counts as a letter). *Stuff* is any text, including text that contains balanced parentheses; it 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.

Things can be defined in terms of other things. For example

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if N is redefined? Or, to say it another way, is M defined as N or as 100? In **m4**, the latter is true, M is 100, so even if N subsequently changes, M does not.

This behavior arises because **m4** expands macro names into their defining text as soon as it possibly can. Here, that means that when the string N is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said:

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now M is defined to be the string N so when you ask for M later, always get the value of N (because the M is replaced by N which, in turn, is replaced by 100).

Quoting

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by single quotation marks ‘ ’ is not expanded immediately, but has the quotation marks stripped off. If you say:

```
define(N, 100)
define(M, 'N')
```

the quotation marks around the N are stripped off as the argument is being collected, but they have served their purpose, and M is defined as the string ‘N’, not 100. The general rule is that **m4** always strips off one level of single quotation marks whenever it evaluates something. This is true even outside of macros. If you want the word “define” to appear in the output, you have to quote it in the input, as in:

```
'define' = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining N:

```
define(N, 100)
 ...
define(N, 200)
```

Perhaps regrettably, the N in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written:

```
define(100, 200)
```

This statement is ignored by **m4**, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine 'N', you must delay the evaluation by quoting:

```
define(N, 100)
  . . .
define('N', 200)
```

In **m4**, it is often wise to quote the first argument of a macro.

If the forward and backward quotation marks 'and' are not convenient for some reason, the quotation marks can be changed with the built-in **changequote**. For example:

```
changequote([, ])
```

makes the new quotation marks the left and right brackets. You can restore the original characters with just:

```
changequote
```

Two additional built-ins are related to **define**. The built-in **undefine** removes the definition of some macro or built-in:

```
undefine('N')
```

removes the definition of N. Built-ins can be removed with **undefine**, as in:

```
undefine('define')
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. For instance, pretend that either the word "xenix" or "unix" is defined according to a particular implementation of a program. To perform operations according to which system you have you might say:

```
ifdef('xenix', 'define(system,1)' )
ifdef('unix', 'define(system,2)' )
```

Don't forget the single quote marks in the above example.

Ifdef actually permits three arguments: if the name is undefined, the value of **ifdef** is then the third argument, as in:

```
ifdef('xenix', on XENIX, not on XENIX)
```

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 the *n th* argument when the macro is actually used. Thus, the macro *bump* , defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1:

```
bump(x)
```

is

```
x = x + 1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through $1 to $9. (The macro name itself is $0.) Arguments not supplied are replaced by null strings, so we can define a macro *cat* which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

The arguments $4 through $9 are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus:

```
define(a,   b   c)
```

defines "a" to be "b   c."

Arguments are separated by commas, but parentheses are counted properly, so a comma protected by parentheses does not terminate an argument. That is, in

```
define(a, (b,c))
```

there are only two arguments; the second is literally "(b,c)." And of course a bare comma or parenthesis can be inserted by quoting it.

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 where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is implementation dependent.

As a simple example, suppose we want M to be 2**N+1. Then:

```
define(N, 3)
define(M, 'eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

Manipulating Files

You can include a new file in the input at any time by the built-in function **include :**

```
include(filename)
```

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** (for "silent include") says nothing and continues if it can't 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 say:

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as n. Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is not the diverted stuff. Furthermore, the diverted material is not rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

Using System Commands

You can run any program in the local operating system with the **syscmd** built-in. For example,

```
syscmd(date)
```

runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function **mktemp:** a string of XXXXX in the argument is replaced by the id 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.

```
define(compare, 'ifelse($1, $2, yes, no)')
```

The quotation marks, prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input:

```
ifelse(a, b, c, d, e, f, g)
```

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise the result is *g*. If the final argument is omitted, the result is null, so:

```
ifelse(a, b, c)
```

is *c* if *a* matches *b* , and null otherwise.

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
```

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.

```
translit(s, f, t)
```

modifies *s* by replacing any character found in *f* by the corresponding character of *t*. That is:

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that don't have an entry in *t* are deleted; as a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*. So:

```
translit(s, aeiou)
```

deletes vowels from "s."

A built-in called **dnl** deletes all characters that follow it up to and including the next newline. It is useful mainly for throwing away empty lines that otherwise tend to clutter up **m4** output. For example, if you say:

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines disappear.

Another way to achieve this, is

```
divert(-1)
        define( . . . )
         . . .
divert
```

Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus, you can say:

```
errprint('fatal error')
```

Dumpdef is a debugging aid that dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget the single quotation marks.

Appendix C. The XENIX Device Driver Guide

Introduction

This guide explains how to write device drivers for the IBM Personal Computer XENIX System. It discusses the XENIX model of devices as files, task time and interrupt time processing, how to configure device drivers with the kernel, and other special considerations that should be made when writing a device driver. The latter sections of this guide provide commentary on sample device driver sources for a line printer, a terminal, and a fixed disk. These source code samples are intended as prototypes from which the experienced programmer can begin writing a driver for a particular device.

If a user of a XENIX system wants to use some peripheral, there must be a *device driver* for that peripheral. A device driver is a set of routines that communicates with a hardware device and provides a uniform interface to the XENIX kernel. This uniform interface allows the kernel to translate user requests such as "read ten bytes" into some sensible action. This document provides the details necessary to write drivers for devices that conform reasonably well to one of the device models XENIX supports.

To a user of a XENIX system, a device will appear for most purposes to act like a *file*. A file consists of an ordered sequence of bytes. Files that contain data are called *regular files* and files that represent devices are called *special files*. Associated with each file is at least one name; the names of special files are, by convention, placed in the directory */dev*.

Associated with each special file is a *device number*. This number uniquely identifies the device, and consists of two parts. The two parts are the *major number*, and the *minor number*. The major number tells the kernel which device driver will handle requests for this special file, and the minor number provides addition information to the driver such as the unit number of the device.

There are two kinds of special files in the XENIX system: *character* and *block*. Character devices conform immediately to the byte stream model of XENIX files, while block devices require intervention by the kernel to provide a byte stream interface. In general, any device that appears to be a randomly addressable set of fixed size records should be a block device; anything else a character device. For example, interfaces for terminals and line printers should be character devices; disk or tape drives should be block devices.

Preliminaries

Depending on the type of processor, the registers that control a device may live either in main memory (referred to as memory mapped) or in I/O space. To provide some portability between these two types of systems, four routines are provided by the kernel: *in ()* and *out (),* and *inb ()* and *outb ().* *In ()* and *inb ()* both take an address as an argument, and return the word or byte, respectively. *Out ()* and *outb ()* take an address as the first argument, and a value as the second argument. The word or byte at the specified address is set to the specified value.

When the CPU is executing instructions in user programs, it is said to be in *user* mode; when it is executing instructions in the XENIX kernel, it is said to be in *kernel* mode. When the CPU receives an interrupt from an external device, it reverts to kernel mode if it was in user mode, and control is passed to the interrupt routine of the appropriate device driver. When the driver is done, it returns, and the processing that was interrupted is resumed. The processing that was interrupted is referred to as *task time* processing, and the processing that took place as a result of the interrupt is called *interrupt time* processing.

If it is necessary to prevent acknowledgement of interrupts during task time processing, this can be done by calling the routine *spl5 ().* *Spl5 ()* takes no arguments, and returns a value that is used when restoring interrupts with the routine *splx ()*. *Splx ()* takes the return value of *spl5 ()* and enables interrupts. Calls to *spl5 ()* and *splx ()* nest correctly.

Sometimes a device driver may receive a request that it cannot service immediately. For example, it may receive a write request when the output buffer is full. In this case, the requesting process can suspend itself by calling *sleep ()*. *Sleep ()* takes two arguments; the first is a unique number, and the second is the priority. When the condition is alleviated, some other process may awaken the suspended process by calling *wakeup ()*. *Wakeup ()* takes as its only argument the unique number mentioned above; all the processes that have called *sleep ()* with that number are awakened. A sleeping process may also be awakened by a signal. When a process is awakened, the call to *sleep ()* returns, and the process should check that the reason for going to sleep has disappeared. The convention for generating unique numbers for *sleep ()* and *wakeup ()* is to use the address of some data structure the driver uses; since no data structures will have the same address, uniqueness is guaranteed. The priority passed to *sleep ()* has two effects:

1. It determines the priority of the process when it awakens.

2. If a process goes to sleep at a priority lower than the manifest constant PZERO, the sleep will not be broken by a signal. The priority is below PZERO if the condition is likely to disappear almost immediately, and above PZERO otherwise. *Sleep ()* should never be called at interrupt time.

The kernel provides a mechanism for scheduling a call to a routine at some later time. The driver calls the routine *timeout ()* with the following three arguments:

1. The routine to be called.

2. The argument to the routine.

3. The number of clock ticks that should elapse before the call.

Busy waiting can be effected by using *timeout ()* with *sleep ()* and *wakeup ()*. The following code fragment illustrates this:

```
#define PERIOD  HZ/10   /* 1/10 second */
#define BUSYPRI (PZERO - 1)     /* somewhat arbitrary */

int status;

int busywait()  /* wait until status is non-zero */
{
        while ( status == 0 ) {
                timeout(stopwait, 0, PERIOD);
                sleep(&status, BUSYPRI);
        }
}

int stopwait()
{
        wakeup(&status);
}
```

A driver should never loop waiting for a status change unless the delay involved is shorter than 100 microseconds.

Each driver has a prefix that is applied to all the names it uses. Typically this prefix is two characters, but it can be up to four characters long. For example, a fixed disk driver might use the prefix hd. In the following discussions, the names all use the prefix xx.

Character Devices

Character devices conform to the file model; their data consists of a stream of bytes delimited only by the beginning and end of file.

The kernel interfaces to character device drivers through six routines named *xxopen, xxclose, xxread, xxwrite, xxioctl,* and *xxintr.*

The *xxopen ()* routine is called each time the device is opened. The kernel calls this routine with two arguments:

1. The minor number of the device

2. The *oflag* argument that was passed to the open system call.

It is the responsibility of the *xxopen ()* routine to initialize the device, and perform any error or protection checking.

The *xxclose ()* routine is called on the last close on a device. It gets two arguments: the minor number of the device being closed, and the flags that were passed on the last open. The flags are ignored. The close routine is responsible for any cleanup that may be required, such as disabling interrupts, clearing device registers, ejecting media, etc.

The *xxread ()* and *xxwrite ()* routines are called when a program makes a read or write system call. Their responsibility is to transfer data to and from the user's address space.

Two routines are available to transfer one character at-a-time to or from the user. The *cpass ()* routine gets one character from the user; *passc ()* passes the character argument to the user. Both routines return -1 when there are no more characters to be transferred.

There is a pool of small buffers called *clists* in the kernel. Clists are small buffers that form linked lists with head and tail pointers; thus, they provide a FIFO buffer. The elements in the linked list are called *cblocks* ;. each cblock can hold a small number of characters. These are used for buffering low-speed character devices. The primary use of the clist buffers is for terminal devices that must interface with the common terminal interface.

A driver that wishes to use the clist buffer mechanism must declare a queue header of type *clist*. If both input and output are buffered, the driver will need two headers. There are six routines that the driver can use to manipulate clist buffers.

Getc () and *putc ()* move one character from or to the clist buffer for each call. *Getc ()* returns the next character in the buffer, or -1 if the buffer is empty. *Putc ()* places in the buffer the character passed as an argument and returns -1 if there is no free space.

Getcb () and *putcb ()* are similar to *getc ()* and *putc (),* but put and get one cblock instead of a character. *Getcb ()* returns a pointer to a cblock, which it takes from the clist passed as an argument. *putcb ()* takes two arguments; the first is a pointer to a cblock, and the second is a pointer to a clist. It puts the cblock onto the clist.

All the cblocks not currently used are kept on a freelist. The routine *getcf ()* takes a cblock from the freelist, and returns a pointer to it. The routine *putcf ()* puts the cblock pointed to by its argument onto the freelist. Since there are a limited number of cblocks in the system, each driver must strive to 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 size exceeds the high water mark; at that point, the requesting processes are suspended via *sleep ()*. When the buffer has drained below the 'low water mark' the suspended processes are awakened, and can fill the buffer again.

For input buffering, the driver usually buffers the data up to some limit. If the buffer length exceeds this limit, data is discarded to make room for the more recent data.

The *xxioctl ()* routine is called by the kernel when a user process makes an *ioctl ()* system call for that device. Its function is to perform hardware dependent functions such as setting data rate. It gets called with four arguments: the minor number, the command and argument passed to the *ioctl* system call, and the flags passed on the *open ()* system call for that device.

The *xxintr ()* routine is called by the kernel when the device issues an interrupt. Since the interrupt signals completion of a data transfer, the interrupt routine must determine the appropriate action; perhaps fetching the received character and placing it in the input buffer, or removing the next character from the output buffer and starting the transmission. The *xxintr ()* routine is called with the interrupt number as its only argument.

If a serial device is intended to be used as a 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 sets of routines that roughly correspond to the character device functions. 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; each entry in this table represents one line discipline, and has entries for eight functions.

The *l_open* routine should be called on the first open of a device. The *l_close* routine should be called on the last close of the device. The *l_read* and *l_write* routines are called by the drivers read and write routines, to pass characters to and from the calling process. The *l_input* routine is called to buffer an incoming character. The *l_output, l_ioctl,* and *l_mdmint* routines are currently unused.

If the device driver is to be used to handle the console, the *putchar ()* routine for printing error and panic messages must be supplied. This routine puts one character on the console, doing a busy wait rather than depending on interrupts. The character to be printed is passed as an argument.

Block Devices

Block devices appear to the kernel to be a randomly addressable set of records of size BSIZE, where BSIZE is a manifest constant defined in the file param.h. The XENIX kernel inserts a layer of buffering software between user requests for block devices and the device driver. This buffering improves system performance by acting as a cache, allowing read ahead and write behind on block devices.

Each buffer in the cache has associated with it a header of type *struct buf* which holds all the data describing the data in the buffer. When an I/O request is passed to the block device driver, all of the relevant information is held in the buffer header, and the request is linked into a chain of pending requests.

Each block device driver must have a header of type *struct iobuf* for this chain of requests, named *xxtab*.

The interface between the kernel and the device driver consists of the routines *xxopen (), xxclose (), xxstrategy (),* and *xxintr ().*

The *xxopen ()* routine is called each time the device is opened. The kernel calls this routine with two arguments:

1. The device number

2. The *oflag* argument that was passed to the open system call.

It is the responsibility of the *xxopen ()* routine to initialize the device, and perform any error or protection checking. The *xxclose ()* routine is called on the last close on a device. It gets two arguments: the device number of the device being closed, and the flags that were passed on the last open. The flags are ignored. The close routine is responsible for any cleanup that may be required, such as disabling interrupts, clearing device registers, and ejecting media.

The *xxstrategy ()* routine is called by the kernel to queue a I/O request. It is called with one argument: a pointer to a buffer header. The *xxstrategy ()* routine must make sure the request is for a valid block, and then insert the request into the queue. Usually the driver will call *disksort ()* to insert the request into the queue; *disksort ()* takes two arguments: a pointer to the head of the queue, and a pointer to the buffer header to be inserted.

The *xxintr ()* routine is called whenever the device issues an interrupt. It depending on the meaning of the interrupt, it may mark the current request as complete, start the next request, continue the current request, or retry a failed operation.

Often a block device driver will provide a character device driver interface so that the device can be accessed without going through the structuring and buffering imposed by the kernel's block device interface. For example, a program might wish to read magtape 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 entails only adding *xxread ()* and *xxwrite ()* routines.

Because it is common for block device drivers to provide the raw interface, all the work has been abstracted into one routine, *physio ()*, which validates the request, builds a buffer header, and calls the strategy routine to queue the request. The arguments to *physio ()* are:

- A pointer to the strategy routine
- A pointer to the buffer header to be filled,
- The device number of the device
- A read/write flag

The driver must be prepared for block sizes different from BSIZE, or requests longer than BSIZE.

Configuration and Installation

Once the device driver is written, you need to link a new kernel that includes the device driver. There are two directories within the directory */usr/sys* that need to be modified, they are: *conf* and *io*. The source for the driver is placed in *io* and compiled into object code. The object is then archived in the library *lib__io*. In the conf directory, the files *master* and *xenixconf* need to have a line added to specify the new device in the configuration.

Following are step-by-step instructions on the method of installing a new device driver. We recommend that you secure a bootable copy of the working kernel and carefully follow these instructions:

1. Login as the super-user.
2. Change directory to *usr/sys/io* and copy the source code of the device driver into that directory.
3. Compile the source and archive the object code generated by typing **make OBJS=***XX.o*, where *XX* is the filename of the source code. (for example: make 'OBJS = prtr.o')

4. Change directory to */dev*. List the files in the directory using **ls -l** and determine a unique major device number for the driver. Execute a **mknod** with the device name, device type, major device number, and minor device number (for example: mknod /dev/prtr c 9 0). (The device name is the same as the name used to access the driver. You may choose any name you wish.) This creates the device special file in the */dev* directory. Remember the major device number; you will need it in step 6.

5. Change directory to */usr/sys/conf*. List the files in the directory and remove the c.c file if it exists.

6. Edit the file */usr/sys/conf/master* and add the appropriate line to the device information table that describes the driver. Refer to master(F) in the IBM Personal Computer XENIX Command Reference for more information about the *master* file. As an example, the lines for the three example device drivers in this appendix might be:

```
* 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    0    6  1   0    5     0     0   0
```

Note that major numbers must be unique; that is, no two devices can have the same block major numbers, or the same character major numbers.

7. Edit the file */usr/sys/conf/xenixconf* and add the name of the device, followed by the number of devices present in the configuration. If more information is needed about this definition file, refer to config(CP) in the IBM Personal Computer XENIX Software Command Reference. For the driver examples in this appendix, you might use:

```
hd      1
td      2
lp      1
```

8. Execute a **make** command with no arguments in your current directory, */usr/sys/conf* (for example: make). The new kernel is linked and placed in a file named *xenix* in that directory.

   If any errors occur while the kernel is being linked, do not go on. Review your device driver for invalid code, such as using functions which do not exist in the kernel and start again.

9. If the kernel is successfully linked, change directory to the root (/). Move the current kernel *xenix* to *xenix-* and copy the new kernel from */usr/sys/conf* to the root directory.

10. Execute a **haltsys** and restart the system as usual. If the system starts normally, the driver is installed and ready for testing. If the system does not start normally, load the system using the backup kernel by entering **hd xenix-** when prompted.

Warnings

What follows are warnings, based on real world experience, about possible problems that can occur when writing a device driver. It is very important to heed the advice presented here when writing a device driver:

- Don't defer interrupts with *spl?()* calls any longer than necessary.

- Don't touch the per process data in the u structure at interrupt time.

- Don't call *seterror ()* or *sleep ()* at interrupt time.

- Don't call *spl* at interrupt time.

- Make interrupt time processing as short as possible.

- Protect buffer and clist processing with *spl ()* calls.

- Avoid busy waiting whenever possible.

Sample Line Printer Driver

This and following sections consist of sample device driver listings. for line printer, terminal, and fixed disk drivers. Each 50-line segment of these listings is preceded by a general commentary on the code itself. This commentary describes each of the routines used in a particular driver and explains the purpose of certain key lines in the program. These key lines are denoted by their linenumber in the listing.

The driver presented here is for a single parallel interface to a printer. It transfers characters one at a time, buffering the output from the user process through the use of cblocks.

11. LPPRI is the priority to sleep at when a process needs to stop. Since the priority is greater than PZERO, a signal sent to the suspended process will awaken the suspended process.

12. If there are fewer than LOWAT characters in the buffer, a process which was suspended because of the buffer being full can be restarted.

13. HIWAT is the maximum number of characters in the queue. If a process fills the buffer up to this point, it will be suspended via *sleep ()* until the buffer has drained below LOWAT.

17. The registers in this interface occupy a contiguous block of address, starting at RBASE, and running through RBASE+2. The data to be printed is placed in RDATA a character at a time. The status of the printer can be read from RSTATUS, and the interface can be configured by writing into RCONTRL.

27. The flags defined here are kept in the variable *lp__flags*. FIRST is set if the interface has been initialized. ASLEEP is set if a process is asleep waiting for the buffer to drain below LOWAT.

31. *lp__queue* is the head of the linked list of cblocks that forms the output buffer.

32. *lp__flags* is the variable in which the flags mentioned above are kept.

lpopen()

The *lpopen ()* routine is called when some 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. Since this driver supports only one device, the minor number is ignored.

37: If this is the first time (since boot) that the device has been touched, we initialize the interface by setting the CRESET bit in the control register.

41: Enable interrupts from this device by setting the IENABL bit in the control register.

lpclose()

The *lpclose ()* routine is called on the last close of the device; that is, when the current *close ()* system call will reduce the number of processes referencing the device to zero. No action is taken.

```
/*
** lp- prototype line printer driver
*/
#include ../h/param.h
#include ../h/dir.h
#include ../h/a.out.h
#include ../h/user.h
#include ../h/file.h
#include ../h/tty.h

#define LPPRI   PZERO+5
#define LOWAT   50
#define HIWAT   150

/* register definitions */

#define RBASE   0x00            /* base address of registers */
#define RDATA   (RBASE + 0)        /* place character here */
#define RSTATUS (RBASE + 1)        /* non zero means busy */
#define RCONTRL (RBASE + 2)        /* write control here */

/* control definitions */
#define CINIT   0x01    /* initialize the interface */
#define CIENABL 0x02    /* +Interrupt enable */

/* flags definitions */
#define FIRST   0x01
#define ASLEEP  0x02
#define ACTIVE  0x04

struct clist lp_queue;
int lp_flags = 0;

lpopen(dev)
int dev;
{
        if ( (lp_flags & FIRST) == 0 ) {
                lp_flags |= FIRST;
                outb(RCtrl, CRESET);
        }
        outb(RCtrl, CIENABL);
}



lpclose(dev)
int dev;
{
}

lpwrite(dev)
int dev;
```

lpwrite()

The *lpwrite ()* routine is called to move the data from the user process to the output buffer.

55. While there are still characters to be transferred . . .

56. Raise the processor priority so that the interrupt routine can't change the buffer. If the buffer is full, we make sure the printer is running, make note of the fact that we are waiting, and go to sleep. When we wakeup, we check to make sure we the buffer is drained enough, and if it has, we go back to the old priority and put the character in the buffer.

65. Make sure the printer is running by locking out interrupts and calling *lpstart ().*

lpstart()

The *lpstart ()* routine ensures that the printer is running. It's called twice from *lpwrite (),* and serves simply to avoid duplicate code.

72. If the printer is running, just return; otherwise, mark it ACTIVE, and call *lpintr ()* to start the transfer of characters.

lpintr()

The *lpintr ()* routine is called from two places: *lpstart ()*, and from the kernel when an interrupt occurs.

84. If *lpintr ()* gets called when we don't expect it to, or we don't have anything to do, we just return without doing anything.

88. While the printer indicates that it can take more characters and we have characters to give it, we get the character from the buffer through *getc (),* and pass it to the interface by writing it to the data register.

92. If the buffer has fewer than LOWAT characters in it, and some process is asleep waiting for room, wake him up.

98. If the queue is now empty turn off the ACTIVE flag. Note that the completion interrupt for the transfer that empties the buffer is in some sense spurious, since it will occur with the ACTIVE flag reset.

```
{
        register int c;
        int s;

        while ( (c = cpass()) >= 0 ) {
                ospl = SPL();
                while ( lp_queue.c_cc > HIWAT ) {
                        lpstart(dev);
                        lpflags = ASLEEP;
                        sleep(&lp_queue, LPPRI);
                }
                splx(ospl);
                putc(c, &lp_queue);
        }
        s = SPL();
        lpstart();
        splx(s);
}

lpstart()
{
        if ( lp_flags & ACTIVE )
                return; /* interrupt chain is keeping printer going */
        lp_flags |= ACTIVE;
        lpintr(0);
}


lpintr(vec)
int vec;
{
        int tmp;

        if ( (lp_flags & ACTIVE) == 0 )
                return;           /* ignore spurious interrupt */

        /* pass chars until busy */
        while ( inb(RSTATUS) == 0 && (tmp = getc(&lp_queue)) >= 0)
                outb(RDATA, tmp);


        /* wakeup the writer if necessary */
        if ( lp_queue.c_cc < LOWAT && lp_flags & ASLEP ) {
                lp_flags &= °ASLEP;
                wakeup(&lp_queue);
        }

        /* wakeup writer if waiting for drain */
        if ( lp_queue.c_cc <= 0 )
                lp_flags &= °ACTIVE;
}
```

Sample Terminal Driver

This driver supports one serial terminal on a hypothetical UART type interface.

11: The interface for each line consists of seven registers. The values defined here represent offsets from the base address, which is defined elsewhere. The data to be transmitted is placed a character at a time into the RTDATA register. Likewise, the received data is read a 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 RCtrl register. Interrupts are enabled or disabled by the setting of the bits in the RIENABL register. The data rate is set by changing the contents of the RSPEED register.

29. The two low order bits determine the length of the character sent. The next two bits control the data-terminal-ready and request-to-send lines of the interface. The next three bits control the number of stop bits, whether parity is generated, and whether generated parity is even or odd. Finally, the most significant bit forces the transmitter to continuous spacing if it is set.

41. 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.

46. After an interrupt, the value in the interrupt identification register will contain one of three values, indicating the reason for the interrupt.

```
/*
** td- terminal device driver
*/
#include ../h/param.h
#include ../h/dir.h
#include ../h/user.h
#include ../h/file.h
#include ../h/tty.h
#include ../h/conf.h

/* registers */
#define RRDATA          /* received data */
#define RTDATA          /* transmitted data */
#define RSTATUS         /* status */
#define RCtrl           /* control */
#define RIENABL         /* interrupt enable */
#define RSPEED          /* data rate */
#define RIIR            /* interrupt identification */

/* status register bits */
#define SRRDY   0x01    /* received data ready */
#define STRDY   0x02    /* transmitter ready */
#define SOERR   0x04    /* received data overrun */
#define SPERR   0x08    /* received data parity error */
#define SFERR   0x10    /* received data framing error */
#define SDSR    0x20    /* status of dsr (cd)*/
#define SCTS    0x40    /* status of clear to send */

/* control register */
#define CBITS5  0x00    /* five bit chars */
#define CBITS6  0x01    /* six bit chars */
#define CBITS7  0x02    /* seven bit chars */
#define CBITS8  0x03    /* eight bit chars */
#define CDTR    0x04    /* data terminal ready */
#define CRTS    0x08    /* request to send */
#define CSTOP2  0x10    /* two stop bits */
#define CPARITY 0x20    /* parity on */
#define CEVEN   0x40    /* even parity otherwise odd */
#define CBREAK  0x80    /* set xmitter to space */

/* interrupt enable */
#define EXMIT   0x01    /* tranmitter ready */
#define ERECV   0x02    /* receiver ready */
#define EMS     0x04    /* modem status change */

/* interrupt ident */
#define IRECV   0x01
#define IXMIT   0x02
#define IMS     0x04

```

51. The values to be loaded into the RSPEED register to get various data rates are defined here.

71. Each line must have a tty structure allocated for it.

72. Here, the base addresses of the registers is defined for each line.

tdopen()

The *tdopen ()* routine is called whenever a process makes an *open ()* system call on a special file corresponds to this driver.

83. If the minor number indicates a device that doesn't exist, indicate the error, and return.

89. If the line is already open for exclusive use, and the current user is not the super user, indicate the error and return.

93. 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 struct, and configure the line by calling *tdparam ()*.

98. Defer interrupts so that the interrupt routines cannot change the state while we are examining it.

99. If the line is not using modem control, or turning on data terminal ready and request to send resulted in carrier detect being asserted by the remote device, indicate that carrier is present on this line. Otherwise, indicate that there is no carrier.

```
/* data rates */
int td_speeds¢| = {
        /* B0    */     0,
        /* B50   */     2304,
        /* B75   */     1536,
        /* B110  */     1047,
        /* B134  */     857,
        /* B150  */     768,
        /* B200  */     0,
        /* B300  */     384,
        /* B600  */     192,
        /* B1200 */     96,
        /* B1800 */     64,
        /* B2400 */     48,
        /* B4800 */     24,
        /* B9600 */     12,
        /* EXTA  */     6, /* 19.2k bps */
        /* EXTB  */     58 /* 2000 bps */
        };

struct tty td_tty¢NTDDEVS|;
int td_addr¢NTDEVS| = { 0x00, 0x10 };


tdopen(dev, flag)
int dev, flag;
{
        register struct tty *tp;
        int addr;
        extern tdproc();
        int     x;

        if ( dev >= NTDDEVS ) {
                seterror(ENXIO);
                return;
        }
        tp = &td_tty¢dev|;
        addr = td_addr¢dev|;
        if( (tp->t_lflag & XCLUDE) && !suser() ) {
                seterror(EBUSY);
                return;
        }
        if ((tp->t_state&(ISOPEN|WOPEN)) == 0) {
                ttinit(tp);
                tp->t_proc = tdproc;
                tdparam(dev);
        }
        x = spl5();
        if ( tp->t_cflag & CLOCAL || tdmodem(dev, TURNON))
                tp->t_state |= CARR_ON;
```

103. If the open is supposed to wait for carrier, wait until carrier is present.

108. Call the *l__open* routine indirectly through the *linesw* table. This completes the machinations required for the current line discipline to open a line.

109. Resume taking interrupts.

tdclose()

The *tdclose ()* routine is called on the last close on a line.

117. Call the close routine through the *linesw* table to do the work required by the current line discipline.

118. If the "hang up on last close" bit is set, drop data terminal ready and request to send.

120. Reset the exclusive use bit.

122. To prevent spurious interrupts, disable all interrupts for this line.

tdread() and tdwrite()

Both of these routines simply call the relevant routine via the *linesw* table; the called routine performs the action appropriate for the current line discipline.

tdparam()

The *tdparam ()* routine configures the line to the mode specified in the appropriate tty structure.

142. Get the base address and flags for the referenced line.

146. The speed BO has the special meaning of "hang up the line."

```
		else
			tp->t_state &= °CARR_ON;
		if (!(flag&FNDELAY))
			while ((tp->t_state&CARR_ON) == 0) {
				tp->t_state |= WOPEN;
				sleep((caddr_t)&tp->t_canq, TTIPRI);
			}
		(*linesw¢tp->t_line|.l_open)(tp);
		splx(x);
}

tdclose(dev)
{
		register struct tty *tp;

		tp = &td_tty¢dev|;
		(*linesw¢tp->t_line|.l_close)(tp);
		if (tp->t_cflag & HUPCL)
			tdmodem(dev, TURNOFF);
		tp->t_lflag &= °XCLUDE;  /* turn off exclusive use bit */
		/* turn off interrupts */
		out(td_addr¢dev| + RIENABL, 0);
}

tdread(dev)
{
		(*linesw¢tp->t_line|.l_read)(&td_tty¢dev|);
}

tdwrite(dev)
{

		(*linesw¢tp->t_line|.l_write)(&td_tty¢dev|);
}

tdparam(dev)
{
		register int cflag;
		register int addr;
		register int temp, speed, x;

		addr = td_addr¢dev|;
		cflag = td_tty¢dev|.t_cflag;

		/* if speed is B0, turn line off */
		if ( (cflag & CBAUD) == B0){
			outb(addr + RCONTRL, inb(addr+RCONTRL) &  CDTR &  CRTS);
			return;
		}

```

152. The remainder of this routine simply loads the device registers with the correct values.

tmodem()

The *tmodem* routine controls the data terminal ready and request to send outputs of the line. Its return value indicates whether data set ready (carrier detect) is present for the line.

180. If *cmd* was TURNON, we turn on modem interrupts, and assert data terminal ready and request to send.

185. We disable modem interrupts, and drop data terminal ready and request to send.

189. Return a zero value if there is no data set ready on this line, otherwise return a non zero value.

tdintr()

The *tdintr ()* routine determines which line caused the interrupt, and what the reason was, and calls the appropriate routine to handle the interrupt.

198. Different lines will result in different interrupt vectors being passed as *tdintr*()'s argument.. Here, we deduce the minor number from the vec that was passed to us.

```
        /* set up speed */
        outb( addr + RSPEED, td_speeds¢ cflag & CBAUD |);

        /* set up line control */
        temp = (cflag & CSIZE) >> 4; /* length */
        if ( cflag & CSTOPB )
                temp |= CSTOP2;
        if ( cflag & PARENB ) {
                temp |= CPARITY;
                if ( (cflag & PARODD) == 0)
                        temp |= CEVEN;
        }
        temp |= CDTR | CRTS;
        out( addr + RCtrl, temp );

        /* setup interrupts */
        temp = EXMIT;
        if ( cflag & CREAD )
                temp |= ERECV;
        outb(addr + RENABL, inb(RENABL) | temp);
}

tdmodem(dev, cmd)
int dev, cmd;
{
        register int addr;

        addr = td_addr¢dev|;
        switch(cmd){
        case TURNON: /* enable modem interrupts, set DTR & RTS true */
                outb(addr + RENABL, inb(RENABL) | EMS);
                outb(addr + RCONTRL, inb(RENABL) | CDTR | CRTS );
                break;
        case TURNOFF: /* disable modem interrupts, reset DTR, RTS */
                outb(addr + RENABL, inb(RENABL) & °EMS);
                outb(addr + RCONTRL, inb(RENABL) * °(CDTR | CRTS) );
                break;
        }
        return (inb(addr + RSTATUS) & SDSR);
}
#endif

tdintr(vec)
int vec;
{
        register int iir, dev, inter;

        switch( vec ) {
                case VECT0:
                        dev = 0;
```

209. While the interrupt identification register indicates that there is more to deal with, call the appropriate routine. When the condition that caused the interrupt is dealt with, the UART will reset the bit in the register by itself.

tdxint()

The *tdxint ()* routine is called when a transmitter ready interrupt is received. It may issue a CSTOP character to indicate that the device on the other end must stop sending characters, or it may issue a CSTART character to indicate that the device on the other end may resume sending characters, or it may call *tdproc ()* to send the next character in the queue.

226. If the transmitter is ready, reset the busy indicator, and go do it.

229. If we are to restart the line, send a CSTART, and reset the indicator.

232. If we are to stop the line, send a CSTOP, and reset the character.

236. Most of the time, we will just call *tdproc ()* and ask it to send the next character in the queue.

tdrint()

The *tdrint ()* routine is called when a receiver interrupt is received. All it has to do is pass the character, along with any errors, to the appropriate routine via the *linesw* table.

250. Get the character and status.

```
                                break;
                        case VECT1:
                                dev = 1;
                                break;
                        default:
                                printf(tdint: wrong level interrupt (%x)\en,vec);
                                return;
                        }
                while( (iir = inb(td_addr¢dev|+RIIR)) != 0) {
                        if( (iir & IXMIT) != 0 )
                                tdxint(dev);
                        if( (iir & IRECV) != 0 )
                                tdrint(dev);
                        if( (iir & IMS) != 0 )
                                tdmint(dev);
                }
        }

        tdxint(dev)
        {
                register struct tty *tp;
                register int addr;

                tp = &td_tty¢dev|;
                addr = td_addr¢dev|;
                if ( inb(addr + RSTATUS) & STRDY )
                {
                        tp->t_state &= °BUSY;
                        if (tp->t_state & TTXON) {
                                outb(addr + RTDATA, CSTART);
                                tp->t_state &= °TTXON;
                        } else if (tp->t_state & TTXOFF) {
                                outb(addr + RTDATA, CSTOP);
                                tp->t_state &= °TTXOFF;
                        } else
                                tdproc(tp, T_OUTPUT);
                }
        }

        tdrint(dev)
        {
                register int c, status;
                register int addr;
                register struct tty *tp;

                tp = &td_tty¢dev|;
                addr = td_addr¢dev|;

                /* get char and status */
                c = inb( addr + RRDATA );
```

256. If we detected any errors, set the appropriate bit in *c*.

262. And finally, pass the character and errors to the *l_input ()* routine for the current line discipline.

tdmint()

The *tdmint ()* routine is called whenever a modem interrupt is caught.

271. If we aren't doing modem support for this line, just return.

276. If we see data set ready for this line, and we didn't before, mark the line as having carrier, and wakeup any processes that are waiting for the carrier before their *tdopen ()* call can complete.

281. If we don't see data set ready for this line, and we did before, we send a hangup signal to all of the processes that are associated with this line, call *tdmodem ()* to hang up the line, flush the output queue for this line by calling *ttyflush ()*, and finally, mark the line as having no carrier.

tdioctl()

The *tdioctl()* routine is called when some process makes a *ioctl* system call on a device associated with the driver. It just calls *ttiocom ()* which returns a non-zero value if the hardware must be reconfigured.

```
        status = inb(addr + RLSR);

        /*
         *  Were there any errors on input?
         */
        if( status & SOERR )            /* overrun error */
                c |= OVERRUN;
        if( status & SPERR )            /* parity error */
                c |= PERROR;
        if( status & SFERR )            /* framing error */
                c |= FRERROR;
        (*linesw¢tp->t_line|.l_input)(tp, c, 0);
}

tdmint(dev)
{
        register struct tty *tp;
        register int addr,c;

        tp = &td_tty¢dev|;
        if ( tp->t_cflag & CLOCAL ) {
                return;,
        }
        addr = td_addr¢dev|;

        if ((inb(addr + RSTATUS & SDSR)) {
                if ((tp->t_state & CARR_ON)==0) {
                        tp->t_state |= CARR_ON;
                        wakeup(&tp->t_canq);
                }
        } else {
                if (tp->t_state & CARR_ON) {
                        if (tp->t_state & ISOPEN) {
                                signal(tp->t_pgrp, SIGHUP);
                                tdmodem(dev, TURNOFF);
                                ttyflush(tp, (FREAD|FWRITE));
                        }
                        tp->t_state &= °CARR_ON;
                }
        }
}

tdioctl(dev, cmd, arg, mode)
int dev;
int cmd;
faddr_t arg;
int mode;
{
        if (ttiocom(&td_tty¢dev|, cmd, arg, mode))
                tdparam(dev);
```

tdproc()

The *tdproc ()* routine is called to effect some change on the output, such as emitting the next character in the queue, or halting or restarting the output.

312. The *cmd* argument determines the action taken.

314. The time delay for outputting a break has finished. Reset the flag that indicates there is a delay in progress, and stop sending a continuous space. Then restart output by jumping to *start.*

321. We are either restarting a line on which output was stopped, or someone is waiting for the output queue to drain. Reset the flag indicating that output on this line is stopped, and start the output again by jumping to *start.*

326. Here, we are trying to put out another character. If some delay is in progress (TIMEOUT) or the line has output stopped (TTSTOP) or a character is in the process of being output (BUSY) give up.

328. If some process was waiting for the output queue to drain, reset the indicator, and wake him up.

332. While we still have characters in the output buffer do the following.

333. If we are doing output postprocessing on this line, and the current character is a delay marker (octal 200), get the next character, which specifies the delay in clock ticks, mark the line as waiting for a delay to expire, and schedule the line to be restarted via *timeout ().*

342. Otherwise, we have a character to output; mark the line BUSY, and pass the character to the controller.

346. If some process is waiting because the buffer went over the high water mark, and it is now below the low water mark, wake him up.

```
}

tdproc(tp, cmd)
register struct tty *tp;
{
        register c;
        register int addr;

        extern ttrstrt();

        addr = td_addr¢tp - td_tty|;
        switch (cmd) {

        case T_TIME:
                tp->t_state &= °TIMEOUT;
                outb(addr + RCtrl, inb(addr + RCtrl) & °CBREAK);
                goto start;

        case T_WFLUSH:
        case T_RESUME:
                tp->t_state &= °TTSTOP;
                goto start;

        case T_OUTPUT:
        start:
                if (tp->t_state&(TIMEOUT|TTSTOP|BUSY))
                        break;
                if (tp->t_state&TTIOW && tp->t_outq.c_cc==0) {
                        tp->t_state &= °TTIOW;
                        wakeup((caddr_t)&tp->t_oflag);
                }
                while ((c=getc(&tp->t_outq)) >= 0) {
                        if (tp->t_oflag&OPOST && c == 0200) {
                                if ((c = getc(&tp->t_outq)) < 0)
                                        break;
                                if (c > 0200) {
                                        tp->t_state |= TIMEOUT;
                                        timeout(ttrstrt,(caddr_t)tp,(c&0177));
                                        break;
                                }
                        }
                        tp->t_state |= BUSY;
                        outb(addr + RTDATA, c);
                        break;
                }
                if (tp->t_state&OASLP && tp->t_outq.c_cc<=
                ttlowat¢tp->t_cflag&CBAUD|) {
                tp->t_state &= °OASLP; wakeup((caddr_t)&tp->t_outq)&se
mi.
                }
                break;
```

352. We want to stop the output on this line. Since there is no way to stop the character we have already passed to the controller, we just flag the line stopped, and drop through.

357. We want to tell the device on the other end to stop sending characters. Reset the flag asking to stop the line and mark the line stopped. If the line is already busy, set the flag; otherwise, output a CSTOP character.

365. Someone is waiting to flush the input queue. If we haven't blocked the device from sending us characters, just return. Otherwise, drop through and unblock him.

369. We want 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 so we will send a CSTART later; otherwise, send the CSTART now.

377. We want 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.

```

        case T_SUSPEND:
                tp->t_state |= TTSTOP;
                break;

        case T_BLOCK:
                tp->t_state &= ~TTXON;
                tp->t_state |= TBLOCK;
                if (tp->t_state&BUSY)
                        tp->t_state |= TTXOFF;
                else
                        outb(addr + RDATA, CSTOP);
                break;

        case T_RFLUSH:
                if (!(tp->t_state&TBLOCK))
                        break;
        case T_UNBLOCK:
                tp->t_state &= ~(TTXOFF|TBLOCK);
                if (tp->t_state&BUSY)
                        tp->t_state |= TTXON;
                else
                        outb(addr + TDATA, CSTART);
                break;

        case T_BREAK:
                outb( addr + RCtrl, inb( addr + RCtrl ) | CBREAK );
                tp->t_state |= TIMEOUT;
                timeout(ttrstrt, tp, HZ/4);
                break;
        }
}
```

Sample Disk Driver

The driver presented here is for an intelligent controller that is attached to one or more drives. The controller can handle multiple sector transfers that cross track and cylinder boundaries.

13. NHD defines the number of drives the controller can be attached to.

14. 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.

21. The controller registers occupy a region of contiguous address space starting at RBASE and running through RBASE+7.

32. To make the controller perform some action, the registers that describe the transfer (RCYL, RTRK, RSEC, RADDRL, RADDRH, RCNT) are set to the appropriate values, and then the bit representing the desired action is written into the RCMD register.

40. *drive ()* and *part ()* are macros to split out the two parts of the minor number. Bits zero through two 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 ten decimal.

44. Large disks are typically broken into several partitions of a more manageable size. The structure that specifies the size of the partitions specifies the length of the partition in blocks, and the starting cylinder of the partition.

49. This driver splits a disk into up to eight pieces, but at present, only four are used. The first partition covers the whole disk. The remaining three split the disk three ways, one partition for each of root, swap, and usr.

```
 /*
 ** hd- prototype fixed disk driver
 */

#include ../h/param.h
#include ../h/systm.h
#include ../h/buf.h
#include ../h/dir.h
#include ../h/conf.h
#include ../h/user.h

/* disk parameters */
#define NHD     4       /* number of drives */
#define NCPD    600     /* # cylinders/disk */
#define NTPC    4       /* # tracks/cylinder */
#define NSPT    10      /* # sectors/track */
#define NBPS    512     /* # bytes/sector */
#define NBPC    (NTPC*NSPT*(BSIZE/NBPS))    /* blocks/cylinder */

/* addresses of controller registers */
#define RBASE   0x00        /* base of all registers */
#define RCMD    (RBASE+0)   /* command register */
#define RSTAT   (RBASE+1)   /* status - nonzero means error */
#define RCYL    (RBASE+2)   /* target cylinder */
#define RTRK    (RBASE+3)   /* target track */
#define RSEC    (RBASE+4)   /* target sector */
#define RADDRL  (RBASE+5)   /* target memory address lo 16 bits*/
#define RADDRH  (RBASE+6)   /* target memory address hi 8 bits*/
#define RCNT    (RBASE+7)   /* number of sectors to xfer */

/* bits in RCMD register */
#define CREAD   0x01        /* start a read */
#define CWRITE  0x02        /* start a write */
#define CRESET  0x03        /* reset the controller */

/*
** minor number layout is 0000dppp
**  where d is the drive number and ppp is the partition
*/
#define drive(d)    (d >> 3)
#define part(d)     (d & 0x07)

/* partition table */
struct partab {
    daddr_t len;            /* # of blocks in partition */
    int     cyloff;         /* starting cylinder of partition */
};

struct partab hd_sizes¢8| = {
    NCPD*NBPC,      0,          /* whole disk */
```

60. 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.*

61. Each block driver that wants to allow "raw" I/O allocates one buffer header for this purpose.

hdstrategy

This is called by the kernel to queue a request for I/O. The single argument is a pointer to the buffer header which contains all of the data relevant to the request. The strategy routine is responsible for validating the request, and linking it into the queue of outstanding requests.

79. First, compute various useful numbers that will be used repeatedly during the validation process.

82. If the request is for a non-existent drive, a non-existent partition, lies completely outside the specified partition, or is a write and ends outside the partition, it is marked as failed by setting the B__ERROR bit in the *b__flags* field of the header, and then marked completed 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.

95. Compute the target cylinder of the request for the benefit of the *disksort ()* routine.

96. Block interrupts to prevent the interrupt routine from changing the queue of outstanding requests.

97. Sort the request into the queue by passing it and the head of the queue to *disksort ().*

98. If the controller is not already active, start it up.

99. Re-enable interrupts and return to the user process.

```
    ROOTSZ*NBPC,    0,          /* root area */
    SWAPSZ*NBPC,    ROOTSZ,     /* swap area */
    USERSZ*NBPC,    USROFS,     /* usr area */
    0,      0,                  /* spare */
    0,      0,                  /* spare */
    0,      0,                  /* spare */
    0,      0,                  /* spare */
};

struct  buf     hdtab;          /* start of request queue */
struct  buf     rhdbuf;         /* header for raw i/o */

/*
**    Strategy Routine:
**    Arguments:
**      Pointer to buffer structure
**    Function:
**      Check validity of request
**      Queue the request
**      Start up the device if idle
*/
int hdstrategy(bp)
register struct buf *bp;
{
    register int dr, pa;    /* drive and partition numbers */
    int sspl;
    long sz;

    dr = drive(bp->b_dev);
    pa = part(bp->b_dev);
    sz = (sz + BMASK) >> BSHIFT;
    if ( dr<NDRIVES && pa<NPARTS && bn>=0 && bn<hd_sizes¢pa|.len &&
        ((bn + sz < hd_sizes¢pa|.len) || (bp->b_flags & B_
READ)))
    {
        if ( bn + sz > hd_sizes¢pa|.len ) {
            sz = (hd_sizes¢pa|.len - bn) * BSIZE;
            bp->b_resid = bp->b_bcount - sz;
            bp->b_bcount = sz;
        }
    } else {
        bp->b_flags |= B_ERROR;
        iodone(bp);
        return;
    }
    bp->b_cylin = (b_blkno / NBPC) + hd_sizes¢pa|.start;
    sspl = spl5();
    disksort(&hdtab, bp)
    if (dp->b_active == NULL)
        hdstart();
    splx(sspl);
```

hdstart()

The *hdstart ()* routine performs the calculation of target address on the disk, and starts the transfer.

117. If there are no active requests, mark the state of the driver as idle, and return.

121. Mark the state of the driver as active.

123. Calculate the starting cylinder, track, and sector of the request, and load the controller registers with these values.

129. Load the controller with the drive number, and the memory address of the data to be transferred.

132. If the request is a read request, issue a read command, otherwise, issue a write command.

hdintr()

The *hdintr ()* routine is called by the kernel whenever the controller issues an interrupt.

149. If we get called when we don't expect to, just return.

```
}

/*
 *      Startup Routine:
 *      Arguments:
 *        None
 *      Function:
 *        Compute device-dependent parameters
 *        Start up device
 *        Indicate request to I/O monitor routines
 */
hdstart()
{
    register struct buf *bp;      /* BUFFER POINTER */
    register unsigned sec;

    if ((bp = hdtab.b_actf) == NULL) {
        hdtab.b_active = 0;
        return;
    }
    hdtab.b_active = 1;

    sec = (unsigned)bp->blkno * (unsigned)(BSIZE / NBPS);
    out(RCYL, sec / NSPC);      /* cylinder */
    sec %= NSPC;
    out(RTRK, sec / NSPT);      /* track */
    out(RSEC, sec % NSPT);      /* sector */
    out(RCNT, bp->b_count / NBPS);  /* count */
    out(RDRV, drive(bp->b_dev));    /* drive */
    out(RADDRL, bp->b_paddr & 0xffff);  /* memory address lo */
    out(RADDRH, bp->b_paddr >> 16);     /* memory address hi */
    if ( bp->b_flags & B_READ )
        out(RCMD, CREAD);
    else
        out(RCMD, CWRITE);
}

/*
 *      Interrupt routine:
 *        Check completion status
 *        Indicate completion to i/o monitor routines
 *        Log errors
 *        Restart (on error) or start next request
 */
hdintr()
{
    register struct buf *bp;

    if (hdtab.b_active == 0)
        return;
```

152. Get a pointer to the first buffer header in the chain; this is the request that is currently being serviced.

154. If the controller indicates an error, and we haven't retried ERRLIM times, try the operation again. If we have retried ERRLIM times, we assume the error is a hard one, and mark the request as failed, and call *deverror ()* to print a console message about the failure.

166. Mark this request complete, take it out of the request queue, and call *hdstart ()* to start on the next request.

hdread()

The *hdread ()* routine is called by the kernel when a process requests raw read on the device. All it has to do is call *physio ()*, passing the name of the strategy routine, a pointer to the raw buffer header, the device number, and a flag indicating a read request. *Physio ()* performs all the necessary machinations, and queues the request by calling the strategy routine.

```

        bp = hdtab.b_actf;

        if ( in(RSTAT) != 0 )
            out(RCMD, CRESET);
            if (++hdtab.b_errcnt <= ERRLIM) {
                hdstart();
                return;
            }
            bp->b_flags |= B_ERROR;
            deverr(&hdtab, bp, in(RSTAT), 0);
        }
        /*
         *      Flag current request complete, start next one
         */
        hdtab.b_errcnt = 0;
        hdtab.b_actf = bp->av_forw;
        bp->b_resid = 0;
        iodone(bp);
        hdstart();
}

/*
 *  raw read routine:
 *    This routine calls physio which computes and validates
 *    a physical address from the current logical address.
 *
 *      Arguments
 *        Full device number
 *      Functions:
 *        Call physio which does the actual raw (physical) I/O
 *        The arguments to physio are:
 *          pointer to the strategy routine
 *          buffer for raw I/O
 *          device
 *          read/write flag
 */
hdread(dev)
{

        physio(hdstrategy, &rhdbuf, dev, B_READ);
}

/*
 *       Raw write routine:
 *       Arguments(to hdwrite):
 *         Full device number
 *       Functions:
 *         Call physio which does actual raw (physical) I/O
 */
```

hdwrite()

The *hdwrite ()* routine is called by the kernel when a process requests a raw write on the device. Its responsibilities and actions are exactly the same as *hdread ()* except that it passes a flag indicating a write request.

```
hdwrite(dev)
{

    physio(hdstrategy, &rhdbuf, dev, B_WRITE);
}
```

Appendix D. Linker Error Messages

This section lists and explains the messages displayed by the IBM Personal Computer XENIX linker. The linker, **ld**, displays a message whenever it encounters an error during processing.

Array element size mismatch

A far communal array has been declared with two or more different array element sizes (for example, declared once as an array of characters and once as an array of reals). Match definitions and recreate object module.

Attempt to put segment name in more than one group in file filename

A segment was declared to be a member of two different groups. Correct the source and recreate the object files.

Cannot find file filename

Specified file cannot be found. Try again after locating the file in question.

Cannot open list file

The directory or disk is full. Make space on the disk or in the directory.

Cannot open run file

The directory or disk is full. Make space on the disk or in the directory.

Cannot open temporary file

The directory or disk is full. Make space on the disk or in the directory.

Common area longer than 65536 bytes

User's program has more than 64K of communal variables. At the present time, only C language programs can possibly cause this message to be displayed. Rewrite your program using fewer communal variables or making some of your communal variables far&sem

Data record too large

LEDATA record contains more than 1024 bytes of data. This is a translator error.

Dup record too large

LIDATA record contains more than 512 bytes of data. Most likely, an assembly module contains a struc definition that is very complex, or a series of deeply nested DUP statements (for example, table db 10 dup(11 dup (12 dup (13 dup (. . .))))). Simplify and reassemble.

Error accessing library

File in question is an invalid library. Use a valid library.

Fixup overflow nearnum in segment name in filename(name) offset num

A fixup overflow can be caused by:

1. A group larger than 64K bytes.

2. The user's program contains an intersegment short jump or intersegment short call.

3. The user has a data item whose name conflicts with that of a subroutine in a library included in the link.

4. An assembly language source file has an EXTRN declaration for a far procedure inside the body of a segment.

Group name larger than 64K bytes

User has defined a group containing more than 64K bytes of code or data. Make the offending group smaller and relink.

Invalid object module

One of the object modules is invalid. Try recompiling.

List file name missing

Name missing after -m option. Try again with correct command line.

Multiple code segments--should be medium model

User's program contains more than one code segment, and the user has not informed the linker that the program is middle or large model. Unless the program is hybrid model, relink using -Mm option.

Multiple data segments--should be large model

User's program contains more than one data segment, and the user has not informed the linker that the program is large model. Unless the program is hybrid model, relink using -Ml option.

Name length missing

Number missing after the -nl option. Try again with correct command line.

NEAR/HUGE conflict

Conflicting near and huge definitions for a communal variable. Revise definitions to be consistent

Note: A communal variable is huge if it is larger than 65536 bytes.

No object files specified

No object files were specified on the command line and the -u option was not used. Try again with correct command line.

No object modules specified

User failed to supply the linker with any object file names. Try again.

Out of space on list file

Disk on which list file is being written is full. Free more space on the disk and try again.

Out of space on run file

Disk on which executable is being written is full. Free more space on the disk and try again.

Out of space on scratch file

Disk in default drive is full. Delete some files on that disk, or replace with another diskette, and restart the linker.

Relocation table overflow

More than 16384 long calls or long jumps or other long pointers in the user's program. Rewrite program replacing long references with short references where possible and recreate object module.

Run file name missing

Name missing after the -o option. Try again with correct command line.

Segment limit set too high

The limit on the number of segments allowed was set higher than 1024 using the -S option. Try link again with a smaller number.

Segment limit too high

There is insufficient memory for the linker to allocate tables to describe the number of segments requested (either the value specified with -S or the default: 128). Try the link again using -S to select a smaller number of segments (for example, 64, if the default were used previously).

Segment size exceeds 64K

User has a small model program with more than 64K bytes of code, or user has a middle model program with more than 64K bytes of data. Try compiling and linking middle or large model.

Stack size exceeds 65536 bytes
The value specified using the -F option exceeds 0x10000. Try again.

Stack size missing
Number missing after -F option. Try again with correct command line.

Symbol missing
Symbol missing after the -u option. Try again with correct command line.

Symbol table overflow
The user's program has greater than 256K of symbolic information (publics, externs, segments, groups, classes, files, etc). Combine modules and/or segments and recreate the object files. Eliminate as many public symbols as possible.

Terminated by user
The user pressed the delete key.

Too many external symbols in one module
User's object module specified more than the allowed number of external symbols. Break up the module.

Too many group-, segment-, and class-names in one module
User's program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes, and recreate the object files.

Too many groups
User's program defines more than nine groups. Reduce the number of groups.

Too many GRPDEFs in one module
Linker encountered more than 9 GRPDEFs in a single module. Reduce the number of GRPDEFs or split up the module.

Too many libraries

User tried to link with more than 32 libraries. Combine libraries or link modules that require fewer libraries.

Too many segments in one module

The user's object module has more than 255 segments. Split the modules or combine segments.

Too many segments

The user's program has too many segments. Relink using the -S option with an appropriate number of segments specified.

Too many TYPDEFs

TYPDEFs are records emitted by the compiler to describe communal variables. Create two sources from the old source, dividing the communal variable definitions between them; recompile and relink.

Unexpected end-of-file on library

The diskette containing the library has probably been removed. Try again after replacing the diskette with the library.

Unknown model specifier +′&-M x′

x was none of the following: s, m, or l. Try again with correct command line.

Unknown option '&-x'

Specified option is not recognized by the linker. Try again with correct command line.

Unrecognized XENIX version number

Number after -v option was neither 2 nor 3. Try again with correct command line.

Use -i option

User's program is not small model impure (that is, it consists of more than one segment). Relink using the -i option.

Version number missing

Number missing after -v option. Try again with correct command line.

Warning: Groups name and name overlap

User's program contains overlapping groups. Unless one group is completely contained by the other, fix the source code, recompile, and relink.

Warning: model mismatch

One or more object modules were not compiled using the memory model specified by the -M option. Recompile the offending module and relink.

Warning: too many public symbols

The user has asked for a sorted listing of public symbols in the list file, but there are too many symbols to sort. The linker will produce an unsorted listing of the public symbols.

-u seen before -nl

User has specified a symbol to look for (using the -u option) before specifying the maximum symbol length with the -nl option. Try again placing the -nl option and its argument before all -u options and their arguments.

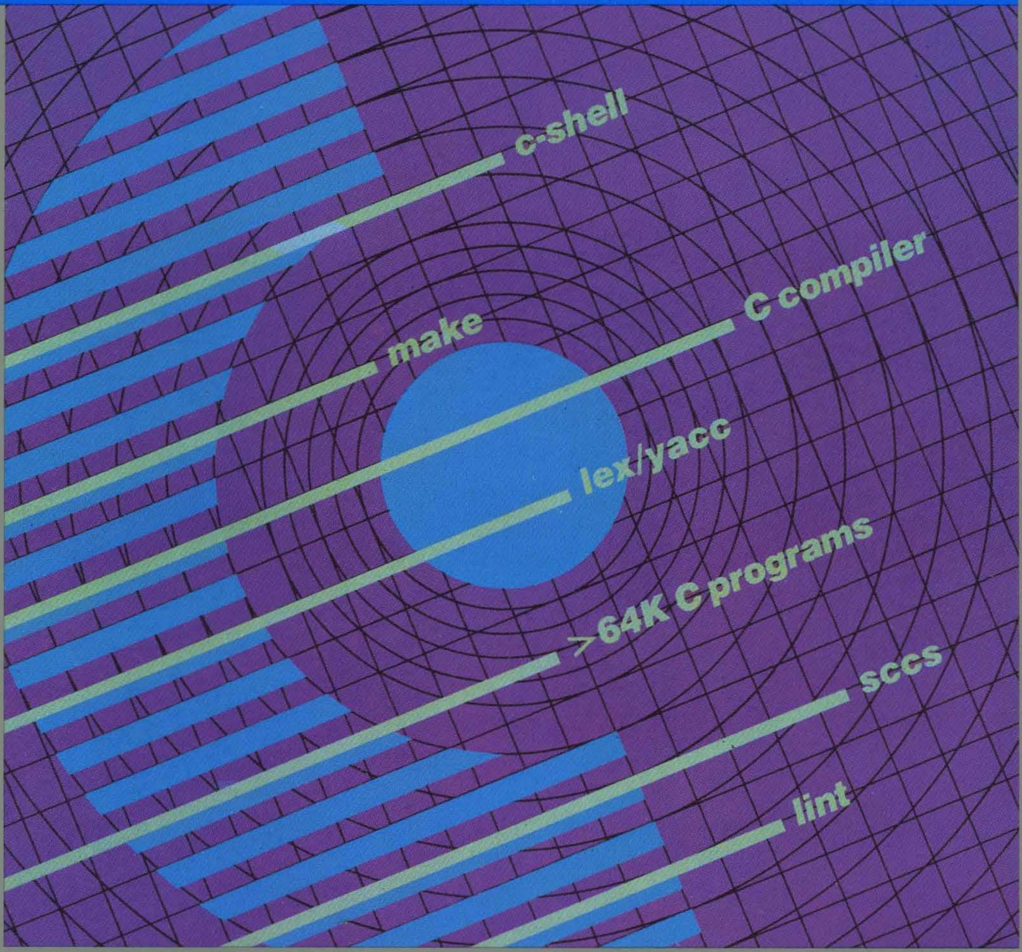
Index

6138694

Software Development Guide

IBM Personal Computer XENIX™ Software Development System

Programming Family



IBM

Personal Computer Software

Software development tools, including language translators, source code management tools, a C compiler, 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.

Software required:

IBM Personal Computer XENIX™ Operating System

Software included:

Three 1.2MB diskettes

System requirements:

IBM Monochrome or Color Display or equivalent (with appropriate adapter)

IBM Personal Computer AT™

512KB RAM memory

IBM 20MB fixed disk

IBM 1.2MB diskette drive

Note:
XENIX is a trademark of Microsoft Corporation.

International Business Machines Corporation
P.O. Box 1328-S
Boca Raton, Florida 33432

Printed in the United States of America

6024209

International Business Machines Corporation **Boca Raton, Florida 33432**
IBM Program License Agreement

You should carefully read the following terms and conditions before opening this diskette package. Opening this diskette 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. 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 a single machine;
b. copy the program into any machine readable or printed form for backup or modification purposes in support of your use of the program on the single machine (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 and license to another party if the other party agrees to accept 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.

You must reproduce and include the copyright notice on any copy, modification or portion merged into another program.

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 license.

If you transfer possession of any copy, modification or merged portion of the program to another party, your license is automatically terminated.

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

The program is provided "as is" without warranty of any kind, either expressed or implied, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. 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 authorized Personal Computer dealer) assume the entire cost of all necessary servicing, repair or correction.

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.

IBM does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error free.

However, IBM warrants the diskette(s) or cassettes on which the program is furnished, to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

Limitations of Remedies

IBM's entire liability and your exclusive remedy shall be:

1. the replacement of any diskette or cassette not meeting IBM's "Limited Warranty" and which is returned to IBM or an authorized **IBM Personal Computer** dealer with a copy of your receipt.
or
2. if IBM or the dealer is unable to deliver a replacement diskette or cassette which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

In no event will IBM be liable to you for any damages, including any lost profits, lost savings or other incidental or consequential damages arising out of the use or inability to use such program even if IBM or an authorized IBM Personal Computer dealer 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.

General

You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the State of Florida.

Should you have any questions concerning this Agreement, you may contact IBM by writing to IBM Personal Computer, Sales and Service, P.O. Box 1328-W Boca Raton, Florida 33432.

You acknowledge that you have read this agreement, understand it and agree to be bound by its terms and conditions. You further agree that it is the complete and exclusive statement of the agreement between us which supercedes any proposal or prior agreement, oral or written, and any other communications between us relating to the subject matter of this agreement.