sun®
microsystems

Programming Utilities & Libraries

Part Number: 800-1774-15
Revision A, of 9 May 1988

Credits and Acknowledgements

UNIX Programming
by Brian W. Kernighan and Dennis M. Ritchie, Bell Laboratories, Murray Hill, New Jersey.

Lint, a C Program Checker
by S. C. Johnson, Bell Laboratories, Murray Hill, New Jersey.

Make — A Program for Maintaining Computer Programs
by S. I. Feldman, Bell Laboratories, Murray Hill, New Jersey.

The M4 Macro Processor
by Brian W. Kernighan and Dennis M. Ritchie, Bell Laboratories, Murray Hill, New Jersey.

Lex — A Lexical Analyzer Generator
by M. E. Lesk and E. Schmidt, Bell Laboratories, Murray Hill, New Jersey.

Yacc — Yet Another Compiler-Compiler
by Stephen C. Johnson, Bell Laboratories, Murray Hill, New Jersey.

Source Code Control System User's Guide
by L. E. Bonanni and C. A. Salemi, Bell Laboratories, Piscataway, New Jersey.

Source Code Control System
by Eric Allman, Formerly of Project Ingres, University of California at Berkeley.

Curses — A Screen Updating and Cursor Movement Optimization Library Package
by Kenneth C. R. C. Arnold, of the University of California at Berkeley.

Trademarks

Multibus® is a trademark of Intel Corporation.

SunOS™, Sun Workstation®, as well as the word ''Sun'' followed by a numerical suffix, are trademarks of Sun Microsystems, Incorporated.

UNIX® and UNIX System V® are trademarks of Bell Laboratories.

OS/360® is a trademark of International Business Machines, Incorporated.

VAX®, VMS®, and PDP-11® are trademarks of Digital Equipment Corporation.

Multics™ is a trademark of Honewell International.

All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations.

Copyright © 1987, 1988 by Sun Microsystems, Inc.

This publication is protected by Federal Copyright Law, with all rights reserved. No part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, or by any means manual, electric, electronic, electro-magnetic, mechanical, chemical, optical, or otherwise, without prior explicit written permission from Sun Microsystems.

Contents

Tables

Figures

1

Introduction

1

Introduction

This manual provides an overview of the SunOS programming environment, and describes a variety of system facilities, utility commands, and libraries, that are of interest primarily to applications developers.

The first portion of the manual describes system interface and support facilities for use with the C programming language:

□ Chapter 2: SunOS Programming

System interface and standard library support facilities

□ Chapter 3: System V Compatibility Package

Comparison of BSD 4.x and System V, and SunOS conformance with the System V Interface Definition (SVID)

□ Chapter 4: Shared Libraries

Overview, system support and development techniques

The next two chapters describe C programming aids to check for correctness and monitor performance:

□ Chapter 5: `lint` — a Program Verifier for C

Checking programs for internal consistency and portability

□ Chapter 6: Performance Analysis

Timing, Profiling and Coverage Analysis tools

The next two chapters describe system utilities for version control and consistent compilation:

□ Chapter 7: SCCS — Source Code Control System

Version control for source files

□ Chapter 8: `make` User's Guide

Consistent compilation for programs and software projects

The next three chapters describe program-generation tools:

□ Chapter 9: `m4` — a Macro Processor

Parametric macro-language (pre)processor

- Chapter 10: `lex` — a Lexical Analyzer Generator

  Preprocessor for scanning routines

- Chapter 11: `yacc` — Yet another Compiler Compiler

  Preprocessor for parsing routines

The last two chapters describe the BSD and System V `curses` terminal-display library routines.

- Chapter 12: `curses` Library: Screen-Oriented Cursor Motions BSD `curses` library routines

- Chapter 13: System V `curses` and `terminfo`

  System V display library and terminal-capabilities database.

Appendix A describes low-level commands for SCCS.

Appendix B summarizes the enhancements made to the SunOS version of the `make` utility.

For detailed information about SunOS utilities, library functions, file- and device-level facilities, and other information about the operating system, refer to the *SunOS Reference Manual.*

1.1. Bibliography and Acknowledgements

This manual has been derived in large part from sources that include technical papers distributed with U.C. Berkeley's BSD release, System V Release 3 documentation, and others. In particular, Sun Microsystems wishes to acknowledge the following sources:

1. L. E. Bonanni and C. A. Salemi, *Source Code Control System User's Guide*, Bell Laboratories, Piscataway, New Jersey.

2. Eric Allman, *Source Code Control System* Formerly of Project Ingres, University of California at Berkeley.

3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.

4. B. W. Kernighan, *UNIX for Beginners — Second Edition*, Bell Laboratories, 1978.

5. S. C. Johnson, *Yacc — Yet Another Compiler-Compiler*, Bell Laboratories Computing Science Technical Report #32, July 1978.

6. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, N. J. (1978).

7. S. C. Johnson and D. M. Ritchie, *'UNIX Time-Sharing System: Portability of C Programs and the UNIX System', Bell Sys. Tech. J.* 57(6) pp. 2021-2048 (1978).

8. S. C. Johnson, *'A Portable Compiler: Theory and Practice', Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).

9. Susan L. Graham, Peter B. Kessler, and Marshall Kirk McKusick, *Gprof — A Call Graph Execution Profiler*, Computer Science Division, Electrical Engineering and Computer Science Department, University of California, Berkeley, California 94720.

   Editor's note:

   > This paper is for the scholar inertested in the theory behind call-graph profiling.

10. Brian W. Kernighan and Dennis M. Ritchie, *The M4 Macro Processor*, Bell Laboratories, Murray Hill, New Jersey.

    Author's note:

    > We are indebted to Rick Becker, John Chambers, Doug McIlroy, and especially Jim Weythman, whose pioneering use of *m4* has led to several valuable improvements. We are also deeply grateful to Weythman for several substantial contributions to the code. The *m4* macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer.

11. B. W. Kernighan and P. J. Plauger, ,*Software*Tools Addison-Wesley, Inc., 1976.

12. M. E. Lesk, *Lex — A Lexical Analyzer Generator*, Computing Science Technical Report #39, October 1975.

    Author's note:

    > [The] outside of `lex` is patterned on *yacc* and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of *lex*, as well as debuggers of it. Many thanks are due to both. The current version of `lex` was designed, written, and debugged by Eric Schmidt.

13. A. V. Aho and M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search*, Comm. ACM **18**, 333-340 (1975).

14. S. C. Johnson, *Yacc: Yet Another Compiler-Compiler*, Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).

15. Kenneth C.R.C. Arnold, *Screen Updating and Cursor Movement Optimization: A Library Package*, Bell Laboratories, Murray Hill, New Jersey.

    Editor's note:

    > The `curses` library was implemented by Ken Arnold, based on the screen-updating and optimizing routines originally written by Bill Joy for the `vi` editor.

    Author's note:

    > This package would not exist without the work of Bill Joy, who, in writing his editor, created the capability to generally describe terminals, wrote the routines which read this database [and] implement optimal cursor movement [. . .] Doug Merritt and Kurt Shoens also were extremely important, as were [. . .] Ken Abrams, Alan Char, Mark Horton and and Joe Kalash.

2

SunOS Programming

2

SunOS Programming

This chapter is an introduction to programming on the SunOS system. The emphasis is on how to write programs that make use of system calls and library functions. The topics discussed include

□ handling command-line arguments

□ rudimentary I/O; the standard input and output

□ the standard I/O library; file system access

□ low-level I/O: open, read, write, close, seek

□ processes: exec, fork, pipes

□ signals — interrupts, etc.

Section 2.7 — *The Standard I/O Library* — describes the standard I/O library in detail.

This chapter describes how to write programs that interface with the SunOS operating system in a nontrivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution. It summarizes material that is described in detail in the *SunOS Reference Manual.*

There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language*. You should also be familiar with SunOS itself.

2.1. Basics

Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function `main()` as an argument count `argc` and an array `argv` of pointers to character strings that contain the arguments. By convention, `argv[0]` is the command name itself, so `argc` is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal — this is essentially the `echo()` command.

```
main(argc, argv)     /* echo arguments */
int argc;
char *argv[ ];
{
    int i;

    for (i = 1;  i < argc;  i++)
        printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

`argv` is a pointer to an array whose elements are pointers to arrays of characters; each is terminated by `\0`, so they can be treated as strings. The program starts by printing `argv[1]` and loops until it has printed `argv[argc-1]`.

The argument count and the arguments are parameters to `main()`. If you want to keep them around so other routines can get at them, you must copy them to external variables.

2.2. Standard Input and Standard Output

The simplest input mechanism is to read from the *standard input*, which is generally the user's terminal. The function `getchar()` returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention (input redirection): if `prog` uses `getchar()`, the command line

```
tutorial% prog < filename
```

makes `prog` read from the file specified by *filename* instead of the terminal. `prog` itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the *pipe* mechanism:

```
tutorial% otherprog | prog
```

provides the standard input for `prog` from the standard output (see below) of `otherprog`.

`getchar()` returns the value `EOF` when it encounters the end of file (or an error) on whatever you are reading. The value of `EOF` is normally defined to be `-1`, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, `putchar(c)` puts the character *c* on the 'standard output', which is also by default the terminal. The output can be captured on a file by using >: if `prog` uses `putchar()`,

```
tutorial% prog > outputfile
```

writes the standard output on *outputfile* instead of the terminal. *outputfile* is created if it doesn't exist; if it already exists, its previous contents are overwritten. A pipe can be used:

```
tutorial% prog | otherprog
```

puts the standard output of `prog` into the standard input of `otherprog`.

The function `printf()`, which formats output in various ways, uses the same mechanism as `putchar()` does, so calls to `printf()` and `putchar()` may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function `scanf()` provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. `scanf()` uses the same mechanism as `getchar()`, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with `getchar()`, `putchar()`, `scanf()`, and `printf()` may be entirely adequate, and it is almost always enough to get started. This is particularly true if the SunOS pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ASCII control characters from its input (except for newline and tab).

```
#include <stdio.h>

main()    /* ccstrip: strip non-graphic characters */
{
     int c;
     while ((c = getchar()) != EOF)
          if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
               putchar(c);
     exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file which does I/O using the standard I/O functions described in section 3(S) of the *SunOS Reference Manual* — the C compiler reads a file (*/usr/include/stdio.h*) of standard routines and symbols that includes the definition of EOF.

If it is necessary to treat multiple files, you can use **cat** to collect the files for you:

```
tutorial% cat file1 file2 ... | ccstrip > output
```

and thus avoid learning how to access files from a program. By the way, the call to `exit()` at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Section 2.5.3 discusses returning status in more detail.

2.3. The Standard I/O Library

The 'Standard I/O Library' is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

This section discusses the basics of the standard I/O library. Section 2.7 — *The Standard I/O Library* — contains a more complete description of its capabilities and calling conventions.

Accessing Files

The above programs have all read the standard input and written the standard output, which we have assumed are magically predefined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is wc, which counts the lines, words and characters in a set of files. For instance, the command

```
tutorial% wc x.c y.c
```

displays the number of lines, words and characters in x.c and y.c and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the filenames to the I/O statements which actually read the data.

The rules are simple — you have to *open* a file by the standard library function fopen() before it can be read from or written to. fopen() takes an external name (like x.c or y.c), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including *stdio.h* is a structure definition called FILE. The only declaration needed for a file pointer is exemplified by

```
FILE    *fp, *fopen();
```

This says that fp is a pointer to a FILE, and fopen() returns a pointer to a FILE. FILE is a type name, like int, not a structure tag.

The actual call to fopen() in a program has the form:

```
fp = fopen(name, mode);
```

The first argument of fopen() is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The allowable modes are read ("r"), write ("w"), or append (a).

In addition, each *mode* may be followed by a + sign to open the file for reading and writing. r+ positions the stream at the beginning of the file, w+ creates or truncates the file, and a+ positions the stream to the end of the file. Both reads and writes may be used on read/write streams, with the limitation that an `fseek()`, `rewind()`, or reading end-of-file must be used between a read and a write or vice versa.

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing discards the old contents. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, `fopen()` returns the null pointer value `NULL` — defined as zero in *stdio.h*.

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc()` and `putc()` are the simplest. `getc()` returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in `c` the next character from the file referred to by `fp`; it returns `EOF` when it reaches end of file. `putc()` is the inverse of `getc()`:

```
putc(c, fp)
```

puts the character `c` on the file `fp` and returns `c` as its value. `getc()` and `putc()` return `EOF` on error.

When a program is started, three streams are opened automatically, and file pointers are provided for them. These streams are the standard input, the standard output, and the standard error output; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. `stdin`, `stdout` and `stderr` are predefined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type `FILE *` can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write `wc`. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used standalone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)    /* wc: count lines, words, chars */
int argc;
char *argv[ ];
{
    int c, i, inword;
    FILE *fp, *fopen();
    long linect, wordct, charct;
    long tlinect = 0, twordct = 0, tcharct = 0;

    i = 1;
    fp = stdin;
    do {
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
            fprintf(stderr, "wc: can't open %s\n", argv[i]);
            continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
            charct++;
            if (c == '\n')
                linect++;
            if (c == ' ' || c == '\t' || c == '\n')
                inword = 0;
            else if (inword == 0) {
                inword = 1;
                wordct++;
            }
        }
        printf("%7ld %7ld %7ld", linect, wordct, charct);
        printf(argc > 1 ? " %s\n" : "\n", argv[i]);
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
    } while (++i < argc);
    if (argc > 2)
        printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
    exit(0);
}
```

The function `fprintf()` is identical to `printf()`, save that the first argument is a file pointer that specifies the file to be written.

The function `fclose()` is the inverse of `fopen()`; it breaks the connection between the file pointer and the external name that was established by `fopen()`, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is another reason to call `fclose()` on an output file — it flushes the buffer in which `putc()` is collecting output. `fclose()` is called automatically for each open file when a program terminates normally.

Error Handling — Stderr and Exit

`stderr` is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` appears on the user's terminal even if the standard output is redirected, unless the standard error is also redirected. *wc* writes its diagnostics on `stderr` instead of `stdout` so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The argument of `exit()` is made available to whatever process called the process that is exiting (see Section 2.5.3, so the success or failure of the program can be tested by another program that uses this one as a subprocess. By convention, a return value of 0 signals that all is well; nonzero values signal abnormal situations.

`exit()` itself calls `fclose()` for each open output file, to flush out any buffered output, then calls a routine named `_exit()`. The function `_exit()` terminates the program immediately without any buffer flushing; it may be called directly if desired.

Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those illustrated above.

Normally output with `putc()`, and such is buffered — use `fflush(fp)` to force it out immediately.

`fscanf()` is identical to `scanf()`, except that its first argument is a file pointer (as with `fprintf()`) that specifies the file from which the input comes; it returns `EOF` at end of file.

The functions `sscanf()` and `sprintf()` are identical to `fscanf()` and `fprintf()`, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for `sscanf()` and into it for `sprintf()`, and no input or output is done.

`fgets(buf, size, fp)` copies the next line from `fp`, up to and including a newline, into `buf`; at most `size-1` characters are copied; it returns `NULL` at end of file. `fputs(buf, fp)` writes the string in `buf` onto file `fp`.

The function `ungetc(c, fp)` 'pushes back' the character `c` onto the input stream `fp`; a subsequent call to `getc()`, `fscanf()`, etc., will encounter `c`. Only one character of pushback per file is permitted.

2.4. Low-Level I/O Functions

This section describes the bottom level of I/O on the SunOS system. The lowest level of I/O in SunOS provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

File Descriptors

In the SunOS operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called 'opening' the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so — does the file exist? Do you have permission to access it? — if all is well, returns a small positive integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. This is roughly analogous to the use of `READ(5,...)` and `WRITE(6,...)` in FORTRAN. All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in Section 2.3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the 'shell') runs a program, it opens three files, with file descriptors 0, 1, and 2, called standard input, standard output, and standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without opening the files.

If I/O is redirected to and from files with < and >, as in

```
tutorial% prog < infile > outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

`read()` and `write()`

All input and output is done by two functions called `read()` and `write()`. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are below:

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than **n** bytes remained to be read. When the file is a terminal, `read()` normally reads only up to the next newline, which is generally less than what was requested. A return value of zero bytes implies end of file, and `-1` indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ('unbuffered'), and 1024, corresponding to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character-at-a-time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define BUFSIZE 1024

main()  /* copy input to output */
{
    char    buf[BUFSIZE];
    int n;

    while ((n = read(0, buf, BUFSIZE)) > 0)
        write(1, buf, n);
    exit(0);
}
```

If the file size is not a multiple of `BUFSIZE`, some `read()` will return a smaller number of bytes, and the next call to `read()` after that will return zero.

It is instructive to see how `read()` and `write()` can be used to construct higher-level routines like `getchar()`, `putchar()`, etc. For example, here is a version of `getchar()` which does unbuffered input.

```
#define CMASK   0377    /* for making char's > 0 */

getchar()    /* unbuffered single character input */
{
    char c;

    return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c *must* be declared char, because read() accepts a character pointer. The character being returned must be masked with 0377 to ensure that it is positive; otherwise sign extension may make it negative. The constant 0377 is appropriate for the Sun but not necessarily for other machines.

The second version of getchar() does input in big chunks, and hands out the characters one at a time:

```
#define CMASK   0377    /* for making char's > 0 */
#define BUFSIZE 1024

getchar()   /* buffered version */
{
    static char buf[BUFSIZE];
    static char *bufp = buf;
    static int  n = 0;

    if (n == 0) {   /* buffer is empty */
        n = read(0, buf, BUFSIZE);
        bufp = buf;
    }
    return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

open(), creat(), close(), and unlink()

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, open() and creat().

open() is rather like the fopen() discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an int.

```
int fd;

fd = open(name, rwmode);
```

As with fopen(), the name argument is a character string corresponding to the external file name. The access mode argument is different, however: rwmode is 0 for read, 1 for write, and 2 for read and write access. open() returns -1 if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to open() a file that does not exist. The entry point creat() is provided to create new files, or to rewrite old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it could create the file called name, and -1 if not. If the file already exists, creat() will truncate it to zero length; it is not an error to creat() a file that already exists.

If the file is brand new, `creat()` creates it with the *protection mode* specified by the `pmode` argument. In the SunOS file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the SunOS utility *cp*, a program which copies one file to another. The main simplification is that our version copies only one file, and does not permit the second argument to be a directory:

```
#define NULL 0
#define BUFSIZE 1024
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)  /* cp: copy f1 to f2 */
int argc;
char *argv[ ];
{
     int f1, f2, n;
     char buf[BUFSIZE];

     if (argc != 3)
          error("Usage: cp from to", NULL);
     if ((f1 = open(argv[1], 0)) == -1)
          error("cp: can't open %s", argv[1]);
     if ((f2 = creat(argv[2], PMODE)) == -1)
          error("cp: can't create %s", argv[2]);

     while ((n = read(f1, buf, BUFSIZE)) > 0)
          if (write(f2, buf, n) != n)
               error("cp: write error", NULL);
     exit(0);
}

error(s1, s2) /* print error message and die */
char *s1, *s2;
{
     printf(s1, s2);
     printf("\n");
     exit(1);
}
```

As we said earlier, there is a limit (typically 20-32) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to reuse file descriptors. The routine `close()` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via `exit()` or return from the main program closes all open files.

The function `unlink(filename)` removes the file `filename` from the file system.

Random Access — seek() and lseek

File I/O is normally sequential: each read() or write() takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call lseek() provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is fd to move to position offset, which is taken relative to the location specified by origin. Subsequent reading or writing will begin at that position. offset is a long; fd and origin are int's. origin can be 0, 1, or 2 to specify that offset is to be measured from the beginning, from the current position, or from the end of the file, respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ('rewind'),

```
lseek(fd, 0L, 0);
```

Notice the 0L argument; it could also be written as (long) 0.

With lseek(), it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
     lseek(fd, pos, 0);     /* get to pos */
     return(read(fd, buf, n));
}
```

Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of –1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external variable errno. The meanings of the various error numbers are listed in *intro*(2) in the *SunOS Reference Manual* so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to display the reason for failure. The routine perror() displays a message associated with the value of errno; more generally, sys_errno is an array of character strings which can be indexed by errno and displayed by your program.

2.5. Processes

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

The `system()` Function

The easiest way to execute a program from another is to use the standard library routine `system()`. `system()` takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to timestamp the output of a program,

```
main( ) {
    system("date"); /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of `sprintf()` may be useful.

Remember that `getc()` and `putc()` normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use `fflush()`; for input, see `setbuf()` in section 2.7.

Low-Level Process Creation — `execl()` and `execv()`

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's `system()` routine is based on[1].

The most basic operation is to execute another program *without returning*, by using the routine `execl()`. To display the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to `execl()` is the *filename* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a placeholder. If the command takes arguments, they are strung out after this; the end of the list is marked by a `NULL` argument.

The `execl()` call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to start the second pass simply by an `execl()` call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where `date` is located, you might try the following calls.

[1] `system()` uses `/bin/sh` (the Bourne shell) to execute the command string, so syntax specific to the C shell will not work.

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of execl() called execv() is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where argp is an array of pointers to the arguments; the last pointer in the array must be NULL so execv() can tell where the list ends. As with execl(), filename is the file in which the program is found, and argp[0] is the name of the program. (This arrangement is identical to the argv array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like <, >, *, ?, and [ ] in the argument list. If you want these, use execl() to invoke the shell sh, which then does all the work. Construct a string commandline that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, **/bin/sh**. Its argument **-c** says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in commandline.

Control of Processes — fork() and wait()

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with execl() or execv(). Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called fork():

```
proc_id = fork( );
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of proc_id, the 'process id.' In one of these processes (the 'child'), proc_id is zero. In the other (the 'parent'), proc_id is nonzero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
     execl("/bin/sh", "sh", "-c", cmd, NULL);  /* in child */
```

And in fact, except for handling errors, this is sufficient. The fork() makes two copies of the program. In the child, the value returned by fork() is zero, so it calls execl() which does the command and then dies. In the parent, fork() returns nonzero so it skips the execl(). If there is any error, fork() returns −1.

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function wait():

```
int status;

if (fork( ) == 0)
    execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the execl() or fork(), or the possibility that there might be more than one child running simultaneously. The wait() returns the process id of the terminated child, if you want to check it against the value returned by fork(). Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in status). Still, these three lines are the heart of the standard library's system() routine, which we'll show in a moment.

The status returned by wait() encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and nonzero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to exit() which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up to point at the right files (see Section 2.4.1), and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither fork() nor the exec() calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the execl(). Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other process reads from the pipe. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
tutorial% ls | pr
```

which connects the standard output of ls to the standard input of pr. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we illustrate how the pipe connection is established and used.

The system call pipe() creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int     fd[2];

stat = pipe(fd);
if (stat == -1)
    /* there was an error ... */
```

fd is an array of two file descriptors, where fd[0] is the read side of the pipe and fd[1] is for writing. These may be used in read(), write() and close() calls just like any other file descriptors.

If a process reads a pipe which is empty, it waits until data arrives; if a process writes into a pipe which is too full, it waits until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent read() will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called popen(cmd, mode), which creates a process cmd (just as system() does), and returns a file descriptor that will either read or write that process, according to mode. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the pr command; subsequent write() calls using the file descriptor fout will send their data to that process through the pipe.

popen() first creates the pipe with a pipe() system call; it then fork()'s to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via execl()) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define  READ 0
#define  WRITE    1
#define  tst(a, b)     (mode == READ ? (b) : (a))
static   int popen_pid;

popen(cmd, mode)
char *cmd;
int mode;
{
     int p[2];

     if (pipe(p) < 0)
          return(NULL);
     if ((popen_pid = fork( )) == 0) {
          close(tst(p[WRITE], p[READ]));
          close(tst(0, 1));
          dup(tst(p[READ], p[WRITE]));
          close(tst(p[READ], p[WRITE]));
          execl("/bin/sh", "sh", "-c", cmd, 0);
          _exit(1);     /* disaster has occurred if we get here */
     }
     if (popen_pid == -1)
          return(NULL);
     close(tst(p[READ], p[WRITE]));
     return(tst(p[WRITE], p[READ]));
}
```

The sequence of close()'s in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first close() closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The close() closes file descriptor 0, that is, the standard input. dup() is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the dup() is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input[2]. Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write to the parent instead of reading. You may find it a useful exercise to step through that case.

[2] Yes, this is a bit tricky, but it's a standard idiom.

The job is not quite done, for we still need a function `pclose()` to close the pipe created by `popen()`. The main reason for using a separate function rather than `close()` is that it is desirable to wait for the termination of the child process. First, the return value from `pclose()` indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the `wait()` lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)    /* close pipe fd */
int fd;
{
     register r, (*hstat)( ), (*istat)( ), (*qstat)( );
     int   status;
     extern int popen_pid;

     close(fd);
     istat = signal(SIGINT, SIG_IGN);
     qstat = signal(SIGQUIT, SIG_IGN);
     hstat = signal(SIGHUP, SIG_IGN);
     while ((r = wait(&status)) != popen_pid && r != -1);
     if (r == -1)
          status = -1;
     signal(SIGINT, istat);
     signal(SIGQUIT, qstat);
     signal(SIGHUP, hstat);
     return(status);
}
```

The calls to `signal()` make sure that no interrupts, etc. interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable `popen_pid`; it really should be an array indexed by file descriptor. A `popen()` function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

2.6. Signals — Interrupts and All That

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside world signals: *interrupt* and *quit*, which are generated from the keyboard[3], *hangup*, caused by hanging up the phone on dialup lines, and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal — the signal terminates the process unless other arrangements have been made. In the *quit* case, a core image file is written for debugging purposes.

[3] The current binding of characters and signals can be discovered by the `stty all` command. On Sun systems, typing control-C usually generates the `kill` signal and control-\ generates the `quit` signal.

signal() is the routine which alters the default action. signal() has two arguments: the first specifies the signal to be processed, and the second argument specifies what to do with that signal. The first argument is just a numeric code, but the second is either a function, or a somewhat strange code that requests that the signal either be ignored or that it be given the default action. The include file *signal.h* gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
...
signal(SIGINT, SIG_IGN);
```

means that interrupts are ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, signal() returns the previous value of the signal. The second argument to signal() may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used so that the program can clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main( )
{
    int onintr( );

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);

    /* Process ... */

    exit(0);
}

onintr( )
{
    unlink(tempfile);
    exit(1);
}
```

Why the test and the double call to signal()? Recall that signals like interrupt are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by &), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the onintr() routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that `signal()` returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command processing loop. Think of a text editor: interrupting a long display should not terminate the edit session and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;

main( )
{
     int (*istat)( ), onintr( );

     istat = signal(SIGINT, SIG_IGN);  /* original status */
     setjmp(sjbuf);    /* save current stack position */
     if (istat != SIG_IGN)
          signal(SIGINT, onintr);

     /* main processing loop */
}

onintr( )
{
     printf("\nInterrupt\n");
     longjmp(sjbuf);    /* return to saved state */
}
```

The include file *setjmp.h* declares the type `jmp_buf` — an object in which the state can be saved. `sjbuf` is such an object. The `setjmp()` routine then saves the state of things. When an interrupt occurs the `onintr()` routine is called, which can display a message, set flags, or whatever. `longjmp()` takes as argument an object set by `setjmp()`, and restores control to the location following the call to `setjmp()`, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called when a signal occurs sets a flag and then returns instead of calling `exit()` or `longjmp()`, execution continues at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that 'execution resumes at the exact point it was interrupted,' the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for 'errors' which are caused by interrupted system calls. The ones to watch out for are reads from a terminal, `wait()`, and `pause()`. A program whose `onintr()` routine just sets `intflag`, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
    if (intflag)
        /* EOF caused by interrupt */
    else
        /* true end-of-file */
```

A final subtlety to keep in mind becomes important when catching signals is combined with executing other programs. Suppose a program catches interrupts, and also includes a method (like '!' in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork( ) == 0)
    execl(...);
signal(SIGINT, SIG_IGN);     /* ignore interrupts */
wait(&status);           /* until the child is done */
signal(SIGINT, onintr);      /* restore interrupts */
```

Why is this? Again, it's not obvious, but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function shownsystem() as

```
#include <signal.h>

system(s)   /* run command string s */
char *s;
{
    int status, pid, w;
    register int (*istat)( ), (*qstat)( );

    if ((pid = fork( )) == 0) {
        execl("/bin/sh", "sh", "-c", s, 0);
        _exit(127);
    }
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return(status);
}
```

As an aside on declarations, the function `signal()` obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values `SIG_IGN` and `SIG_DFL` have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the Sun system — the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
#define SIG_DFL     (int (*)())0
#define SIG_IGN     (int (*)())1
```

2.7. The Standard I/O Library

The standard I/O library was designed with the following goals in mind:

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it, no matter how critical the application.

2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.

3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines non-Sun running a version of SunOS.

General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore _ to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

`stdin` the name of the standard input stream

`stdout` the name of the standard output stream

`stderr` the name of the standard error stream

`EOF` is actually −1, and is the value returned by the read routines on end-of-file or error

`NULL` is a notation for the null pointer, returned by pointer-valued functions to indicate an error

`FILE` expands to `struct _iob` and is a useful shorthand when declaring pointers to streams

`BUFSIZ` is a number (viz. 1024) of the size suitable for an I/O buffer supplied by the user. See `setbuf()`, below

`getc(), getchar, putc(), putchar, feof, ferror, fileno` are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names `stdin()`, `stdout()`, and `stderr()` are constants and may not be assigned to.

Standard I/O Library Calls

```
FILE *fopen(filename, type)
     char *filename;
     char *type;
```

opens the file and, if needed, allocates a buffer for it. `filename` is a character string specifying the name. `type` is a character string (not a single character). It may be `"r"`, `"w"`, or `"a"` to indicate intent to read, write, or append. In addition, each *mode* may be followed by a + sign to open the file for reading and writing. `r+` positions the stream at the beginning of the file, `w+` creates or truncates the file, and `a+` positions the stream to the end of the file. Both reads and writes may be used on read/write streams, with the limitation that an `fseek()`, `rewind()`, or reading end-of-file must be used between a read and a write or vice versa. The value returned is a file pointer. If it is `NULL` the attempt to open failed.

freopen()

```
FILE *freopen(filename, type, ioptr)
    char *filename;
    char *type;
    FILE *ioptr;
```

The stream named by ioptr is closed, if necessary, and then reopened as if by fopen(). If the attempt to open fails, NULL is returned, otherwise ioptr is returned, which now refers to the new file. Often the reopened stream is stdin or stdout. The filename and type parameters are as for fopen().

getc()

```
int getc(ioptr)
    FILE *ioptr;
```

returns the next character from the stream named by ioptr, which is a pointer to a file such as returned by fopen(), or the name stdin. The integer EOF is returned on end-of-file or when an error occurs. The null character \0 is a legal character.

fgetc()

```
int fgetc(ioptr)
    FILE *ioptr;
```

acts like getc() but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

putc()

```
int putc(c, ioptr)
    int  c;
    FILE *ioptr;
```

putc() writes the character c on the output stream named by ioptr, which is a value returned from fopen() or perhaps stdout or stderr. The character is returned as value, and EOF is returned on error.

fputc()

```
int fputc(c, ioptr)
    int  c;
    FILE *ioptr;
```

acts like putc() but is a genuine function, not a macro.

fclose()

```
int fclose(ioptr)
    FILE *ioptr;
```

The file corresponding to ioptr is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. fclose() is automatic on normal termination of the program.

fflush()

```
int fflush(ioptr)
    FILE *ioptr;
```

Any buffered information on the (output) stream named by ioptr is written out. Output files are normally buffered if they are not directed to the terminal.

exit()

```
(void) exit(errcode);
    int errcode;
```

terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls fflush() for each output file. To terminate without flushing, use _exit().

feof()

```
int feof(ioptr)
    FILE *ioptr;
```

returns nonzero when end-of-file has occurred on the specified input stream.

ferror()

```
int ferror(ioptr)
    FILE *ioptr;
```

returns nonzero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

getchar()

```
int getchar();
```

is identical to getc(stdin).

putchar()

```
int putchar(c);
```

is identical to putc(c, stdout).

fgets()

```
char *fgets(s, n, ioptr)
    char *s;
    int  n;
    FILE *ioptr;
```

reads to n−1 characters, or up to a newline character, whichever comes first, from the stream ioptr into the string pointed to by the character pointer s. A null character is placed after the last character read in the strings s. fgets returns the first argument, or NULL if error or end-of-file occurred.

puts()

```
int puts(s)
    char *s;
```

puts() copies the null-terminated strings specified by s onto the standard output stream and appends a newline character.

fputs()

```
int fputs(s, ioptr)
    char *s;
    FILE *ioptr;
```

writes the null-terminated string (character array) s on the stream ioptr. No newline is appended. The last character transmitted is returned as value, or EOF is returned on error.

ungetc()

```
int ungetc(c, ioptr)
    int  c;
    FILE *ioptr;
```

The argument character c is pushed back on the input stream named by ioptr. Only one character may be pushed back.

printf()

```
int printf(format, a1, ...)
    char *format;

int fprintf(ioptr, format, a1, ...)
    FILE *ioptr;
    char *format;

int sprintf(s, format, a1, ...)
    char *s;
    char *format;
```

printf() writes on the standard output. fprintf() writes on the output stream named by ioptr. sprintf() puts characters in the character array (string) named by s. The specifications are as described in printf(3S).

printf() and fprintf() return the number of characters actually transmitted, or return EOF if any error condition exists on the output file. sprintf() returns a pointer to the buffer where the formatted string is placed.

scanf()

```
int scanf(format, a1, ...)
    char *format;

int fscanf(ioptr, format, a1, ...)
    FILE *ioptr;
    char *format;

int sscanf(s, format, a1, ...)
    char *s;
    char *format;
```

scanf() reads from the standard input. fscanf() reads from the named input stream. sscanf() reads from the character string supplied as s. scanf() reads characters, interprets them according to the format, and stores the results in its arguments. Each routine expects as arguments a control string format, and a set of arguments, *each of which must be a pointer*, indicating where the converted input should be stored.

scanf() returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

fread()

```
int fread(ptr, sizeof(*ptr), nitems, ioptr)
    unsigned nitems;
    FILE *ioptr;
```

reads nitems of data of the type of *ptr from file ioptr into the memory area starting at ptr. No advance notification that binary I/O is being done is required. fread() returns the number of items actually read from the specified stream.

fwrite()

```
int fwrite(ptr, sizeof(*ptr), nitems, ioptr)
    unsigned nitems;
    FILE *ioptr;
```

Like fread(), but in the other direction. fwrite returns the number of items actually transmitted to the specified stream. This may possibly be less than the number of items requested if an error occurs while the transfer is in process.

rewind()

```
(void) rewind(ioptr)
    FILE *ioptr;
```

rewinds the stream named by ioptr. It is not very useful except on input, since a rewound output file is still open only for output.

system()

```
int system(string)
    char *string;
```

The string is executed by the shell as if typed at the terminal. The return value is the exit code of the invoked shell, which is usually the exit code of the last command executed by it.

getw()

```
int getw(ioptr)
    FILE *ioptr;
```

returns the next word from the input stream named by ioptr. EOF is returned on end-of-file or error, but since this a perfectly good integer, feof() and ferror() should be used. A ‘word’ is 32 bits on the Sun Workstation.

putw()

```
int putw(w, ioptr)
    FILE *ioptr;
```

writes the integer w on the named output stream. putw() returns the current error status of the specified stream, as if an ferror() call had been made.

setbuf()

```
(void) setbuf(ioptr, buf)
    FILE *ioptr; char *buf;
```

setbuf() may be used after a stream has been opened but before I/O has started. If buf is NULL, the stream is unbuffered. Otherwise the buffer supplied is used. It must be a character array of sufficient size:

```
char    buf[BUFSIZ];
```

`setbuffer()`

```
(void) setbuffer(ioptr, buf, size)
    FILE *ioptr;
    char *buf;
    int  size;
```

`setbuffer()` is like `setbuf()` (described above), but can be used when a specified, nonstandard buffer size should be used.

`fileno()`

```
int fileno(ioptr)
    FILE *ioptr;
```

returns the integer file descriptor associated with the file.

`fseek()`

```
int fseek(ioptr, offset, ptrname)
    FILE *ioptr;
    long offset;
    int ptrname;
```

The location of the next byte in the stream named by `ioptr` is adjusted. `offset` is a long integer. If `ptrname` is 0, the offset is measured from the beginning of the file; if `ptrname` is 1, the offset is measured from the current read or write pointer; if `ptrname` is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. When this routine is used on non SunOS systems, the offset must be a value returned from `ftell()` and the `ptrname` must be 0.

`ftell()`

```
long ftell(ioptr)
    FILE *ioptr;
```

The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. On non SunOS systems the value of this call is useful only for handing to `fseek()`, so as to position the file to the same place it was when `ftell()` was called.

`getpw()`

```
int getpw(uid, buf)
    int uid;
    char *buf;
```

The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array `buf`, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

`malloc()`

```
char *malloc(num)
    int  num;
```

allocates `num` bytes. The pointer returned is aligned so as to be usable for any purpose. `NULL` is returned if no space is available.

`free()`

```
int free(ptr)
    char  *ptr;
```

`free()` frees up memory previously allocated by `malloc()`. `free()` returns a 0 if any errors were detected (such as `ptr` being misaligned), and returns 1 otherwise. Disorder can be expected if the pointer was not obtained from `malloc()`.

`calloc()`

```
char *calloc(num, size);
     unsigned num;
     unsigned size;
```

allocates space for `num` items, each of size `size`. The space is guaranteed to be set to 0 and the pointer is aligned so as to be usable for any purpose. `NULL` is returned if no space is available.

`cfree()`

```
(void) cfree(ptr, num, size)
     char *ptr;
     unsigned num;
     unsigned size;
```

Space is returned to the pool used by `calloc()`. Disorder can be expected if the pointer was not obtained from `calloc()`.

The following are macros whose definitions may be obtained by including `<ctype.h>`.

Character Type Checking

`isalpha(c)` returns nonzero if *c* is alphabetic.

`isupper(c)` returns nonzero if *c* is upper-case alphabetic.

`islower(c)` returns nonzero if *c* is lower-case alphabetic.

`isdigit(c)` returns nonzero if *c* is a digit.

`isxdigit(c)` returns nonzero if *c* is a hexadecimal digit — that is, one of '0' through '9', 'a' through 'f', or 'A' through 'F'.

`isspace(c)` returns nonzero if *c* is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

`ispunct(c)` returns nonzero if *c* is any punctuation character, that is, not a space, letter, digit or control character.

`isalnum(c)` returns nonzero if *c* is a letter or a digit.

`isprint(c)` returns nonzero if *c* is printable — a letter, digit, space, or punctuation character.

`iscntrl(c)` returns nonzero if *c* is a control character.

`isascii(c)` returns nonzero if *c* is an ASCII character, that is, less than octal 0200.

`isgraph(c)` returns nonzero if *c* is a printing character — like `isprint(c)` but doesn't include the space character.

Character Type Conversion

`toupper(c)` returns the upper-case character corresponding to the lower-case letter *c*.

`tolower(c)` returns the lower-case character corresponding to the upper-case letter *c*.

3

System V Compatibility Features

3

System V Compatibility Features

This overview is intended for both users and programmers who want to learn about System V compatibility features in SunOS 4.0.

3.1. Introduction

SunOS 4.0 offers Sun users nearly complete System V compatibility. Sun's compatibility package allows programmers to write software that meets the Base Level of the System V Interface Definition (SVID). SunOS 4.0 represents yet another phase of joint efforts by AT&T and Sun to unify the different versions of the UNIX system. The two principal versions have been 4.2 BSD (now 4.3 BSD),† and System V in its various releases.

System V and 4.3 BSD are not radically different, either in the interface they present to the user, or the routines they provide for the programmer. They are derived from UNIX systems written by Ken Thompson and Dennis Ritchie in the mid-seventies, and many features are essentially unchanged since then.

The System V compatibility package permits programmers to write and test software targeted for either System V or 4.x BSD. Users who acquire software that runs only on System V can run it by means of the compatibility library. Commands, system calls, and library routines can be drawn concurrently from either the Berkeley or System V set. It is even possible to have one window that uses BSD by preference, and another window that uses System V by preference.

System V Enhancements in SunOS 4.0

SunOS 4.0 incorporates the full SVID Release 3 Base Level system, which reflects further progress on System V and BSD convergence. However, SunOS 4.0 does not support mandatory record and file locking. New features include:

□ System calls compatible with SVID Base Level system calls, including: `chown()`, `creat()`, `fcntl()`, `kill()`, `mknod()`, `open()`, and `utime()`.

□ Complete System V STREAMS interface, to support portable communication protocol modules, and to simplify the writing of device drivers.

□ Fully System V and BSD compatible `tty`(4) interface using STREAMS, and supporting all character sizes and parity settings.

† An outgrowth of research at U.C. Berkeley, BSD stands for Berkeley Software Distribution.

- System V compatible archive utility ar(1V).
- System V batch utilities and job scheduling facilities: at(1), batch(1), cron(1), and crontab(1).
- Access to Sun's value-added libraries (SunView for example) from inside System V programs.

A Brief History

In early 1985, AT&T released the *System V Interface Definition* (SVID). This was a major step because it made explicit exactly what was standard about System V, and by omission, what was not. In late 1985, Sun and AT&T agreed to work together to converge the two major strands of UNIX into a single system. In late 1986, Sun's Release 3.2 combined System V with 4.2 BSD, including almost full Base Level compatibility. Now in early 1988, SunOS 4.0 offers full Base Level compatibility, plus compatibility with additional SVID features.

How the Compatibility Tools Work

System V programs that are upwards compatible with those in 4.x BSD have already been added to the regular system directories. For example, /usr/bin/sh is the new Bourne shell, and /usr/bin/make is backward compatible System V enhancements.

Programs that existed only on System V have been added to a regular system directories as well. For example, the text manipulation programs cut(1) and paste(1) both reside in /usr/bin.

System V programs that are incompatible with those in 4.x BSD reside in the directory /usr/5bin. For example, /usr/5bin/stty has an entirely different set of options from /usr/bin/stty. If you want to use System V programs by preference, simply include /usr/5bin early in your path, as in these lines from the .login or .profile files:

```
(csh) set path = (/usr/5bin /usr/bin /usr/ucb .)

(sh)  PATH=/usr/5bin:/usr/bin:/usr/ucb::
      export PATH
```

The directories /usr/5bin, /usr/5lib, and /usr/5include contain material that has not yet been converged. Libraries and include files for compiling System V software reside in /usr/5lib and /usr/5include respectively. If you want to compile a program written for System V, don't use /usr/bin/cc but rather /usr/5bin/cc, which will read all the correct include files and load the correct libraries. You may want to make an alias or shell function that invokes the System V compiler, to obviate the need for changing your PATH:

```
(csh) alias cc5 /usr/5bin/cc

(sh)  cc5() {
          /usr/5bin/cc $*
      }
```

The directories that constitute the System V compatibility package are optional, requiring several megabytes of disk space. The suninstall(8) program lets you decide whether or not to load these directories.

The Group Mechanism

Sun's Release 3.2 used the group mechanism from BSD rather than from System V. SunOS 4.0, by contrast, allows both group mechanisms to work together. When the GIDset- bit is set on a directory, a file created in that directory will be assigned the directory's GID (BSD semantics). Otherwise, it will be assigned the effective GID of the creating process (System V semantics).

In either case, the GIDset- bit will be set when mkdir(2) creates new directories. Users will be able to set up their login directories to follow the semantics they prefer. SunOS 4.0 distribution tapes are shipped with the GIDset- bit set on all directories, thereby giving BSD semantics as the default. When you install SunOS 4.0, if you want to mount old filesystems and have them act as they did in the past, type the following command line for each mounted file system:

```
# find mounted.directory -type d -exec chmod g+s "{}"
```

To set System V semantics on some portion of the installed system, use g-s instead of g+s in the above command line. There is a mount option called grpid that always provides BSD semantics. This option may be needed when a SunOS 4.0 client mounts a file system from a server that has not yet been upgraded to SunOS 4.0.

Compatibility of System Calls

For security reasons, the system call chown() requires root privilege. On System V, by contrast, the owner of a file may change its ownership. This would make the quota mechanism completely unenforceable.

The system call utime() now allows file time stamps to be set by any process with write permission on the file.

The system call kill() may now send signals to any process with an effective or real UID that matches the effective or real UID of the sender. As before, root processes may send signals to any process.

The system call mknod() may now be used to create directories, although the system call mkdir() is preferred.

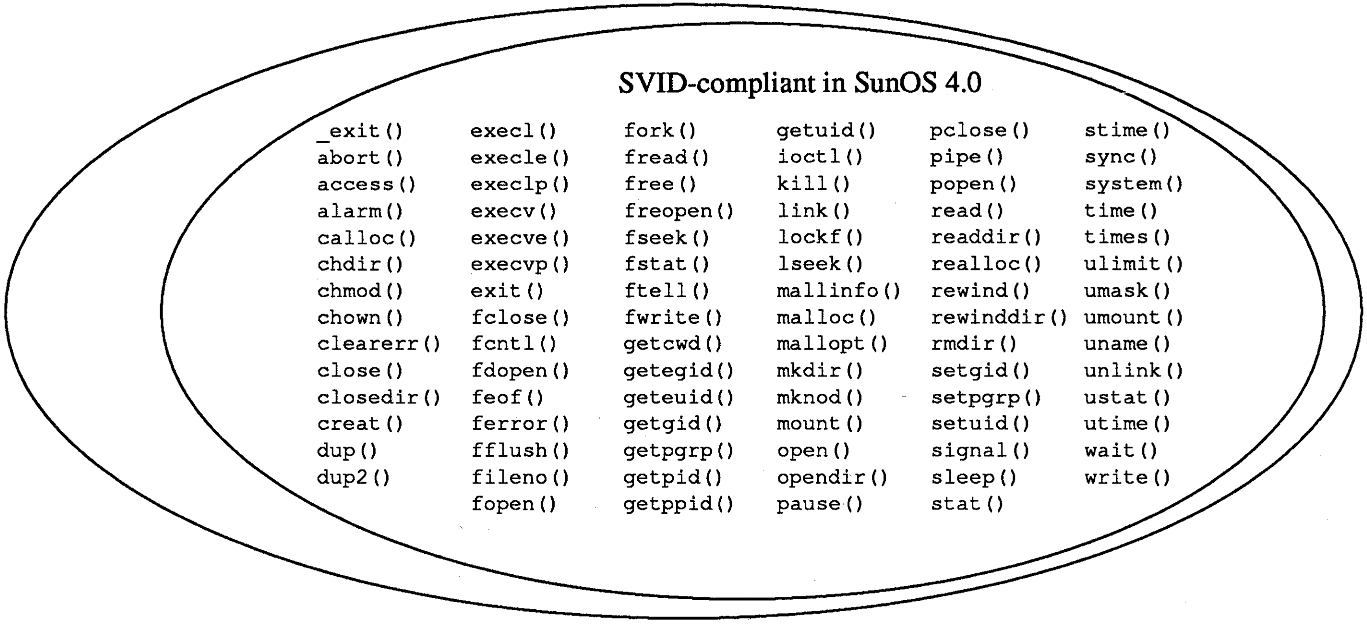
With the system call fcntl(), the flags F_SETFL and O_NDELAY differ between the include files in /usr/include and /usr/5include. In either case they should do the right thing. For BSD, they affect all references to the underlying file. For System V, they affect only file descriptors associated with the same file table entry.

The terminal driver now supports 5-bit and 6-bit characters, and arbitrary settings of VMIN and VTIME. However, the default erase and kill characters are not # and @ but rather Delete and Control-U.

3.2. SVID Compliance in SunOS 4.0

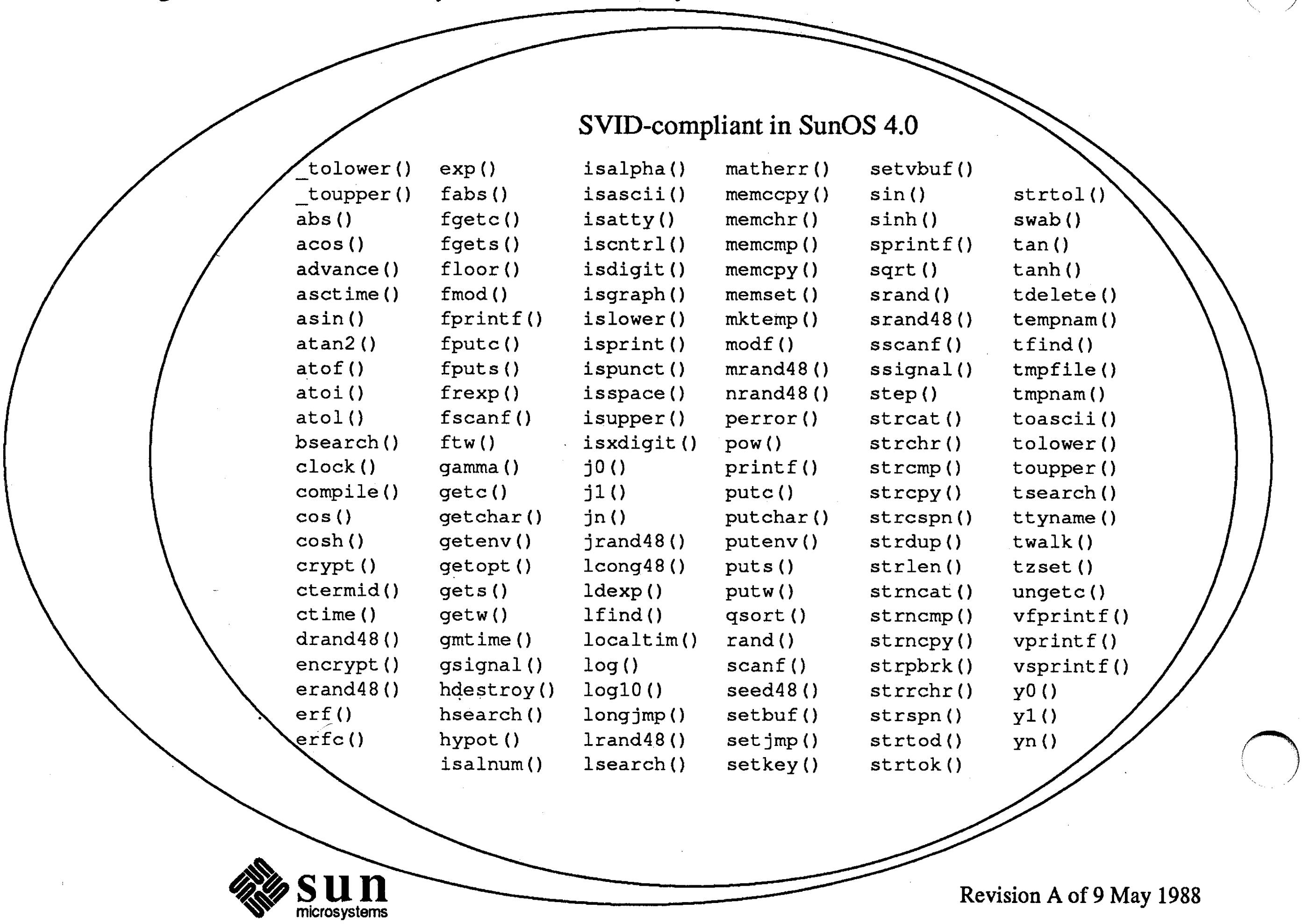
The Venn diagrams in this section demonstrate how SunOS 4.0 complies with release 3 of the System V Interface Definition SVID).(

Figure 3-1 *SVID Base System OS Service Routines*



```
_exit()
abort()
access()
alarm()
calloc()
chdir()
chmod()
chown()
clearerr()
close()
closedir()
creat()
dup()
dup2()
execl()
execle()
execlp()
execv()
execve()
execvp()
exit()
fclose()
fcntl()
fdopen()
feof()
ferror()
fflush()
fileno()
fopen()
fork()
fread()
free()
freopen()
fseek()
fstat()
ftell()
fwrite()
getcwd()
getegid()
geteuid()
getgid()
getpgrp()
getpid()
getppid()
getuid()
ioctl()
kill()
link()
lockf()
lseek()
mallinfo()
malloc()
mallopt()
mkdir()
mknod()
mount()
open()
opendir()
pause()
pclose()
pipe()
popen()
read()
readdir()
realloc()
rewind()
rewinddir()
rmdir()
setgid()
setpgrp()
setuid()
signal()
sleep()
stat()
stime()
sync()
system()
time()
times()
ulimit()
umask()
umount()
uname()
unlink()
ustat()
utime()
wait()
write()
```

Figure 3-2 *SVID Base System General Library Routines*



```
_tolower()
_toupper()
abs()
acos()
advance()
asctime()
asin()
atan2()
atof()
atoi()
atol()
bsearch()
clock()
compile()
cos()
cosh()
crypt()
ctermid()
ctime()
drand48()
encrypt()
erand48()
erf()
erfc()
exp()
fabs()
fgetc()
fgets()
floor()
fmod()
fprintf()
fputc()
fputs()
frexp()
fscanf()
ftw()
gamma()
getc()
getchar()
getenv()
getopt()
gets()
getw()
gmtime()
gsignal()
hdestroy()
hsearch()
hypot()
isalnum()
isalpha()
isascii()
isatty()
iscntrl()
isdigit()
isgraph()
islower()
isprint()
ispunct()
isspace()
isupper()
isxdigit()
j0()
j1()
jn()
jrand48()
lcong48()
ldexp()
lfind()
localtim()
log()
log10()
longjmp()
lrand48()
lsearch()
matherr()
memccpy()
memchr()
memcmp()
memcpy()
memset()
mktemp()
modf()
mrand48()
nrand48()
perror()
pow()
printf()
putc()
putchar()
putenv()
puts()
putw()
qsort()
rand()
scanf()
seed48()
setbuf()
setjmp()
setkey()
setvbuf()
sin()
sinh()
sprintf()
sqrt()
srand()
srand48()
sscanf()
ssignal()
step()
strcat()
strchr()
strcmp()
strcpy()
strcspn()
strdup()
strlen()
strncat()
strncmp()
strncpy()
strpbrk()
strrchr()
strspn()
strtod()
strtok()
strtol()
swab()
tan()
tanh()
tdelete()
tempnam()
tfind()
tmpfile()
tmpnam()
toascii()
tolower()
toupper()
tsearch()
ttyname()
twalk()
tzset()
ungetc()
vfprintf()
vprintf()
vsprintf()
y0()
y1()
yn()
```

Figure 3-3 *SVID Kernel Extension OS Service Routines*

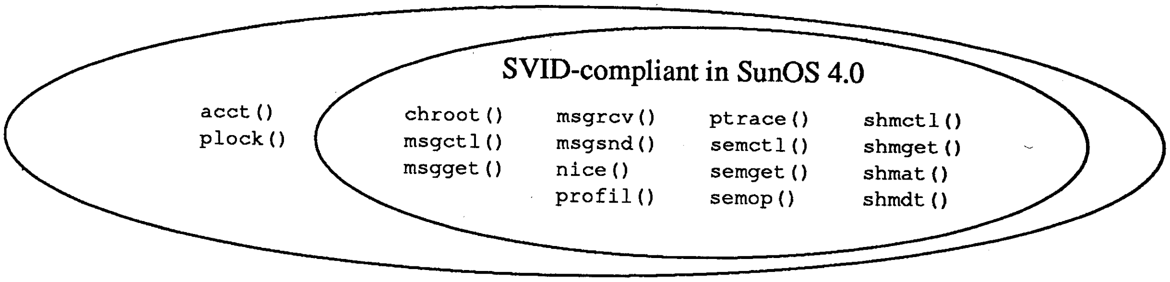


Figure 3-4 *SVID Basic Utilities Extension*

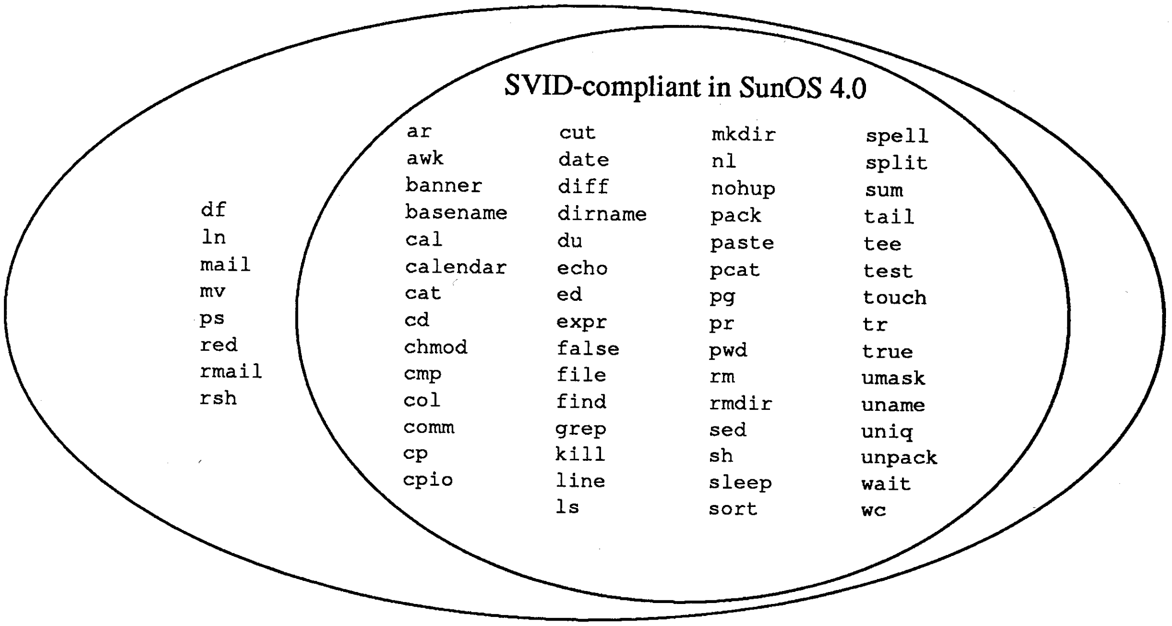


Figure 3-5 *SVID Advanced Utilities Extension*

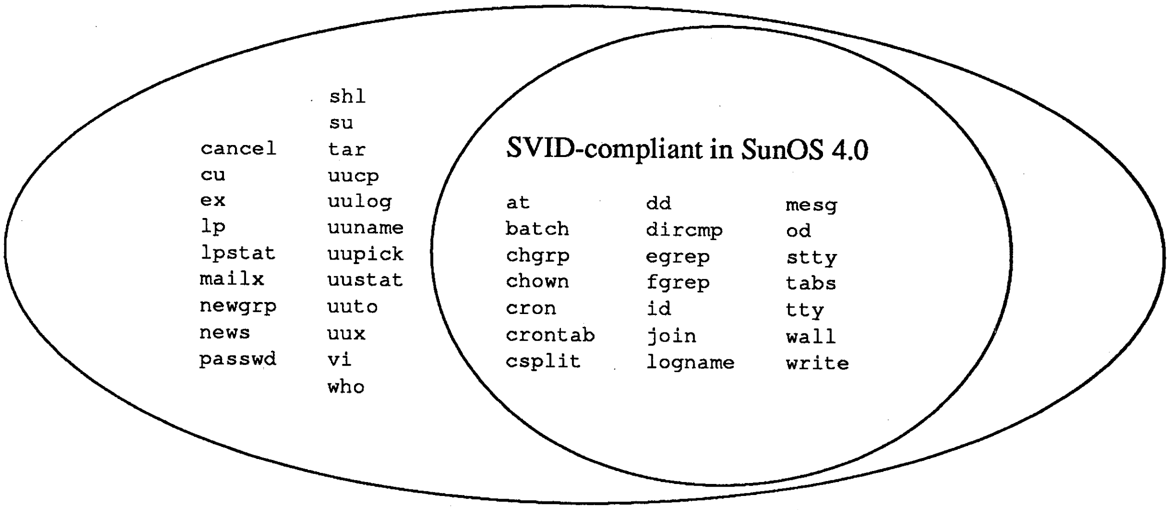


Figure 3-6 *SVID Administered Systems Extension Utilities*

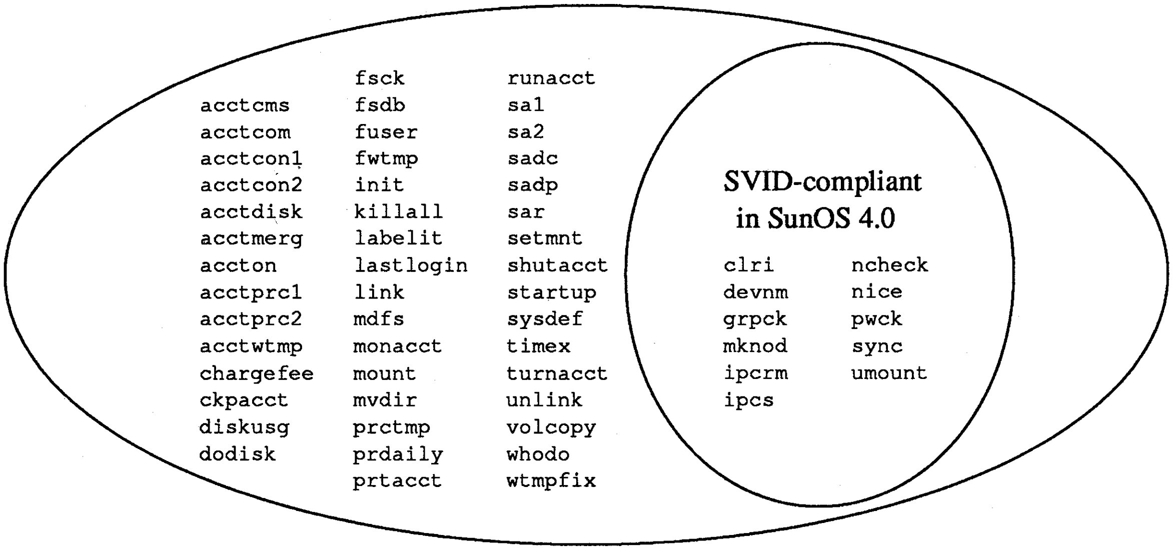


Figure 3-7 *SVID Software Development Extension Utilities*

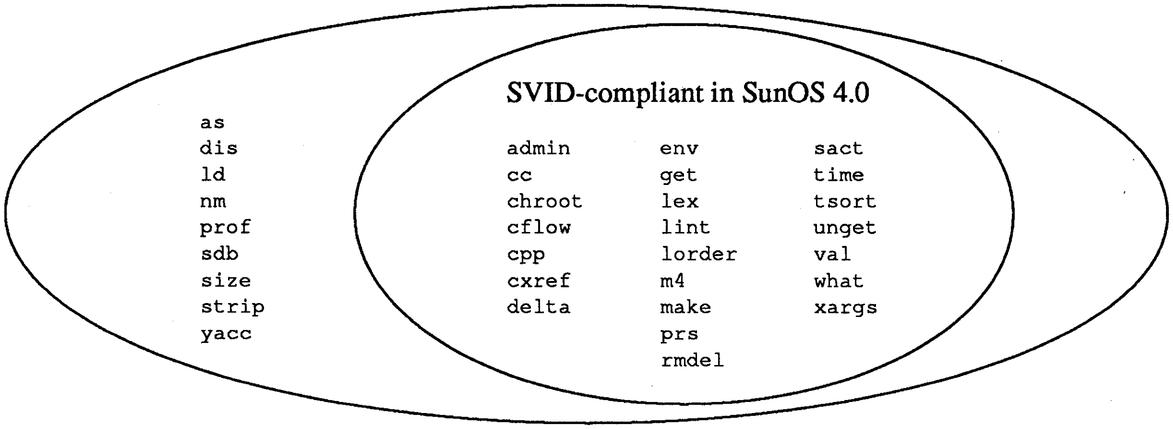


Figure 3-8 *SVID Software Development Extension Additional Routines*

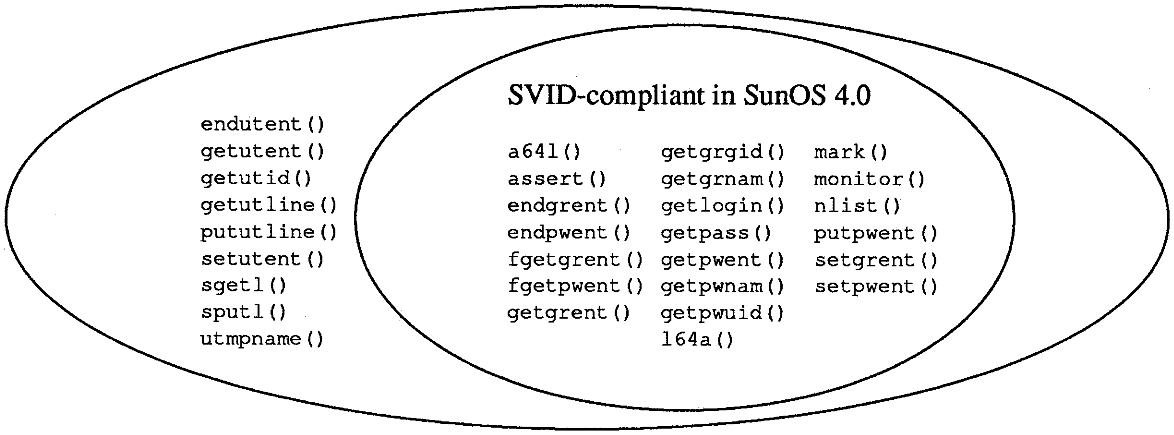


Figure 3-9 *SVID Terminal Interface Extension Utilities*

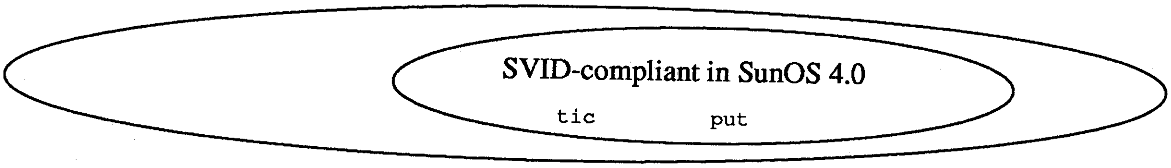


Figure 3-10 *SVID Terminal Interface Extension Library Routines*

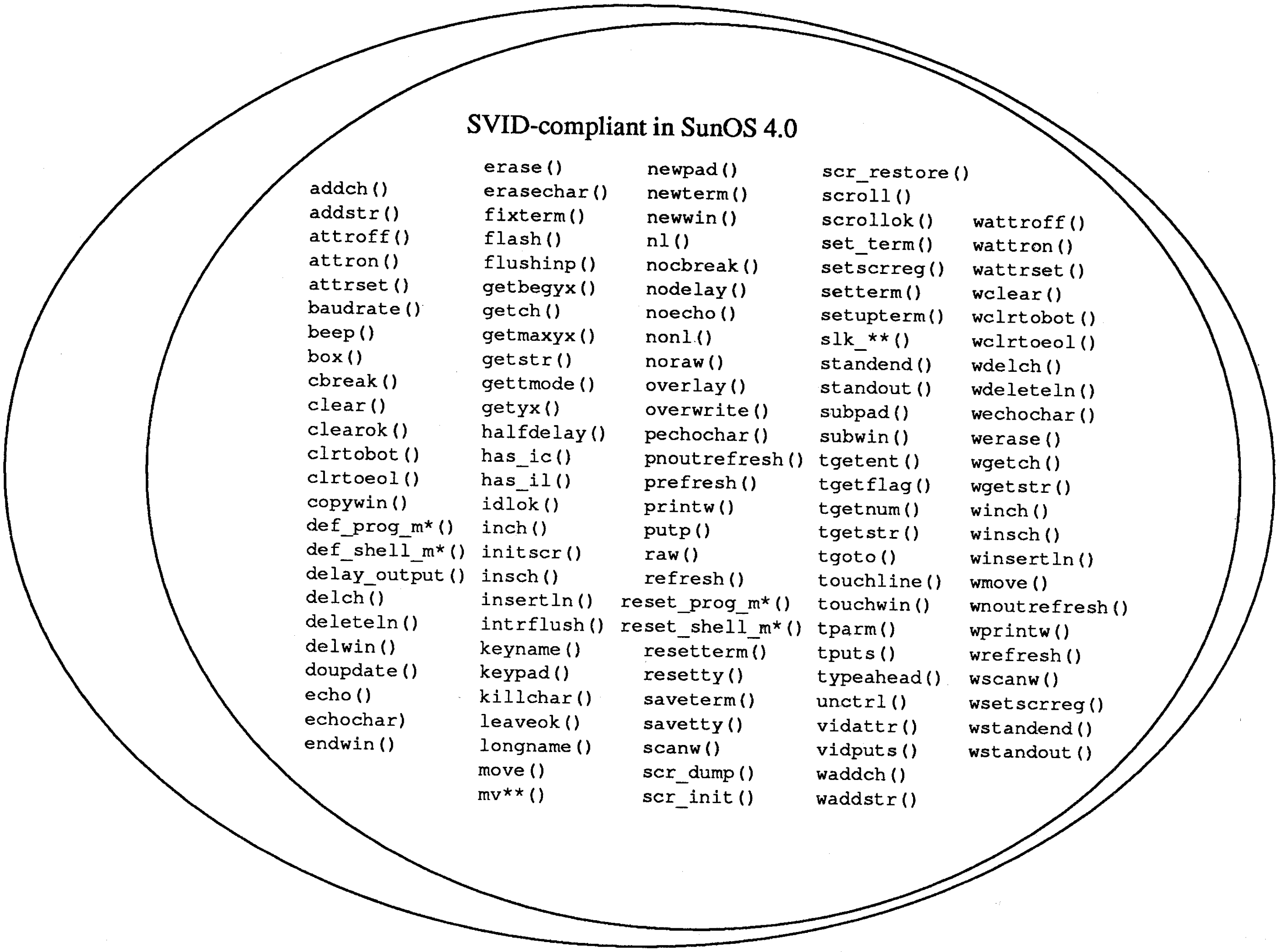


Figure 3-11 *SVID Open Systems Networking Interfaces Library Routines*

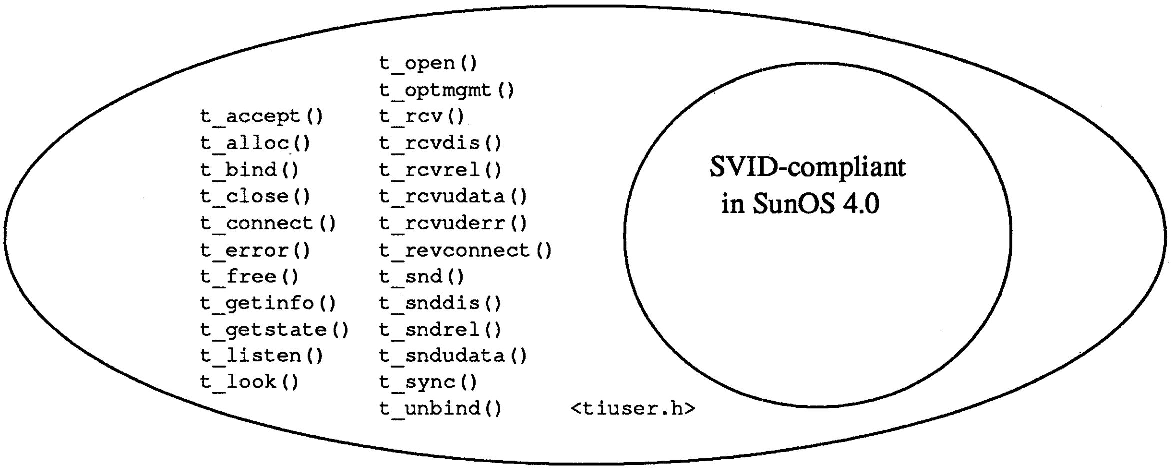


Figure 3-12 *SVID STREAMS I/O Interface Operating System Service Routines*

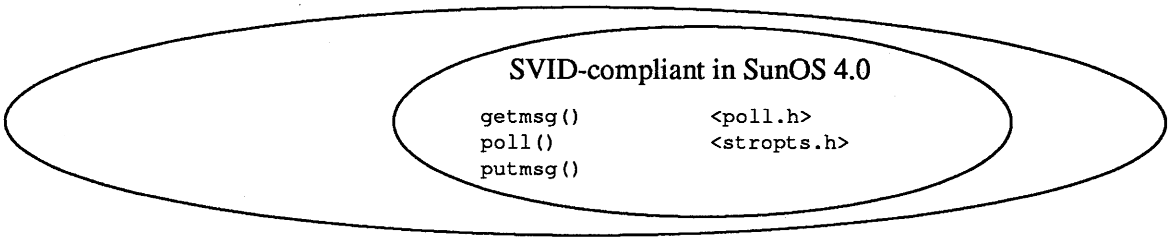
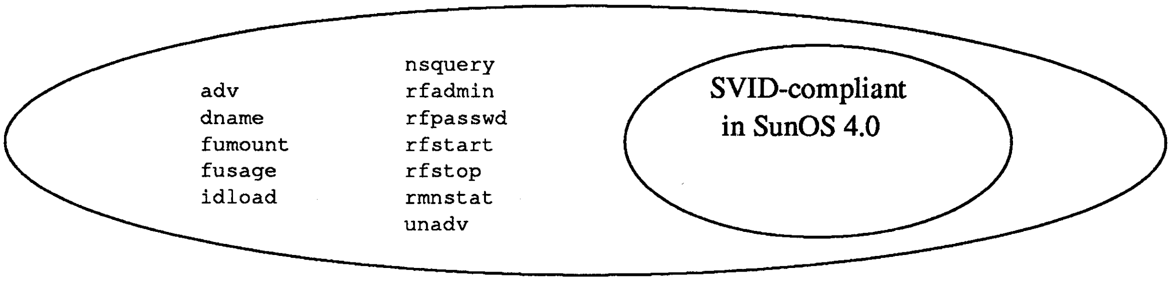


Figure 3-13 *SVID Shared Resource Environment Utilities*



4

Shared Libraries

4

Shared Libraries

Operating systems like SunOS have long achieved more efficient use of memory by sharing a single physical copy of a program's *text* (code) among the processes executing it. But while the text of a program may be shared among its concurrent invocations, a significant portion of that text, consisting of library routines, may be duplicated as part of *other* running programs. For example, widely-used library functions such as `printf()` may be replicated any number times throughout memory, and again in various executables throughout the file system. This suggests that still-greater efficiencies can be had by sharing text at the *library* level whenever possible.

The SunOS shared library mechanism improves resource utilization in a way that is both straightforward and flexible:

□ No specialized kernel support is required; it uses the standard memory-mapping and copy-on-write features provided by the `mmap`(2) system call and the kernel memory management facilities.

□ It is designed to minimize the burdens placed on users of existing code. In particular:

  - Shared libraries are transparent to the programs that use them, as well as the build procedures for those programs.
  - They are largely transparent to standard system utilities, including debuggers.
  - Shared libraries are transparent to library source code written in C. However, some special procedures are necessary when building the shared libraries themselves.
  - The allocation of address space for shared library routines is handled automatically.
  - Unlike statically-linked executables, programs that rely on shared libraries need not be rebuilt if an underlying library changes (so long as that library's calling interface remains compatible).
  - The use of shared libraries is not required. You can specify the static version of a SunOS shared library as desired.

In addition, shared libraries enhance the development environment by making it easier to modify and test compatible updates to library functions.

4.1. Definitions

Shared Object

A *shared object*, or `.so` file, is an `a.out(5)` format file produced by `ld(1)`. A shared object differs from a runnable program in that it lacks an initial entry point.

Shared Library

A ''shared library'' is a shared object file that is used as a library. In cases where the shared library exports initialized data, the shared object (`.so`) may be paired with an optional ''data interface description'' (`.sa`) file. (See *Building a Shared Library*, below, for details.)

Static vs. Dynamic Link Editing

Link editing is the set of operations necessary to build an executable program from one or more object files. *Static linking* indicates that the results of these operations are saved to a file. *Dynamic linking* refers to these same link-edit operations when performed at run-time; the executable that results from dynamic linking appears in the running process, but is not saved to a file.

Position Independent Code (PIC)

Position-Independent code (PIC) requires link editing only to relocate references to objects that are external to the current object module. Position-independent code is readily shared.

Static and Dynamic Link Editors

The link-editing facilities of `ld` have been made available for use at run-time as well as at compile-time. At compile time, the static link editor, `ld`, can build an executable file in which some symbols remain unresolved. An executable (`a.out`) file that contains unresolved symbols is said to be *incomplete*. Incomplete executables require dynamic link editing at run-time.

The dynamic link editor, `/usr/lib/ld.so`, uses the system's memory management facilities to map in and bind the shared object files that are required at run-time, and performs the link editing operations that were deferred by `ld`. As long as the text bound-in at run-time is not subsequently modified (say, by a link-edit operation or an update to initialized external data), it remains shared among the various (disparate) programs that use it. However, if the text of a shared routine should need to be modified by a process during the course of execution, local (exclusive) copies of the affected pages are created and maintained.

4.2. Using Shared Libraries

For the application developer, the decision to use shared libraries is made at the static linking phase, when running `ld`. By default, if a shared version of a library is available, `ld` constructs an executable that uses the shared version.

Building a Program to Use Shared Libraries

`ld` combines a variety of object files to produce an executable (`a.out`) file. Exactly what code gets produced, and how complete the `a.out` is, depends on the command-line options and input files supplied as arguments on the command line. `ld` simply defers the resolution of any symbols that remain after it has run out of definitions, and assumes that the program will be fully linked by `ld.so` at run-time. `ld` accepts as input:

□ Simple object files. `ld` simply concatenates (and links) `.o` files in the order that they are encountered.

□ ar(1) libraries. Each .a file is searched exactly once as it is encountered, and only those definitions that match an unresolved external symbol are extracted, concatenated to the text (or data), and linked.

□ Shared objects. Any .so encountered is searched for symbol definitions and references, but does not normally contribute to the concatenated text (see *Binding of PIC with non-PIC*, for exceptions having to do with ld's -dc option). However, the occurrence of each shared object is noted in the resulting a.out file; this information is used by ld.so to perform dynamic link editing at run-time.

ld's output can be one of two basic types:

□ An ''executable'' (a.out) file. This file is either a *program*, if it has an entry point, or a *shared object* (.so), if it does not.

□ Another ''simple object'' (.o) file. When given the -r flag, ld combines the input object files to form a single, larger one. (This is a special use for ld which is of little relevance to shared libraries.)

You can indicate which libraries are to be used by supplying a -l*name* option on the ld command line for each. ld searches each library in the order specified. The *name* string is an abbreviated version of the library's filename; the full name is of the form 'lib*name*.a' if in archive format, or 'lib*name*.so.*version*' if it is in shared object form. (see *Version Control* below, for a detailed discussion of the *version* suffix). At ld-time, this version information is noted; it must be matched properly for successful binding at run-time by ld.so.

The location of the library specified by a -l option is determined by an ordered list of directories in which to search called the library search path. This search path is specified as follows. First, the value of the LD_LIBRARY_PATH environment variable (a colon-separated list of pathnames). Then, any and all directories specified with -L*directory* options. And finally, the (default) directories /usr/lib and /usr/local/lib.

Each directory supplied with -L is recorded for use when the program is executed, as are the default directories. Directory search information obtained from LD_LIBRARY_PATH is *not* recorded in this manner. However, the search path that LD_LIBRARY_PATH contains at run-time *is* searched at that time; this allows an alternate set of libraries to be used.

At ld-time, the library search is satisfied by the first occurrence of either form of the library (.so or .a if no .so is found), but if both versions are found in the *same* directory, the .so form is used by default. However, the choice of whether a .so or .a version is used by ld can be controlled by the binding mode options described in the next section.

Binding Mode Options

-Bstatic and -Bdynamic

You can specify the binding mode by supplying one of the -B*keyword* options on the command line:

-Bdynamic Allow dynamic binding, do not resolve symbolic references, and allow creation of execution-time symbol and relocation information. This is the default setting. Note that ld records the name of the .so file with the highest version number in the executable.

-Bstatic Force static binding, this mode is also implied by options that generate non-sharable executable formats.

-Bdynamic and -Bstatic may both be specified a number of times to toggle the binding mode for specific libraries. Like -l, their influence is dependent upon their location in the command line. Libraries that appear after a -Bstatic are linked statically. Libraries that appear after a -Bdynamic are treated as shared (when a shared version is available).

NOTE *Since* -Bdynamic *is the default setting, the use of shared libraries in the construction of a program thus "falls out" from installing the* .so *in* ld*'s library search path.*

If -Bstatic is in effect, ld refuses to use the .so form of a library; it continues searching for an equivalent library with the .a suffix, and an explicit request to load a .so file is treated as an error.

The following example shows how -Bstatic and -Bdynamic can be used to use selected shared and static libraries. This cc command:

```
cc -o test test.c -lsuntool -Bstatic -lsunwindow -Bdynamic -lsunwindow -lpixrect
```

generates the ld command:

```
  /bin/ld -dc -dp -e start -X -o test /usr/lib/crt0.o test.o -Bstatic -lsuntool \
      -Bdynamic -lsunwindow -lpixrect -lc
```

Since -Bstatic turns off the use of shared libraries, ld finds the static (.a) suntool library and uses it for link editing immediately. The subsequent -Bdynamic option tells ld to use shared versions of the sunwindow, pixrect and C libraries, if available.

-N and -n Options for ld

The ld options -N and -n instruct ld to build a non-pageable executable. Their use implies a -Bstatic option.

Binding of PIC with Non-PIC

-dc and -dp Options

As noted in the above example, the cc command generates an ld command with the -dp and -dc options. These options are included to facilitate binding of non-PIC code (generated by default) with the PIC shared libraries that a program might use. The bindings of interest are to:

- □ **commons**, (externs): allocated after the program is completely assembled (-dc);
- □ **initialized data**: imported from the shared libraries (-dc); and
- □ **entry points**: supplied by the shared libraries (-dp).

Without special handling, references to these objects would require execution-time link editing, resulting in unsharable code. To improve the degree of sharing for such programs, -dc and -dp force the allocation of commons and the creation of aliases for library entry points, respectively. These allocations and aliases are created as part of the non-PIC executable, and result in programs that are considered to be ''pure-text'' non-PIC programs, even though they may require dynamic link editing.

NOTE *While it is possible to invoke the* ld *command directly, it is generally better practice to rely on the compiler-driver (such as* cc*) to generate the appropriate* ld *command, so as to remain insulated from any future changes in the compilation environment. Compiler commands such as* cc *accept and pass on options to* ld.

Use of Assertions

The -assert Option

To help detect any potential sharability or correctness problems, ld can validate certain assertions about an executable that it builds. This assertion checking is invoked by the ''-assert *keyword*'' option, where *keyword* is one of:

definitions if the resulting program were run now, there would be no run-time undefined symbol diagnostics. This assertion is set by default, and is sufficient for validating applications that make use of shared libraries.

pure-text the resulting executable requires no further relocations to its text. The code of a shared library should be validated using this assertion.

Run-Time Use of Shared Libraries

At run-time, ld.so finishes the job started by ld. That is, it performs the link-editing operations needed to resolve a program's remaining references using shared-library code and data. ld.so's first task is to find and map in the required libraries. It applies the same library search rules as ld, looking first in the directories specified by the *current* value of LD_LIBRARY_PATH, and then in the directories in the search path recorded by ld (the default directories and those specified by -L). In addition, ld.so attempts to find the ''best'' version of a shared library, that is, the version with the highest minor number (as described under *Version Control* below).

SunOS Shared Libraries

The shared libraries provided in SunOS are:

□ The C library (both BSD and System V variants)

□ Window libraries (`suntool` and `sunwindow`)

□ `pixrect`

□ kernel virtual memory access (`kvm`)

Static (`.a`) versions of these libraries are also provided.

Dynamic vs. Static Binding Semantics

There are some semantic differences between dynamic and static binding. These are not expected to cause a problem with programs that avoid questionable practices with regard to library search order. However, there is a potential for problems when programs are built from some components that have become dynamically loadable, while others remain static. Given the case where:

```
hermes% ld -o x ...dc sc
```

The executable `x` is composed of several objects, including a dynamic component, `dc`, and a static component, `sc`. `dc` was, prior to the introduction of shared libraries, an unordered archive file, and both `dc` and `sc` contain definitions for the symbol `getsym`. Suppose that `dc` contains a *reference* to `getsym`. If, in `dc`'s archive version, the definition for `getsym` preceded its reference, `ld` might have resolved that reference using the definition from `sc`. But in `dc`'s current (dynamic) form, its own definition is used instead. This is a result of the fact that at run-time, `ld.so` searches for a symbol definition starting with the main program, and then all `.so`'s in load order. Even though it allows for an inconsistency of this sort, this behavior preserves the ability to *interpose* definitions on library entry points.

Debuggers

The SunOS debuggers have been modified to deal with the dynamic linking environment provided by the new `ld`. In particular, they understand that symbol definitions may appear after a program starts executing. However debugger users must be aware that library symbols will not be resolved until `main()` has been called, as the next example shows.

```
hermes% cc -g -o test test.c
hermes% dbx test
Reading symbolic information...
Read 40 symbols
(dbx) stop in printf
no module, procedure or file named 'printf'
(dbx) stop in main
(1) stop in main
(dbx) run
Running: test
stopped in main at line 4 in file "test.c"
    4           printf("%d 0, errno);
(dbx) stop in printf
(3) stop in printf
(dbx) cont
stopped in printf at 0xed76954
0xed76954:              moveml  #<d7,a5>,sp@
Current function is main
    4           printf("%d 0, errno);
```

Users of debugging tools also need to be aware that core files have incomplete information on the state of shared code. Core files contain only the stack and data regions of a process image. The text, and more importantly, the static data regions of dynamically loaded objects do *not* appear. Thus, modifications made to initialized data are not reflected in the core file.

Performance Issues

Shared libraries represent a classic space vs. time trade-off. The work of incorporating the library code into an address space is deferred in order to save both primary and secondary storage. Therefore, one can expect to pay a slight CPU time penalty with programs that use shared libraries. This penalty can be attributed to added cost of:

- dynamically loading the libraries,
- performing the link editing operations, and
- the execution of the library PIC code.

However, these costs can be offset by the savings in I/O access time when library code is already mapped in by another program, since the (real) I/O time required to bring in a program and begin execution will be greatly reduced. As long as the CPU time required to merge the program and its libraries does not exceed the I/O time saved, the apparent performance of the program will be the same or better. However, if sharing does *not* occur, or if the system's CPU is already saturated, such savings may not be achieved.

Dependencies on Other Files

A dynamically bound program consists not only of the executable file that is the output of `ld`, but also of the files referred to during execution. Moving a dynamically bound program *may* also involve moving a number of other files as well. Moving (or deleting) a file on which a dynamically bound program depends may prevent that program from functioning.

Setuid Programs

For those programs that execute with an effective UID (user ID) or GID (group ID) different than the real UID or GID, `ld.so` ignores libraries in directories other than `/usr/lib` and `/usr/5lib` in the search path.

4.3. Version Control

A version numbering mechanism has been provided for shared libraries. This allows newer compatible versions of a library to be bound at run-time. It also allows the link editors to distinguish between compatible and incompatible versions of a library.

Version Numbers of `.so`'s

The version number is composed of two parts, a *major* version, and a *minor* version number. This version-control suffix can be extended to an arbitrary string of numbers in Dewey-decimal format, although only the first two components are significant to the link editors at this time.

As noted earlier, `ld` records the version number of the shared library in the executable it builds. When `ld.so` searches for the library at run-time, it uses this number to decide which of the (possibly multiple) versions of a given library is "best," or whether *any* of the available versions are acceptable. The rules it follows are:

- **Major Versions Identical**: the major version used at execution time must exactly match the version found at `ld`-time. Failure to find an instance of the library with a matching major version will cause a diagnostic to be issued and the program's execution terminated.
- **Highest Minor Version**: in the presence of multiple instances of libraries that match the desired major version, `ld.so` will use the highest minor version it finds. However, if the highest minor version found at execution time is lower than the version noted at `ld`-time, a warning diagnostic is issued.

Major version numbers should be changed whenever there is an incompatible change to the library's interface.

NOTE *As always, the detection of incompatibilities between library versions remains the responsibility of the library's developer.*

Version Management Issues

Whenever there is an incompatible change to the library's calling interface, the major number of that library should be changed. A library's interface is defined by:

- the names and types of exported functions and their parameters; and
- the names and types of exported data (initialized or not)

Incompatible changes would include the deletion of a exported procedure, deletion of exported data, changes to an procedure's parameter list, and changes to data structures declared in a `.h` file normally included by both the library and the applications that use it.

Changes to internal library procedures and data do *not* constitute an interface change.

Minor versions should be changed to reflect *compatible* updates to libraries. An example of a compatible update would be changing a procedure's algorithm without changing its parameter list. Although adding a new library routine constitutes an interface change, it can be considered a compatible change.

Note that link-editors silently select the highest compatible version they can obtain. If the minor version used at `ld`-time is *higher* than the highest one found at run-time, then although the interfaces should remain compatible, it is possible that certain bug fixes or compatible enhancements on which the application depends might be missing: hence the warning message mentioned above.

4.4. Shared Library Mechanisms

There is no single mechanism in SunOS that implements shared libraries. Instead, the ability to construct a shared library comes as a consequence of enhancements to various existing facilities. The system components and their features that are instrumental in supporting shared libraries are:

- Virtual memory supports file mapping and ``copy-on-write'' sharing
- PIC generation by the compiler and assembler
- Link editor support for dynamic linking and loading

Memory Sharing

Memory sharing is provided by the kernel's virtual memory (VM) system. The mechanisms of interest for shared libraries are:

- File mapping by way of `mmap()`.
- Sharing at the granularity of a file page
- A per-page copy-on-write facility that allows run-time modification of a shared file, without affecting other users of that same file.

The VM system uses these features internally, so that an `exec()` of a program is reduced to establishing a copy-on-write mapping of the file containing the program. A *shared library* is added to the address space in exactly the same way, using this general file-mapping mechanism.

The C Compiler

The C compiler's `-pic` option generates position-independent code. When `-pic` is specified, references to objects that are external to the body of the code are made by way of *linkage tables*. These indirect references can degrade execution performance slightly, depending on of the number of dynamic references to global objects. The code sequences generated often assume that the linkage tables are no larger than a limit that is convenient for the specific machine (64K bytes for an MC68000, or 8K for a SPARC, for instance). In the (presumably rare) event the tables require a larger size, the compiler can be coerced into generating code sequences that permit larger linkage-table entries with the `-PIC` option.

The Assembler

Code generated by the `-pic` option requires support from the assembler. This support is enabled by the `-k` assembler flag, and is generated automatically by `cc` when invoking the assembler for a compilation performed with the `-pic` or the `-PIC` option.

User-written assembly code for use in a shared object must also be PIC. Refer to the appropriate *Assembly Language Reference* for your Sun system for details.

`crt0()`

Every main program produced by the standard languages is linked with a program prologue module, `crt0()`. This module contains the program's entry point, and performs various initializations of the environment prior to calling the program's `main()` function. `crt0()` refers to the symbol `__DYNAMIC`. As described above, when `ld` builds an executable requiring execution-time link editing, it defines this symbol as the address of a data structure containing information needed for execution-time link editing operations. If the structure is not needed, any reference to the symbol `__DYNAMIC` is relocated to zero.

At program start-up, `crt0()` tests to see whether or not the program being executed requires further link editing. If not, `crt0()` simply proceeds with the execution of the program as it always has – no further processing is involved. However, if `__DYNAMIC` is defined, `crt0()` opens the file `/usr/lib/ld.so` and requests the system to map it into the program's address space via the `mmap()` system call. It then calls `ld.so`, passing as an argument the address of its program's `__DYNAMIC` structure. `crt0()` assumes that `ld.so`'s entry point is the first location in its text. When the call to `ld.so` returns, the link editing operations required to begin the program's execution have been completed.

Link Editors: `ld` and `ld.so`

After `ld` has processed all of its input files, it attempts to resolve each symbolic reference to a relative offset within the executable being built. `ld` is able to complete this *symbolic* reduction at `ld`-time only if:

- all information relating to the program has been given and no `.so` will be added at execution time or
- the program has an entry point and symbolic reduction can be made for those symbols defined in the program

After performing all the reductions it can, if there are no further symbols to resolve, the output is a fully linked (static) executable. However, if any unresolved symbols remain, then the executable will require further link editing at run-time. In this case, `ld` deposits the information (including version number) needed to obtain any needed `.so` files, in the data space of the incomplete executable.

It should be noted that uninitialized ''common'' areas (essentially all uninitialized C globals) are allocated by the link editor *after* it has collected all references. In particular, this allocation can not occur in a program that still requires the addition of information contained in a `.so` file, as the missing information may affect the allocation process. Initialized ''commons,'' however, are allocated in the executable in which their definition appears.

After `ld` has performed all the symbolic reductions it can, it attempts to transform all relative references to absolute addresses. `ld` is able to do this "relative reduction" only if it has been provided some absolute address.

ld.so

At run-time, after receiving control from crt0(), ld.so, executes a short bootstrap routine that performs any relocations ld.so itself requires. It then processes the information contained in the __DYNAMIC structure of the program that called it. ld.so examines the list of required dynamic objects Each element of the list contains an offset relative to the __DYNAMIC structure of an array of link_object structures and has information to identify a .so that must be incorporated. The identification is the name specified on the ld command line used to build the program, and includes a bit indicating whether the object was named explicitly or via a -l option. Some version control information is also recorded for each entry in the ld_need array. ld.so looks up the indicated file, and maps it into the process's address space.

After all modules comprising the program have been placed in the address space, ld.so attempts to resolve the remaining symbols. After performing allocations for all uninitialized commons ld.so attempts to resolve all unbound references that occur *outside of procedure linkage tables.*

Unresolved procedural references in the linkage tables are not processed during program startup. Instead, such references are initialized such that the initial call results in a transfer of control to ld.so. When called in this way, ld.so first resolves the reference to an absolute address, and then modifies the linkage table entry to use that address. Deferring the binding of procedural entry-points until the first call eliminates unnecessary bindings to entry points that the program may never require.

4.5. Building a Shared Library

In the simplest of cases, the commands needed to build a shared library might be:

```
hermes% cc -pic -c *.c
hermes% ld -o libx.so.1.1 -assert pure-text *.o
```

But note that this assumes that the library exports *no* initialized data. And it makes no guarantee that the library text makes the most efficient possible use of space, or allows for a minimal amount of paging.

As noted earlier, a shared library should be structured to avoid undue modification in the course of dynamic linking and execution. Otherwise, it is possible that some or all of the shared text may be rendered unsharable when run. Although this lack of sharing would not effect the *correct execution* of library routines, it will impact system performance. If only a few programs use the library, this impact is small. But for a widely-used library, the impact on system performance could be significant. Thus, shared library objects should be PIC, they should be validated using the pure-text assertion, and those libraries that export initialized data should be accompanied by a data interface description (.sa) file.

PIC Components

Components of shared library code should be PIC. For C source code, the objects must be built with the -pic or -PIC options. User-written assembly code requires that the programmer to follow the same rules that cc followed; details of specific coding sequences can be found in the appropriate *Assembly Language Reference* for a particular Sun system assembler.

Building the .so File

To build the .so portion of a shared library, simply invoke ld with the list of object files that will comprise it. The version number is not automatically generated by ld (which creates a file named a.out by default), but you can specify the full name of the library, including the version number, with ld's -o option. It is strongly suggested that you use the -assert pure-text assertion to uncover any instances of non-PIC code.

The .sa File

The .sa file is used to support ld's -dc option, which provides a space/time efficient implementation of the interface between non-position-independent code and dynamically linked objects. The .so is an archive library that contains only the exported *initialized* data used by the shared library. When present, it is statically linked at ld-time to insure correct allocation.

A data item is *exported* from a library if a program that uses the library refers to the data item *by name*. The contents of the data item are included if they are specified *by value* in the declaration. For instance, with a definition of the form:

```
char *strlist[] = { "string 1", "string 2" };
```

the data itself must be included in the .sa file, whereas with:

```
struct *strlist[] = { ptr1, ptr2 };
```

definitions for the objects named ptr1 and ptr2 would not *necessarily* have to be included. Note that if ptr1 were itself defined as an initialized global in the library source, say:

```
extern char *ptr1 = NULL
```

then this definition would *also* have to go into the .sa file.

Uninitialized data (exported or not) is handled automatically, and need not be included in the .sa file. If the library does not export any data, then a .sa would be unnecessary. The full name of a .sa also includes a version number that must match the version string of the .so it accompanies.

CAUTION **Failure to create a .sa file where it is required risks feeding ld erroneous information about the nature of the interfaces to a library. Furthermore, failure to use a .sa can result in the application's text segment becoming unsharable when run.**

For example, suppose a library's source module contains global initialized data and is compiled with the -R option (merge initialized data with the text segment). Furthermore, this data is not included in the .sa file. When a program references this data, ld.so assumes this a procedure and consequently, it will use this data item improperly.

Building the .sa File

To build a .sa file:

1. Segregate the declarations of exported initialized data from the sources for each object, and place them in a separate source file. Make sure that an up-to-date object is compiled from each of those data-description sources, and include each of those data-description objects in both the static and shared versions of the library.

2. Create a separate (static) archive library composed of *only* the data-description objects, and give it a name of the form '`lib`*name*`.sa.`*version*'. This archive constitutes the `.sa` file. Be sure that the `.sa` has the same version number as the `.so` it is to accompany.
3. Use `ranlib(1)` to incorporate a symbol table within the `.sa` archive.

4.6. Building a Better Library

Library code that maximizes sharing is considered "better" because it makes more efficient use of the system's memory resources. Building the library components PIC is an important and easy first step, but there are other tuning strategies to consider as well.

Sizing Down the Data Segment

One way to maximize sharing is to minimize a `.so`'s data segment (containing initialized data), and its bss segment (containing uninitialized data). Often a `.so`'s data requirements are large because a significant portion of that data that is functionally read-only. There are several problems with this mix of read-only and modifiable data:

- data that could be shared is not,
- an unnecessary amount of swap space is reserved, and
- read-only data fragments the read-write storage, spreading it over more pages.

One approach is to move initialized read-only data into the text segment. This is done by compiling with the `-R` option. However caution needs to be exercised, since *initialized* data structures that contain pointers require relocation at runtime.

For instance, given the declarations:

```
void test();
int x;
struct fxy{
    void (*p0)();
    int *p1;
    };
struct fxy example = {test, &x};
```

The references to `&x` and `test` are instances of pointers embedded in an initialized structure, and should be avoided in shared code. You can avoid problems of this sort by using an uninitialized pointer:

```
struct fxy example;
```

and an adding an initialization routine to set the value(s).

Using xstr to Extract String Definitions

Another common example of initialized data containing pointers is an array of strings:

```
    char *errlist[] = {"err1", "err2"};
```

The xstr(1) utility can be used to make code containing initialized strings more sharable. It segregates the literal string data from its relocatable references, which allows the literal data to be merged safely into the text segment. However, files containing references to the string data should *not* be compiled with the -R option.

If there are several related pieces of data, another strategy is to coalesce the smaller items into a larger structure and allocate the space from the heap.

Better Ordering of Objects

The order of the objects in the executable can be important to minimizing the memory requirements. Since objects are concatenated together, linking in the wrong order may result in a unnecessarily large memory requirement. Two approaches that encourage better utilization of memory resources are:

- Routines that are frequently called should be packaged together, and isolated from startup or rarely-called code.
- A set of routines that represent a common sequence should also be packaged together. For example, given modules A, B, C, D, and E, where A and B fit on one VM page, C and D fit on another, and E fits on a partial page, if A always calls into E and never calls into B, the memory requirements may be reduced by a page if E follows A.

crt0.o Dependency

Sometimes a program will define its own crt0() initial routine. If it is intended that the program use shared libraries, then the programmer needs to provide a hook for the run-time linker. Further discussion of this can be found under link(5) in the *SunOS Reference Manual.*

The ldconfig Command

ldconfig(8) is a program used to construct a run-time linking cache for use by ld.so. The cache has a default list of directories /usr/lib, /usr/5lib, /usr/lib/fsoft, /usr/lib/f68881, /usr/lib/ffpa, and /usr/lib/fswitch and will accept as input a list of additional directories to augment this list. ldconfig records the pathname of the highest compatible version of each shared library in the specified search path.

At runtime, ld.so first queries the cache to determine which is the best version of a library in a particular directory. If the cache is unable to satisfy the request, ld.so enumerates the directory entries for the best version.

4.7. Shared Library Problems

`ld.so` Is Deleted

Since many system utilities are built to use shared libraries, and thus rely on dynamic link-editing, the potential exists for chaos if an important shared library (such as the C library) or `/usr/lib/ld.so` should be deleted.

If the latter has been deleted, you will see the following message:

```
crt0: no /usr/lib/ld.so
```

To deal with the chaos resulting from either the shared C library or `ld.so` being deleted, a number of commands and utilities have been statically linked. These include: `rcp`(1) `init`(8), `getty`(8), `sh`(1), `csh`(1), `mv`(1), `ln`(1), `tar`(1) and `restore`(8). Since most system utilities may be rendered unusable by this condition, it may be necessary to boot the system single-user in order to restore either `/usr/lib/ld.so` or the C library. Refer to *System and Network Administration* for procedures to restore these files.

Wrong Library Is Used

`ld.so` will not *detect* a library that is newly installed in the cache unless the cache is rebuilt using `ldconfig`. Thus, a program that depends on the newly-installed library may not be able to find it. You can use the `ldd`(1) command to identify the libraries on which a program depends.

Error Messages

```
ld.so: libname.so.major not found
```

`ld.so` failed to find a library with the appropriate major version number.

```
ld.so: open error for library
ld.so: can't read struct exec for library
ld.so: library is not for this machine type
```

Either the shared object has been corrupted, has incorrect access permissions, or was built to execute on another processor architecture.

```
ld.so: call to undefined procedure symbol from address
ld.so: Undefined symbol symbol
```

These messages generally indicate that the execution path attempts to refer to an undefined symbol. This is usually the result of a programming error.

```
ld.so: warning library has older version than expected
```

The version of the shared library that is currently being used has a minor version number that is lower than the version that was present at the time the application was compiled.

5

lint — a Program Verifier for C

5

lint — a Program Verifier for C

lint examines C source programs, detecting a number of bugs and obscurities. lint enforces the type rules of C more strictly than the C compiler. lint may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error-prone, constructions which nevertheless are, strictly speaking, legal.

lint accepts multiple input files and library specifications, and checks them for consistency.

The separation of function between lint and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible in part because the compilers do not do sophisticated type checking, especially between separately compiled programs. lint takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This document discusses the use of lint, gives an overview of its implementation, and gives some hints on writing machine-independent C code.

5.1. Using lint

Suppose there are two C source files, *file1.c* and *file2.c*, which are ordinarily compiled and loaded together. The command:

```
tutorial% lint  file1.c  file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. lint enforces the typing rules of C more strictly than the C compiler (for both historical and practical reasons) enforces them. The command:

```
tutorial% lint -p file1.c file2.c
```

produces, in addition to the types of messages described above, additional messages relating to portability of the programs to other operating systems and machines. Replacing the **–p** by **–h** produces messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying **–hp** gets the whole works.

The next several sections describe the major messages; the document closes with sections discussing the implementation and giving suggestions for writing portable C. There is a summary of `lint` options in section `lint` Options.

5.2. A Word About Philosophy

Many of the facts which `lint` needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether `exit` is ever called is equivalent to solving the famous 'halting problem,' which is known to be recursively undecidable.

Thus, most of the `lint` algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, `lint` assumes it can be called; this is not necessarily so, but in practice is quite reasonable.

`lint` tries to give information with a high degree of relevance. Messages of the form '*xxx* might be a bug' are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages which `lint` produces.

5.3. Unused Variables and Functions

As programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These 'errors of commission' rarely make working programs fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!

`lint` complains about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit `extern` statements but are never referenced; thus the statement:

```
    extern  float  sin();
```

will evoke no comment if *sin* is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the **–x** option to the `lint` invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **–v** option is available to suppress the printing of complaints about unused arguments. When **–v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments; this can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when lint is applied to some, but not all, files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The **–u** option may be used to suppress the spurious messages which might otherwise appear.

5.4. Set/Used Information

lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well; many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. lint detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a 'use,' since the actual use may occur at any later time, in a data-dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that lint can complain about some programs which are legal, but these programs would probably be considered bad on stylistic grounds (for example, might contain at least two goto's). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables which are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

5.5. Flow of Control

lint attempts to detect unreachable portions of the programs which it processes. It complains about unlabeled statements immediately following goto, break, continue, or return statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases while(1) and for(;;) as infinite loops. lint also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.

lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to exit may cause unreachable code which lint does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

One form of unreachable statement that lint does not complain about is a break statement that cannot be reached — programs generated by yacc, and especially lex, may have literally hundreds of unreachable break statements. The **–O** option in the C compiler often eliminates the resulting object code inefficiency. Thus, these unreached statements are of little importance — there is typically nothing the user can do about them, and the resulting messages would clutter up the lint output. If these messages are desired, lint can be invoked with the **–b** option.

5.6. Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function 'values' which are never returned. `lint` addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both:

```
    return( expr );
```

and:

```
    return;
```

statements results in the message

```
    function name contains return( expr ) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

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

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a complaint from `lint`. If *g*, like `exit`, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature; it also accounts for a substantial fraction of the 'noise' messages produced by `lint`.

On a global scale, `lint` detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (for example, not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem. Amazingly, this bug has been observed on a couple of occasions in 'working' programs; the desired function value just happened to have been computed in the function return register!

5.7. Type Checking

`lint` enforces the type checking rules of C more strictly than the compiler does. The additional checking is in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

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

The type checking rules also require that, in structure references, the left operand of the —> be a pointer to structure, the left operand of the **.** 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` may be freely matched, as may the types `char`, `short`, `int`, and `unsigned`. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

5.8. Type Casts

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

```
    p = 1 ;
```

where *p* is a character pointer. `lint` will quite rightly complain. Now, 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. It seems harsh for `lint` to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **–c** option controls the printing of comments about casts. When **–c** is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

5.9. Nonportable Character Use

In some implementations, characters are signed quantities, with a range from –128 to 127. In other C implementations, characters take on only positive values. Thus, `lint` will mark certain comparisons and assignments as being illegal or nonportable. For example, the fragment:

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

works on the PDP-11, but will fail on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, `lint` will say 'nonportable character comparison'.

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 because on some machines bitfields are considered as signed quantities. While it may seem unintuitive to consider that a two-bit field declared of type `int` cannot hold the value 3, the problem disappears if the bitfield is declared to have type `unsigned`.

5.10. Assignments of Longs to Ints

Bugs may arise from the assignment of a `long` to an `int`, which may lose accuracy. This may happen in programs which have been incompletely converted to use `typedefs`. When a `typedef` variable is changed from `int` to `long`, the program can stop working because some intermediate results may be assigned to `int`'s, losing accuracy. Since there are a number of legitimate reasons for assigning `longs` to `ints`, the detection of these assignments is enabled by the **–a** option.

5.11. Strange Constructions

`lint` flags several perfectly legal, but somewhat strange, constructions — it is hoped that the messages encourage better code quality, clearer style, and may even point out bugs. The **–h** option is used to enable these checks. For example, in the statement:

```
*p++ ;
```

the * does nothing; this provokes the message 'null effect' from `lint`. The program fragment:

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

is clearly somewhat strange; the test will never succeed. Similarly, the test:

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

is equivalent to:

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

which may not be the intended action. `lint` will say 'degenerate unsigned comparison' in these cases. If one says:

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

`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 which 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 was intended. The best solution is to parenthesize such expressions, and `lint` encourages this by an appropriate message.

Finally, when the **–h** option is in force lint complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many (including the author) to be bad style, usually unnecessary, and frequently a bug.

5.12. Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers, since double-precision values may begin on any integer boundary. On the Honeywell 6000, double-precision values must begin on even word boundaries; thus, not all such assignments make sense. lint tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message 'possible pointer alignment problem' results from this situation whenever either the **–p** or **–h** options are in effect.

5.13. Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine-dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

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

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

will draw the complaint:

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

5.14. Implementation

lint consists of two programs and a driver. The first program is a version of the Portable C Compiler, which is the basis of many C compilers, including Sun's. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file which is passed to a code generator, as the compilers do, lint produces an intermediate file which consists of lines of ASCII text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of `lint`.

5.15. Portability

Many C programs have been successfully ported to a wide variety of operating systems, partly as a result of the `lint` features that increase portability. While there is no guarantee that a given C program will run unmodified within a different system environment, passing it through `lint` identifies and eliminates many potential portability problems.

For instance, uninitialized external variables are treated differently in different implementations of C. Suppose two files both contain a declaration without initialization, such as:

```
int a ;
```

outside of any function. The loader resolves these declarations, and sets aside only a single word of storage for *a*. Under the GCOS and IBM implementations, this is not feasible (for various stupid reasons!) so each such declaration sets aside a word of storage called *a*. When loading or library editing takes place, this creates fatal conflicts which prevent the proper operation of the program. `lint` detects such multiple definitions if it is invoked with the **–p** option.

A related difficulty comes from the amount of information retained about external names during the loading process. On the SunOS system, externally known names have seven significant characters, with the upper/lower case distinction kept. On the IBM systems, there are eight significant characters, but the case distinction is lost. On GCOS, there are only six characters, of a single case. This leads to situations where programs run on one system, but encounter loader problems on others. `lint` **–p** maps all external symbols to one case and truncates them to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling: characters in SunOS are eight bit ASCII, while they are eight bit EBCDIC on the IBM, and nine bit ASCII on GCOS. Moreover, character strings go from high to low bit positions ('left to right') on GCOS and IBM, and low to high ('right to left') on the PDP-11. This means that code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, must be looked at with great suspicion. `lint` is of little help here, except to option multi-character character constants.

Of course, the word sizes are different! This is less troublesome than might be expected, however. The main problems are likely to arise in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing:

```
    x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on the PDP-11, but fails badly on GCOS and IBM. If the bit field feature cannot be used, the same effect can be obtained by writing:

```
    x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the PDP-11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed `unsigned`. Characters are considered signed integers on the PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, C could be changed; in any case, `lint` is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of system utilities has been the inability to mimic essential system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, `lint` has been very helpful in moving the operating system and associated utility programs to other machines.

5.16. Shutting `lint` Up

There are occasions when the programmer is smarter than `lint`. There may be valid reasons for 'illegal' type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by `lint` often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with `lint`, typically to shut it up, is desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to make `lint` recognize a number of words when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, `lint` directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the `lint` directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to `lint`, this can be asserted by placing the directive

```
/*NOTREACHED*/
```

just before that spot in the program. The –v option can be turned on for one function by the directive:

```
/*ARGSUSED*/
```

Complaints about variable numbers of arguments in calls to a function can be turned off by the directive:

```
/*VARARGS*/
```

preceding the function definition. In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the `VARARGS` keyword immediately with a digit giving the number of arguments which should be checked; thus,

```
/*VARARGS2*/
```

checks the first two arguments and leaves the others unchecked. Finally, the directive:

```
/*LINTLIBRARY*/
```

at the head of a file identifies this file as a library declaration file; this topic is worth a section by itself.

5.17. Library Declaration Files

`lint` accepts certain library directives, such as:

```
-ly
```

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

```
/*LINTLIBRARY*/
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The `VARARGS` and `ARGSUSED` directives can be used to specify features of the library functions.

`lint` library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined in a library file, but not used in a source file, draw no complaints. `lint` does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

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

5.18. Considerations When Using lint

lint was a difficult program to write, partially because it is closely connected with matters of programming style, and partially because users usually don't notice bugs which cause lint to miss errors which it should have caught. By contrast, if lint incorrectly complains about something that is correct, the programmer reports that immediately!

A number of areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of typedef is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

lint shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the preprocessor to be constructed which checks for things such as unused macro definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with lint is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features. There are pressures to add even more of these options.

In conclusion, it appears that the general notion of having two programs is a good one. The compiler concentrates on quickly and accurately turning the program text into bits which can be run; lint concentrates on issues of portability, style, and efficiency. lint can afford to be wrong, since incorrectness and over-conservatism are merely annoying, not fatal. The compiler can be fast since it knows that lint will cover its flanks. Finally, the programmer can concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of lint, the desirable properties of universality and portability.

5.19. lint Options

The lint command currently has the form

```
tutorial% lint [-abchnpsuvx ] filename. . . library-descriptors.
```
.

The options are

a Report assignments of long to lint or shorter

b Report unreachable break statements

c Complain about questionable casts

h Perform heuristic checks

n	Do not do library checking
p	Perform portability checks
s	Same as h (for historical reasons)
u	Don't report unused or undefined externals
v	Don't report unused arguments
x	Report unused external declarations

6

Performance Analysis

6

Performance Analysis

Tools discussed in this chapter cover facilities for timing programs and getting performance analysis data. Some tools work only with the C programming language, while others will work on modules written in any language. Performance analysis tools provide a variety of levels of analysis from very simple timing of a command down to a statement-by-statement analysis of a program. You can select which level of granularity you like depending on the amount of detail and optimization you wish to perform. Here are the performance analysis tools available from the simplest to the most detailed:

`time` A simple command (built in to the C shell) to display the time that a program takes. The C shell's built in `time` command display statistics about how a command uses the system resources as well as just the raw time consumed.

`prof` Generates a profile for the modules in a program, showing which modules are using the time.

`gprof` Generates not only a profile as for `prof`, but also generates a *call graph* showing what modules call which, and which modules are called by other modules. The call graph can sometimes point out areas where removing calls can speed up a program.

`tcov` Generates a detailed statement-by-statement analysis of a C program.

6.1. `time` — Display Time Used by a Program

Two distinct versions of the `time` command exist in the Sun system. Here we discuss the `time` command that is built in to the C shell. The other `time` command is a program (in `/bin/time`) that you get when you use the Bourne shell.

As a first example, we show the `time` command being used to display statistics on the run-time of the `index.assist` program we've used in other examples in this manual. In all the examples shown here we direct the output from `index.assist` into `/dev/null`. Here is the simplest example of using `time`:

```
tutorial% time index.assist < index.entries > /dev/null
13.5u 0.8s 0:15 92% 3+19k 19+1io 0pf+0w
```

Now to explain the items in the display from the time command above.

The 13.5u means that this program used 13.5 seconds of *user* time — time spent in the application program itself. The 0.8s means that the program spent 0.8 seconds in the *system* — this is time spent in the operating system kernel on behalf of the program. The third field is the elapsed or wallclock time for the application. The percentage figure is the percent of the user and system time as a fraction of the elapsed time. The rest of the display is of lesser interest just now and is explained in more detail below.

Effects of Optimizer on Timing

Just for the sake of interest, let's see what effect the C optimizer has on the run time of this program — we make the program with the -O option and see what happens:

```
tutorial% time index.assist < index.entries > /dev/null
13.1u 1.4s 0:38 37% 3+19k 19+0io 1pf+0w
```

What has happened here? The optimized version takes longer to run! This demonstration tells us that simple timing is not so simple after all — in a multitasking system there are many other factors that can effect the simple timing. Note that the user time for the program is actually slightly less — 0.4 seconds less. But, the system time and the elapsed time are very different. These timings are affected by the load on the system. If we look at the last field in the time display, note that in the unoptimized version there were zero page faults, while in the optimized version there was one page fault. This is an indication that there was other activity in the system at the time the program was run and this other activity will adversely affect the elapsed time. There are two rules you can apply to this situation:

- Run such timing tests on a quiet system late at night. Make sure that 'late at night' is not midnight when a whole bunch of cron daemons start up.
- Run timing tests several times and take averages.

Controlling the display from the time Command

The time command built into the C shell has the capability of altering the information displayed under control of an environment variable. This is not true of /bin/time — the command you'd have to use if you were using the Bourne shell. Here is how to set up the time variable to control the time display.

You can control how the C shell times programs by setting the time variable in your .login or .cshrc file.

The time variable can be supplied with one or two values, such as set time=3 or set time=(3 "%E %P%").

Setting the time variable via a set command of the form:

```
set time=nnn
```

means that the shell displays a resource-usage summary for any command running for more than *nnn* CPU seconds.

Control Key Letters for the time Command

The second form controls exactly what resources are displayed. The character string can be any string of text with embedded control key-letters in it. A control key-letter is a percent sign (`%`) followed by a single *upper-case* letter. To print a percent sign, use two percent signs in a row. Unrecognized key-letters are simply printed. The control key-letters are:

Table 6-1 *Control Key Letters for the* `time` *Command*

Letter	*Description*
D	Average amount of unshared data space used in Kilobytes.
E	Elapsed (wallclock) time for the command.
F	Page faults.
I	Number of block input operations.
K	Average amount of unshared stack space used in Kilobytes.
M	Maximum real memory used during execution of the process.
O	Number of block output operations.
P	Total CPU time — U (user) plus S (system) — as a percentage of E (elapsed) time.
S	Number of seconds of CPU time consumed by the kernel on behalf of the user's process.
U	Number of seconds of CPU time devoted to the user's process.
W	Number of swaps.
X	Average amount of shared memory used in Kilobytes.

Default Timing Summary

The default resource-usage summary is a line of the form:

```
uuu.u u sss.s s ee:ee pp % xxx +dddk iii +ooo io mmm pf+ww w
```

Table 6-2 *Default Timing Summary Chart*

Field	*Description*
uuu.u	user time (U),
sss.s	system time (S),
ee:ee	elapsed time (E),
pp	percentage of CPU time versus elapsed time (P),
xxx	average shared memory in Kilobytes (X),
ddd	average unshared data space in Kilobytes (D),
iii and *ooo*	the number of block input and output operations respectively (I and O),
mmm	number of page faults (F)
ww	number of swaps (W).

C shell `time` Command versus `/bin/time`

One final note on the `time` commands. As mentioned previously, there are two versions of `time`: the one built in to the C shell as described above, and the original Bourne shell `time` command which can be found in `/bin/time`.

The C shell `time` command does not time a command which is a component of a pipeline. This is what happens:

```
tutorial% echo timing a pipeline | time cat
timing a pipeline
```

whereas the Bourne shell `time` command gives completely different results:

```
tutorial% echo timing a pipeline | /bin/time cat
timing a pipeline
        0.8 real         0.0 user         0.1 sys
```

6.2. `prof` — Generate Profile of a Program

After simple timing, a *profile* of a program displays a finer level of analysis to assist in optimizing performance. Getting a profile is the next step after simple timing — more detailed analysis is provided by the *call-graph* profile and the *code coverage* tools described later in this chapter.

Taking the `index.assist` program from before as an example, let's make the program compiled for profiling. To compile a program for profiling, you use the `-p` option to the C compiler:

```
tutorial% make CFLAGS=-p
                    .
                    .
        messages from the make command
                    .
                    .
```

Now we can run the index.assist program as before. When a program is profiled, the results appear in a file called `mon.out` at the end of the run. Every time you run the program a new `mon.out` file is created, overwriting the old version. You then use the `prof` command to interpret the results of the profile, as shown by the example below.

```
tutorial% index.assist < index.entries > /dev/null
tutorial% prof index.assist
 %time  cumsecs  #call  ms/call  name
  19.4     3.28  11962     0.27  _compare_strings
  15.6     5.92  32731     0.08  _strlen
  12.6     8.06   4579     0.47  __doprnt
  10.5     9.84                  mcount
   9.9    11.52   6849     0.25  _get_field
   5.3    12.42    762     1.18  _fgets
   4.7    13.22  19715     0.04  _strcmp
   4.0    13.89   5329     0.13  _malloc
   3.4    14.46  11152     0.05  _insert_index_entry
   3.1    14.99  11152     0.05  _compare_entry
   2.5    15.41   1289     0.33  lmodt
   0.9    15.57    761     0.21  _get_index_terms
   0.9    15.73   3805     0.04  _strcpy
   0.8    15.87   6849     0.02  _skip_space
   0.7    15.99     13     9.23  _read
   0.7    16.11   1289     0.09  ldivt
   0.6    16.21   1405     0.07  _print_index
                    .
                    .
      everything else is insignificant
                    .
                    .
```

This display points out that most of the program's running time is spent in the routine that compares character strings to establish the correct place for the index entries, and that after that, the majority of the time is spent in the `_strlen` library routine — to find the length of a character string. If we wish to make any appreciable improvements to the program we must concentrate our efforts on the `compare_strings` function.

Interpreting Profile Display

Let's interpet the results of the profiling run though. The results appear under these column headings:

```
%time  cumsecs  #call  ms/call  name
```

Here's what the columns mean:

`%time` Percentage of the total run time of the program, that was consumed by this routine.

`cumsecs` A running sum of the number of seconds accounted for by this function and those listed above it. This information isn't really worth much — the important data comes from the percentage of total time and from the time consumed per call.

`#call` The number of times this routine was called.

`ms/call` How many milliseconds this routine consumed each time it was called.

`name` The name of the routine.

Now what advice can we derive from the profile data? Notice that the `compare_strings` function consumes nearly 20% of the total time. To improve the run time of `index.assist` we must either improve the algorithm that `compare_strings` uses, or we must cut down the number of calls. Not obvious from the flat profile is the information that `compare_strings` is heavily recursive — we get that fact from using the call graph profile described below. In this particular case, improving the algorithm also implies reducing the number of calls.

6.3. gprof — Generate a Call Graph Profile

While the flat profile described in the last section can provide valuable data for performance improvements, sometimes the data obtained is not sufficient to point out exactly where the improvements can be made. A more detailed analysis can be obtained by using the *call graph* profile that displays a list of which modules are are called by other modules, and which modules call other modules. Sometimes, removing calls altogether can result in performance improvements.

Compiling with the -pg Option

Using the same `index.assist` program an example, let's make the program compiled for call-graph profiling. To compile a program for call-graph profiling, you use the `-pg` option to the C compiler:

```
tutorial% make CFLAGS=-pg
                      .
                      .
          messages from the make command
                      .
                      .
```

Now we can run the index.assist program as before. When a program is call-graph profiled, the results appear in a file called `gmon.out` at the end of the run. You then use the `gprof` command to interpret the results of the profile:

```
tutorial% index.assist < index.entries > /dev/null
tutorial% gprof index.assist
                      .
                      .
          voluminous output from the gprof command
                      .
                      .
```

Output from gprof

The output from `gprof` is really voluminous — it's usually intended that you take the summaries away and read them later. The output from `gprof` consists of the two major items listed below.

- The 'flat' profile. This is similar to the summary that the `prof` command supplies. `gprof` gives you slightly more information. The output from `gprof` contains an explanation of what the various parts of the summary mean, so you don't need to go look the things up in a manual.
- The full call-graph profile. There are some fragments of the output from the profiling run just below with some examples of how to interpret them.

The output from `gprof` contains an explanation of what the various parts of the summary mean, so you don't need to go look the things up in a manual.

Interpreting Call Graph

Here is a fragment of the output from the `gprof` summary. Most of the output has been deleted from before and after the fragment. One thing that `gprof` does tell you is the granularity of the sampling:

```
granularity: each sample hit covers 4 byte(s) for 0.14% of 14.74 seconds
```

Then comes part of the call-graph profile itself:

```
                              called/total      parents
index  %time    self descendents  called+self    name           index
                              called/total      children

----------------------------------------------

                0.00       14.47      1/1           start [1]
[2]     98.2    0.00       14.47      1          _main [2]
                0.59        5.70    760/760         _insert_index_entry [3]
                0.02        3.16      1/1           _print_index [6]
                0.20        1.91    761/761         _get_index_terms [11]
                0.94        0.06    762/762         _fgets [13]
                0.06        0.62    761/761         _get_page_number [18]
                0.10        0.46    761/761         _get_page_type [22]
                0.09        0.23    761/761         _skip_start [24]
                0.04        0.23    761/761         _get_index_type [26]
                0.07        0.00    761/820         _insert_page_entry [34]

----------------------------------------------

                               10392                _insert_index_entry [3]
                0.59        5.70    760/760         _main [2]
[3]     42.6    0.59        5.70    760+10392    _insert_index_entry [3]
                0.53        5.13   11152/11152      _compare_entry [4]
                0.02        0.01     59/112         _free [38]
                0.00        0.00     59/820         _insert_page_entry [34]
                               10392                _insert_index_entry [3]

----------------------------------------------
```

Noting that there are 761 lines of data in the input file to the index.assist program, here are some of the things we can determine from the call graph:

- fgets is called 762 times — one more than the number of lines in the input file. The last call to fgets returns an end-of-file.
- The insert_index_entry function is called 760 times from main — one less times than the number of lines. Why is this? The first index entry is inserted 'manually' in the main function when there are no previous index entries to insert.
- Note that in addition to the 760 times that insert_index_entry is called from main, insert_index_entry also calls *itself* the grand total of 10392 times — insert_index_entry is heavily recursive. Index entries appear in the input file in unsorted order and are sorted on the fly by inserting them into a binary tree.
- Note also that compare_entry (which is called from insert_index_entry) is called 11152 times, which is equal to 760+10392 times, so there is one call of compare_entry for every time that insert_index_entry is called. This is as it should be. If there was a discrepancy in the number of calls, we might suspect some problem in the program's logic.
- Notice the number of calls to the insert_page_entry and free() functions — insert_page_entry is called 820 times in total: 761 times from main while the program is building index nodes, and then insert_page_entry is called 59 times from insert_index_entry. This indicates that there are 59 index entries that are duplicated, so their page number entries are linked into a chain with the index nodes. The duplicate index entries are then freed, hence the 59 calls to free().

6.4. tcov — Statement-Level Analysis

After a certain level of performance enhancements have been made, the profile data obtained from a program starts to look 'flat' and the granularity of the data collection makes further improvements difficult. At this point, you can use a tool that performs statement-by-statement analysis on a program, showing which statements are executed and how many times. This facility is called *code coverage.*

Code coverage can also be valuable in identifying areas of 'dead' code — areas of code that never get executed. Code coverage can also point out areas of code that are not being tested.

Compiling with the -a Option

Using the same index.assist program an example, let's make the program compiled for code coverage. To compile a program for code coverage, you use the -a option to the C compiler, as shown by the example below.

```
tutorial% make CFLAGS=-a
                    .
                    .
         messages from the make command
                    .
                    .
```

For every *thing*.c file you compile with the -a option, the C compiler generates a *thing*.d file — these are used by the code coverage program later in the analysis.

Using tcov

Now we can run the index.assist program as before. After a program has been run, you can then run tcov to get the summaries of execution counts for each statement in the program:

```
tutorial% index.assist < index.entries > /dev/null
tutorial% tcov *.c
```

Now, for every *thing*.c file you specify, tcov uses the *thing*.d file and generates a *thing*.tcov file containing and annotated listing of your code. The listing shows the number of times each source statement was executed. At the end of each *thing*.tcov file there is a short summary.

Below is a small fragment of the C code from one of the modules of index.assist — the module in question is the insert_index_entry function that's called so recursively.

```
                 struct index_entry *
         insert_index_entry(node, entry)
11152 ->        struct   index_entry       *node;
                struct   index_entry       *entry;
         {

                int      result;
                int      level;

                result = compare_entry(node, entry);

                if (result == 0) {         /*  exact match  */
                                           /*  Place the page entry for the duplicate */
                                           /*  into the list of pages for this node */
   59 ->                 insert_page_entry(node, entry->page_entry);
                         free(entry);
                         return(node);
                }

11093 ->        if (result > 0)            /*  node greater than new entry -- */
                                           /*  move to lesser nodes */
 3956 ->                 if (node->lesser != NULL)
 3626 ->                          insert_index_entry(node->lesser, entry);
                         else {
  330 ->                          node->lesser = entry;
                                  return (node->lesser);
                         }
                else                       /*  node less than new entry -- */
                                           /*  move to greater nodes */
 7137 ->                 if (node->greater != NULL)
 6766 ->                          insert_index_entry(node->greater, entry);
                         else {
  371 ->                          node->greater = entry;
                                  return (node->greater);
                         }
         }
```

Notice that the `insert_index_entry` function is indeed called 11152 times as we determined in the output from `gprof`. The numbers to the side of the C code show how many times each statement was executed.

`tcov` Summary

Below is the summary that `tcov` placed at the end of `build.index.tcov`.

```
             Top 10 Blocks

             Line      Count

              240      21563
              241      21563
              245      21563
              251      21563
              250      21400
              244      21299
              255      20612
              257      16805
              123      12021
              124      11962


      77   Basic blocks in this file
      55   Basic blocks executed
   71.43   Percent of the file executed

          439144   Total basic block executions
         5703.17   Average executions per basic block
```

7

SCCS — Source Code Control System

7

SCCS — Source Code Control System

The Source Code Control System (SCCS) is a collection of commands that control changes to selected files, such as the source files for programs and software projects. SCCS allows you to:

1. Place a file under the control of SCCS. Once a file is under SCCS control, copies of any subsequent version can be extracted from a history file.
2. Check a file out for editing and lock it, so that only you can make changes.
3. Check in a new version of the file that incorporates your changes. When you check a file in, you can also supply comments that summarize the changes you've made.
4. Back out your changes if necessary.
5. Inquire about the status or current version of a file.
6. Inquire about the line-by-line differences between versions.
7. Inquire about the version history, including a record who checked in which changes, and when they did so.

Collectively, functions such as these are referred to as *version control*. They are important in situations where source files are updated frequently, perhaps by more than one person, or where files need to be audited. SCCS allows you to recover the current, or any previous version of a file, as needed. It reduces the amount of data that must be kept on disk by recording only the *differences* between successive versions. With this information, SCCS can reconstruct the initial version, the current version, or any version in between.

Low-Level SCCS Commands vs. the `sccs` Command

The Source Code Control System, or SCCS, consists of a set of low-level commands to perform individual functions, and a high-level front-end-command called `sccs`.

The `sccs` command provides a reasonable and consistent user interface to the various and sundry low-level commands. Although they *can* be used directly, the low-level commands are more difficult to work with. The remainder of this chapter describes the high-level `sccs` command. Refer to Appendix A, *SCCS Low-Level Commands*, for information about the SCCS low-level commands.

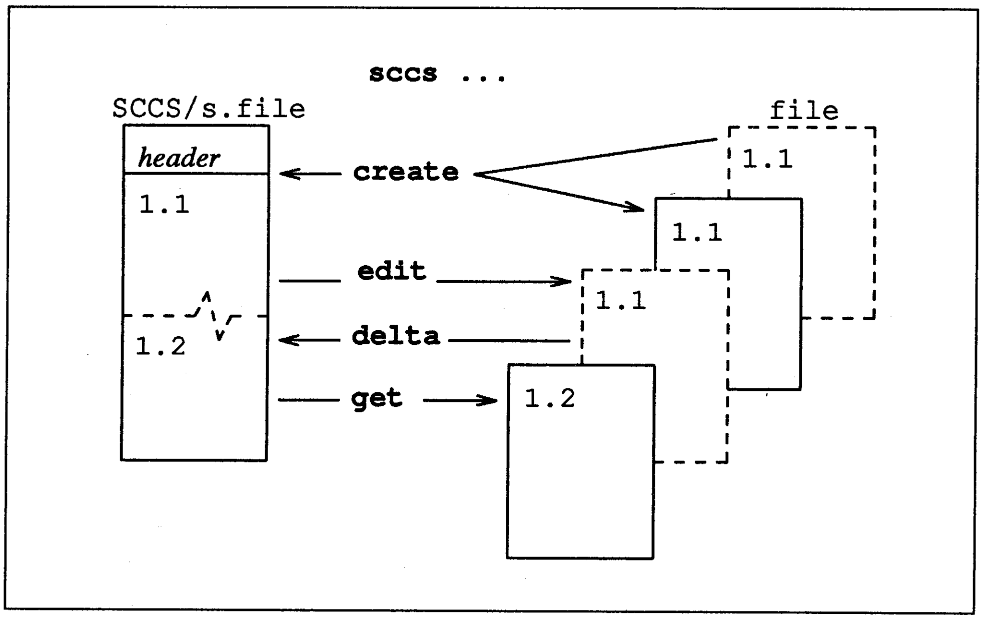
Conventions

Throughout this chapter, we assume that you are using the C shell on a system called 'tutorial', and so the hostname is shown followed by the % prompt in the examples. What you type is shown in **`bold typewriter text like this`**, and the system's responses are shown in `typewriter font like this`.

```
tutorial% sccs create prog.c

prog.c:
1.1
87 lines
tutorial%
```

A record of each version of your source file, along with the version log and other information, is kept in a history file. This history file is also called an *s.file* (''s-dot-file''). The illustration below shows the four basic version-control operations provided by `sccs`, and how they effect the history file.

Figure 7-1 *Basic* `sccs` *Subcommands*



As the picture illustrates, there are four basic `sccs` subcommands that operate on the s.file:

- `create` the history file.
- `edit`, or check out a file for editing. This operation extracts a version of the file that is writable only by you, and locks the history file so that no one else can get an editable copy.
- `delta` (merge) changes that you've made back into the s.file. This is the complement to the `sccs edit` operation. Line-by-line differences (see `diff(1)`) are recorded in the history file; the set of differences associated

with a given version of the file is called a *delta*. A new version number is assigned, and you are prompted for your comments, which are added to the header along with other information about the new version.

□ `get` a read-only copy of the file. This operation extracts, or *gets* a version of the file from the s.file. By default, a read-only copy of the latest version is retrieved. The read-only copy is intended to be static, can be used as a source file for compilation, printing, or whatever—it is specifically *not* intended to be edited or changed in any way. (Attempting to bend the rules by changing permissions of an extracted version, and then editing it, can result in your changes being lost. If you want to edit a file under SCCS control, check it out using `sccs edit`.)

The s.file is the final authority and archive for whatever SCCS-file you are working with. The version you get using either `sccs get` or `sccs edit` is merely a *copy* derived from data in that file; if deleted, the current version can be gotten once again from the history file. Of course, if you have a file checked out for editing, you must take care to check in any changes you wish to incorporate.

Backing Out Pending Changes

Changes that have been made to a checked-out version, but are not yet checked in, are said to be *pending*. You can use the **`sccs unedit`** command to back out changes that are still pending. This comes in handy if a version of the file should become damaged during editing. The **`unedit`** subcommand removes the checked-out version, unlocks the history file, and gets a read-only copy of the most recent version checked in. In other words, after using `unedit`, it is as if you hadn't checked the file out in the first place.

7.1. Terminology

There are a number of terms worth learning before going any farther.

S.File, or History File

The *s.file* is the history file or archive containing the information needed to get any desired version of a file; it contains only the original version, and a record of the differences between versions, rather than the entire text of each version. This saves disk space, since there is no need to duplicate the lines that *haven't* changed between versions, and it allows selective changes to be removed later. The s.file also includes some header information for each version, including comments that a user provides when checking in each version.

SCCS-File

A file under SCCS control is sometimes referred to as an SCCS-file. In some cases, the history file is also referred to in this way. The context in which the term is used usually makes clear which is meant.

Deltas

A *delta* is a set of line-by-line differences associated with a given version of the file. A delta only includes the specific changes made between two successive versions. Normally, extracted versions reflect all deltas made earlier. However, it is possible to get a version that omits selected deltas associated with specific (earlier) versions.

SIDs, or Version Numbers

An SID, or SCCS-ID, is a number that represents a delta. This is normally a two-part number, starting with `1.1` composed of of a *release* number, and a *level* number. The level number is incremented with each new version. Note that the version number is *only* associated with the particular set of differences between two successive versions of a file. It does *not* represent the cumulative set of changes from the original. However, since versions are normally extracted so as to reflect the accumulated changes, the SID of the most recent delta is often used to represent the version of the file that it corresponds to.

The release number is normally carried forward between versions; it is possible to alter the release number using low-level commands.

ID keywords

SCCS recognizes and expands certain keywords of the form:

```
%X%
```

where *X* is an upper case letter. These *ID Keywords* can be used to introduce the current version number, as well as other information, into the read-only (extracted, but not checked out for editing) versions of the file. For instance, `%I%` expands to the SID of the most recent delta checked in. `%W%` includes the filename, the SID, and the (rather unique) string `@(#)`, which is recognized by certain SCCS commands, and makes the expanded SID easy to search for. The `%G%` keyword expands to the date of the latest delta. Other ID keywords are listed in Appendix A, under *Identification Keywords.*

For example, a line such as:

```
static char SccsId[ ] = "%W%\t%G%";
```

will be replaced with something like:

```
static char SccsId[ ] = "@(#)prog.c  1.2  08/29/80";
```

This tells you the name and version of the source file and the time the delta was created. The string `@(#)` is a special string that signals the beginning of an expanded SCCS-ID keyword.

When you check out a file for *editing*, the ID keywords are *not expanded*; they are only expanded when you `get` a read-only version. If a version of a file with the keywords already expanded should happen to be checked in, version-dependent information is no longer updated automatically because the unexpanded keywords are replaced with text. To alert you to this situation, if an SCCS command finds no ID keywords in a version being checked in, it give you the warning:

```
No Id Keywords (cm7)
```

Note that this does not prevent the file from being checked in or out.

ID keywords can be inserted anywhere in a file, and are typically inserted in comments. They can also be compiled into an object file, as shown in the example below.

```
static char SccsId[ ] = "%W%    %G%";
```

While this allows the version information to be compiled into the object file, it also takes up data space when the program is run. If you use this technique to put ID keywords into header files, it is important to use a different variable for each header. Otherwise, you will get an error when the compiler attempts to redefine the variable. However, if a header file is included by many files that are subsequently loaded together, the version information for that file may be included in the object file several times; you may find it more to your taste to place the ID keywords for header files within comments:

```
/* %W%  %G% */
```

7.2. Creating SCCS History Files with `sccs create`

To put a set of source files under SCCS control, you must:

- Make a subdirectory called `SCCS`, if it isn't there already (note that `SCCS` is in upper-case, so that will appear near the top of an `ls` directory listing):

```
tutorial% mkdir SCCS
```

- Use the `sccs create` command to create the history files for each source file. Suppose that you want to have all your *.c* and *.h* files under SCCS control:

```
tutorial% sccs  create  *.[ch]
```

For each filename argument you supply, the `sccs create` command:

creates a file called *s.filename* in the `SCCS` subdirectory,

renames each *filename* by placing a comma in front of the name, so that you end up with files of the form *,filename*.

gets (extracts) a read-only copy of each *filename* by using the `sccs get` command.

After verifying that `sccs` has correctly created the s.files, you can remove the filenames starting with a comma.

If you want to embed ID keywords in the files, it is best to put them in before you create the s.files. If you do not, *create* will print the warning message:

```
No Id Keywords (cm7)
```

You can add the keywords in the same way that you would make any other change to the file. Check it out for editing using `sccs edit`, add the keywords, and check it back in using `sccs delta`.

7.3. Extracting Current Versions with `sccs get`

To get a copy of the most recent version of a file, use the command:

```
sccs get filename ...
```

For example, the command:

```
tutorial% sccs get prog.c
```

`sccs` responds with the version number, and the number of lines extracted:

```
1.1
87 lines
```

meaning that a version containing cumulative deltas through 1.1 has been retrieved, and that it contains 87 lines. The file `prog.c` is created in the current directory. It's permissions are set to read-only, which indicates that no one has it checked out for editing.

This copy of the file should not be changed, since `sccs` cannot merge the changes back into the s.file unless the file has been checked out. If you do manage to force in some changes, those changes may well be lost the next time someone does an `sccs get`, or `sccs edit`.

7.4. Changing Files (Creating Deltas)

To change a version of a file, you must obtain a copy of the file that can be edited. You obtain such a copy using `sccs edit` as shown below. Having made the changes and satisfied yourself that the changes are correct, you can then merge the changes back into the `sccs` history file using `sccs delta` also shown below.

Retrieving a File for Editing with `sccs edit`

To edit a source file, you must first `get` it, requesting permission to edit it[4]. The response will be the same as with `sccs get` except that it also says that a new delta is being created:

```
tutorial% sccs edit prog.c
New delta 1.2
```

You can then edit it, using a text editor:

```
tutorial% vi prog.c
```

Merging Changes Back Into the S.File with `sccs delta`

When the desired changes have been made, you can put your changes into the SCCS-file using the *delta* command:

```
tutorial% sccs delta prog.c
```

[4] The `sccs edit` command is equivalent to using the `-e` option to `sccs get`.

Delta prompts you for 'comments?' before merging the changes in. At this prompt you should type a one-line description of what the changes mean (more lines can be entered by ending each line except the last with a backslash). *Delta* then types:

```
1.2
5 inserted
3 deleted
84 unchanged
```

saying that delta 1.2 was created, and it inserted five lines, removed three lines, and left 84 lines unchanged[5]. The `prog.c` file is then removed; it can be retrieved using `sccs get`.

If you give several filename arguments to `delta`, they will all be checked in with the same comment.

Version Control for Binary Files

Although `sccs` is typically used for source files that contain ASCII text, this version of SCCS allows you to apply version control to binary files (files that contain NULL or control characters, or do not end with a NEWLINE). The binary files are encoded[6] into an ASCII representation as they are checked in, and versions are decoded as they are extracted.

Version control functions can be useful for data files, such as icons, raster images, or screen font tables, that you may wish to edit and track. For instance, you might want to track changes to a screen font, which can be created and maintained using `fontedit(1)`. When you `create` or `delta` a binary file such as this, you get the warning message:

```
Not a text file (ad31)
```

You may also get the:

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

message. Otherwise, everything proceeds normally, as shown by the example below.

[5] Changes to a line are counted as a line deleted and a new line inserted.

[6] See `uuencode(1C)` for details.

```
tutorial% sccs create special.font

special.font:
Not a text file (ad31)
No id keywords (cm7)
1.1
20 lines
No id keywords (cm7)
tutorial% sccs get special.font
1.1
20 lines
tutorial% file special.font SCCS/s.special.font
special.font:   vfont definition
SCCS/s.special.font:    sccs
```

You can use `sccs create -b` to force `sccs` to treat a file as a binary file as opposed to a text file.

Since binary files (and their encoded representation) can vary significantly between versions, their history files tend to grow at a much faster rate than text file histories. In fact, when it comes to archiving object files and executables, it can take less disk space simply to store each version of the file as is. Fortunately, those files are not normally edited, so they don't really require version control. Using `sccs` to control the source files from which an object file is built, and using `make` to build it in a consistent manner, is a more practical method for maintaining object files and executable programs.

When Making Deltas

A little forethought helps when deciding whether to check in a file. Making a new delta after every single edit during the debugging phase, for instance, can get to be excessive. On the other hand, leaving a file checked out for so long that you forget about it can be very inconvenient for someone else who may need to edit it later.

So long as you are certain that you are the only one who requires access to the file, it makes sense to complete a set of related changes before checking the file back in.

When you provide comments for a delta, it is important to make them meaningful. You may have to return to the file several months later, at which time a useful summary of what you've done in each delta will be a big help. Numerous marginal deltas with meaningless comments such as:
''fixed compilation problem in previous delta,'' or, ''fixed botch in 1.3.'', are seldom helpful or welcome.

It is very important to check in all changes before compiling or installing a module for general use. A good technique is to `edit` the files you need, make all necessary changes and tests, compile and debug the files repeatedly until you are satisfied, and make a delta. After making the delta, it is a good idea to `get` the files, and then recompile and/or install the finished versions.

Finding Out What's Going On with sccs info

To find out what files are being edited, type:

```
tutorial% sccs info
```

to display a list of all the files being edited and other information — such as the name of the user who did the edit. Also, the command:

```
tutorial% sccs check
```

is nearly equivalent to the *info* command, except that it is silent if nothing is being edited, and returns a non-zero exit status if anything is being edited. It can thus be used in an 'install' entry in a makefile to abort the installation if anything has not been properly delta'ed.

If you know that everything being edited should be delta'ed, you can use:

```
tutorial% sccs delta `sccs tell`
```

The *tell* command is similar to *info* except that only the names of files being edited are output, one per line.

All of these commands take a –b option to ignore 'branches' (alternate versions, described later) and the –u option to give only files being edited by you. The –u option takes an optional *user* argument, giving only files being edited by that user. For example:

```
tutorial% sccs info -ujohn
```

gives a listing of files being edited by user john.

Finding Out What Versions Are Being Used with sccs what

To find out what version of a program is being run, use:

```
sccs what prog.c /usr/bin/prog
```

which will print all strings it finds that begin with '@(#)'. This works on all types of files, including binaries and libraries, provided that the ID keywords have been compiled in. For example, the above command will output something like:

```
tutorial% sccs what prog.c /usr/bin/prog
prog.c:
        prog.c  1.2     08/29/80
/usr/bin/prog:
        prog.c  1.1     02/05/79
```

From this one can see that the source in prog.c will not compile into the same version as the binary in /usr/bin/prog.

Keeping SIDs Consistent Across Files

With some care, it is possible to keep the SID's consistent in multi-file systems. The trick here is to always `sccs edit` all files at once. The changes can then be made to whatever files are necessary and then all files (even those not changed) are `delta`'ed. This can be done fairly easily by just specifying the `SCCS` subdirectory as the filename argument to both `edit` and `delta`:

```
tutorial% sccs edit SCCS
...
tutorial% sccs delta SCCS
```

With the `delta` subcommand, are prompted for comments only once; the comment is applied to all files being checked in.

Creating New Releases

To create a new release of a program, specify the release number you want to create when you check the file out for editing, using the `-r`*n* option to `edit`; *n* is the new release number:

```
tutorial% sccs edit -r2 prog.c
```

In this case, when the new version is `delta`'ed, it will be the first level delta in release 2, with SID 2.1. To change the release number for all SCCS-files in the directory, use:

```
tutorial% sccs edit -r2 SCCS
```

7.5. Restoring Old Versions

Reverting to Old Versions

`sccs` allows you to extract any previously checked-in version of a file. This can come in handy if you need to backtrack to an earlier version. In this case, you can check out the current version, extract a writable copy of an earlier ''good'' version (under a different name) using a command of the form:

`sccs get -k -r`*SID* `-G`*newname filename*

and then move the old version to the given *filename*, and check the file back in.

Getting a Delta by Date

In some cases you don't know what the SID of the delta you want is, but you do know the date on (or before) which it was checked in. You can extract the version of the file that was the last one checked in before the given date using the –c (cutoff) option. For example,

```
tutorial% sccs get -c800722120000 prog.c
```

retrieves whatever version was current as of July 22, 1980 at 12:00 noon. Trailing components can be stripped off (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line could be equivalently stated as:

```
tutorial% sccs get -c"80/07/22 12:00:00" prog.c
```

Selectively Deleting Old Deltas

Suppose that you later decided that you liked the changes in delta 1.4, but that delta 1.3 should be removed. You could do this by *excluding* delta 1.3:

```
tutorial% sccs edit -x1.3 prog.c
```

When delta 1.5 is made, it will include the changes made in delta 1.4, but will exclude the changes made in delta 1.3. You can exclude a range of deltas using a dash. For example, if you want to get rid of 1.3 and 1.4 you can use:

```
tutorial% sccs edit -x1.3-1.4 prog.c
```

which will exclude all deltas from 1.3 through 1.4. Alternatively,

```
tutorial% sccs edit -x1.3-1 prog.c
```

will exclude a range of deltas from 1.3 to the current highest delta in release 1.

In certain cases when using –x (or –i — see below) there will be conflicts between versions; for example, it may be necessary to both include and delete a particular line. If this happens, sccs always displays a message telling the range of lines affected; these lines should then be examined very carefully to see if the version sccs got is ok.

Since each delta (in the sense of 'a set of changes') can be excluded at will, it is most useful to put each semantically distinct change into its own delta.

7.6. Auditing Changes

Displaying Delta Comments with sccs prt

When you created a delta, you presumably gave a reason for the delta to the 'comments?' prompt. To display these comments later, use:

```
tutorial% sccs prt prog.c
```

which produces a report for each delta of the SID, time and date of creation, user who created the delta, number of lines inserted, deleted, and unchanged, and the comments associated with the delta. For example, the output of the above command might be:

```
tutorial% sccs prt prog.c
D 1.2   80/08/29 12:35:31       bill    2       1       00005/00003/00084
removed "-q" option

D 1.1   79/02/05 00:19:31       eric    1       0       00087/00000/00000
date and time created 80/06/10 00:19:31 by eric
```

Finding Why Lines Were Inserted

To find out why you inserted lines, you can get a copy of the file with each line preceded by the SID that created it:

```
tutorial% sccs get -m prog.c
```

You can then find out what changes were made by this delta by printing the comments using *prt*.

To find out what lines are associated with a particular delta, 1.3 for instance, use:

```
tutorial% sccs get -m -p prog.c | grep '^1.3'
```

The –p option makes sccs output the generated source to the standard output rather than to a file.

Discovering What Changes You Have Made with sccs diffs

When you are editing a file, you can find out what changes you have made using:

```
tutorial% sccs diffs prog.c
```

Most of the options to diff can be used. To pass the –c option to diff, however, use –C. You can also use the -r and -c options to compare the version being edited with an earlier checked-in version.

To compare two checked-in versions, use:

```
tutorial% sccs sccsdiff -r1.3 -r1.6 prog.c
```

to see the differences between delta 1.3 and delta 1.6. Again, most options to diff can be used, as can the -c option of sccs; for the -c diff option, use -C.

7.7. Shorthand Notations

There are several sequences of commands that are used frequently. sccs tries to make it easy to do these.

Making a Delta and Getting a File with sccs delget

A frequent requirement is to make a delta of some file and then get that file. This is done by using

```
tutorial% sccs delget prog.c
```

which is entirely equivalent to:

```
tutorial% sccs delta prog.c
tutorial% sccs get prog.c
```

except that if an error occurs while making a delta of *any* of the files, none of them will be gotten. The sccs deledit command is equivalent to sccs delget except that the sccs edit command is used instead of the sccs get command; this is useful for checking in a set of changes while you continue editing.

Replacing a Delta with the sccs fix

Frequently, there are small bugs in deltas, for instance, compilation errors, for which there is no reason to maintain an audit trail. To *replace* a delta, use:

```
tutorial% sccs fix -r1.4 prog.c
```

This gets a copy of delta 1.4 of prog.c for you to edit and then deletes delta 1.4 from the SCCS-file. When you do a delta of prog.c, it will be delta 1.4 again. The `-r` option must be specified, and the delta that is specified must be a leaf delta, that is, no other deltas may have been made subsequent to the creation of that delta.

Backing Out of an Edit with `sccs unedit`

If you found you edited a file that you did not want to edit, you can back out by using:

```
tutorial% sccs unedit prog.c
```

Working From Other Directories

If you are working on a project where the history files are in another directory, you may be able to simplify things by making a symbolic link to the true `SCCS` subdirectory:

```
tutorial% ln -s /usr/src/cmd/SCCS SCCS
```

With this method, you can get a separate set of source files in a location that is more convenient. While in the working directory, you can also check files in and out—just as you could if you were in the original directory from which the history files were created.

To extract a complete set of duplicate sources, use the command `sccs get SCCS`.

7.8. Using SCCS on a Project

Working on a project with several people has its own set of special problems. The main problem occurs when two people attempt to modify a file at the same time. `sccs` prevents this by locking an s.file while it is being edited.

As a result, you should not check files out unless you are actually making changes to them, since this prevents other people making needed changes. For example, a good scenario for working might be:

```
tutorial% sccs edit a.c g.c t.c
tutorial% vi a.c g.c t.c
# do testing of the (experimental) version
tutorial% sccs delget a.c g.c t.c
tutorial% sccs info
# should respond "Nothing being edited"
tutorial% make install
```

As a general rule, all source files should be checked in before installing the program for general use. This will ensure that it is possible to restore any version in use at any time.

7.9. Saving Yourself

Recovering a Corrupted Edit File

Sometimes you may find that you have destroyed or trashed a file that you were trying to edit[7]. Unfortunately, you can't just remove it and re-`sccs edit` it; `sccs` keeps track of the fact that someone is trying to edit it, so it won't let you do it again. Simply using `sccs get`, would expand the ID keywords, and besides, if there are edited portions edited file that you want to preserve, you don't want to overwrite it. Instead, you can get a writable copy of the file (with unexpanded keywords) using the `-k` and `-G`*filename* options in combination. The `-k` option tells `sccs` to get a writable version. The `-G`*filename* tells it to place the copy in the named file:

```
tutorial% sccs get -k -G/tmp/prog.c prog.c
tutorial% ls -l Prog.c prog.c
-rw-r--r--  1 user       42652 May 20 17:21 /tmp/prog.c
-rw-r--r--  1 user       43654 May 19 17:40 prog.c
```

From here, you can use `diff` and your favorite editor to selectively restore the changes you wish to keep. Of course, if you just want to start over, you can simply `sccs unedit`, and then `sccs edit` the file once again.

Restoring the History File

In particularly bad circumstances, the history file itself may get corrupted. The most common way this happens is for someone to edit it. Since the file contains a checksum, you will get errors every time you read a corrupted file. To correct the checksum, use:

```
tutorial% sccs admin -z prog.c
```

CAUTION **When `sccs` says that the history file is corrupted, it may indicate serious damage beyond an incorrect checksum. Be careful to safeguard your current changes before attempting to correct a history file.**

7.10. Managing SCCS-Files with `sccs admin`

There are a number of parameters that can be set using the *admin* command. The most interesting of these are flags. Flags can be added by using the `–f` option. For example:

```
tutorial% sccs admin -fd1 prog.c
```

sets the 'd' flag to the value '1'. This flag can be deleted by using:

```
tutorial% sccs admin -dd prog.c
```

The most useful flags are:

`b` Allow branches to be made using the `-b` option to `sccs edit`.

[7] Or given up and decided to start over.

d*SID*

Default SID to be used on a `sccs get` or `sccs edit`. If this is just a release number it constrains the version to a particular release only.

`i` Give a fatal error if there are no ID keywords in a file. This is useful to guarantee that a version of the file does not get merged into the s.file that has the ID keywords inserted as constants instead of internal forms.

`y` The 'type' of the module. Actually, the value of this flag is unused by `sccs`, except that it replaces the `%Y%` keyword.

−t*file*

store descriptive text from *file* in the SCCS-file. This descriptive text might be the documentation or a design and implementation document. Using the −t option ensures that if the SCCS-file is passed on to someone else, the documentation will go along with it. If *file* is omitted, the descriptive text is deleted. To see the descriptive text, use `prt -t`.

The *admin* command can be used safely any number of times on files. A file need not be gotten for *admin* to work.

7.11. Maintaining Different Versions (Branches)

Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production. This can be done using a 'branch'. Normally deltas continue in a straight line, each depending on the delta before. Creating a branch 'forks off' a version of the program.

The ability to create branches must be enabled in advance using:

```
tutorial% sccs admin -fb prog.c
```

The −fb option can be specified when the SCCS-file is first created.

Creating a Branch

To create a branch, use:

```
tutorial% sccs edit -b prog.c
```

This will create a branch with (for example) SID 1.5.1.1. The deltas for this version will be numbered 1.5.1.*n*.

Getting From a Branch

Deltas in a branch are normally not included when you do a get. To get these versions, you will have to say:

```
tutorial% sccs get -r1.5.1 prog.c
```

Merging a Branch Back into the Main Trunk

At some point you will have finished the experiment, and if it was successful you will want to incorporate it into the released version. But in the meantime someone may have created a delta 1.6 that you don't want to lose. The commands:

```
tutorial% sccs edit -i1.5.1.1-1.5.1 prog.c
tutorial% sccs delta prog.c
```

will merge all of your changes into the release system. If some of the changes conflict, `delta` will print an error. The generated result should be carefully examined before the delta is made.

A More Detailed Example

The following technique might be used to maintain a different version of a program. First, create a directory to contain the new version:

```
tutorial% mkdir ../newxyz
tutorial% cd ../newxyz
```

Edit a copy of the program on a branch:

```
tutorial% sccs -d../xyz edit -b prog.c
```

When using the old version, be sure to use the –b option to *info*, *check*, *tell*, and *clean* to avoid confusion. For example, use:

```
tutorial% sccs info -b
```

when in the 'xyz' directory.

If you want to save a copy of the program (still on the branch) back in the s.file, you can use:

```
tutorial% sccs -d../xyz deledit prog.c
```

which will do a delta on the branch and reedit it for you.

When the experiment is complete, merge it back into the s.file using delta:

```
tutorial% sccs -d../xyz delta prog.c
```

At this point you must decide whether this version should be merged back into the trunk, that is, the default version, which may have undergone changes. If so, it can be merged using the `-i` option to `sccs edit` as described above.

A Warning

Branches should be kept to a minimum. After the first branch from the trunk, SID's are assigned rather haphazardly, and the structure gets complex fast.

7.12. SCCS Quick Reference

Commands

This list is not exhaustive; for more options see Appendix A of this manual.

`sccs get`

Gets files for compilation (not for editing). ID keywords are expanded.

`-r`*SID*	Version to get.
`-p`	Send to standard output rather than to the actual file.
`-k`	Don't expand ID keywords.
`-G`*filename*	Get to a named file.
`-i`*list*	List of deltas to include.
`-x`*list*	List of deltas to exclude.
`-m`	Precede each line with SID of creating delta.
`-c`*date*	Don't apply any deltas created after *date*.

`sccs edit`

Gets files for editing. ID keywords are not expanded. Should be matched with a `delta` command.

`-r`*SID*	Same as for `sccs get`. If *SID* specifies a release that does not yet exist, the highest numbered delta is retrieved and the new delta is numbered with *SID*.
`-b`	Create a branch.
`-i`*list*	Same as for `sccs get`.
`-x`*list*	Same as for `sccs get`.

`sccs delta`

Merge a file gotten using `edit` back into the s.file. Collect comments about why this delta was made.

`sccs unedit`

Remove a file that has been edited previously without merging the changes into the s.file.

`sccs prt`

Produce a report of changes.

`-t`	Print the descriptive text.
`-e`	Print (nearly) everything.

`sccs info`

Give a list of all files being edited.

`-b`	Ignore branches.
`-u[`*user*`]`	Ignore files not being edited by *user*.

`sccs check` Same as `info`, except that nothing is printed if nothing is being edited and exit status is returned.

`sccs tell` Same as `info`, except that one line is produced per file being edited containing only the file name.

`sccs clean` Remove all files that can be regenerated from the s.file.

`sccs what` Find and print ID keywords.

`sccs admin` Create or set parameters on s.files.

`-i`*file* Create, using *file* as the initial contents.

`-z` Rebuild the checksum in case the file has been trashed.

`-f`*flag* Turn on *flag*.

`-d`*flag* Turn off (delete) *flag*.

`-t`*file* Replace the descriptive text in the s.file with the contents of *file*. If *file* is omitted, the text is deleted. Useful for storing documentation or design and implementation documents to ensure they get distributed with the s.file .

Useful flags that can be introduced via the `-F` and `-d` options are:

`b` Allow branches to be made using the `-b` option to `edit`.

`d`*SID* Default SID to be used on a `get` or `edit`.

`i` Make the 'No Id Keywords' error message a fatal error rather than a warning.

`t` The module 'type'; the value of this flag replaces the `%Y%` keyword.

`sccs fix` Remove a delta and reedit it.

`sccs delget` Do a `delta` followed by a `get`.

`sccs deledit` Do a `delta` followed by an `edit`.

`sccs create` Create a history file. Move the original file to a backup file with a comma prefix.

`-b` Force the file to be treated as a binary file.

`sccs diffs` Show line-by-line differences between the edited version and a checked-in version (the most recent by default).

`-r`*SID* Specify a version to compare against.

`sccs sccsdiff` Show line-by-line differences between two checked-in versions.

`-r`*SID* Specify a version to compare against. (You must specify two versions to compare.)

ID Keywords

%Z%	Expands to '@(#)' for the what command to find.
%M%	The current module name, for example, prog.c.
%I%	The highest SID applied.
%W%	A shorthand for %Z%%M% < *tab* > %I%.
%G%	The date of the delta corresponding to the %I% keyword.
%R%	The current release number, that is, the first component of the %I% keyword.
%Y%	Replaced by the value of the t flag (set by admin).

8

make User's Guide

8

make User's Guide

8.1. Overview

This chapter describes Sun's enhanced version of make, which includes new features such as hidden dependency checking, command dependency checking, pattern-matching implicit rules, and automatic extraction of SCCS files. It is highly compatible with makefiles written for previous versions. Makefiles that rely on Sun's enhancements may not be compatible with other versions of make. Refer to *Appendix A* for a complete summary of enhancements.

make streamlines the process of generating and maintaining object files and executable programs. It helps you to compile programs consistently, and eliminates unnecessary compilation of modules that are unaffected by source code changes.

make provides a number of features that simplify compilations, but you can also use it to automate any complicated or repetitive task that isn't interactive. For instance, you can use *make* to update and maintain object libraries, run test suites, and install validated files onto a filesystem or tape. In conjunction with SCCS, you can use make to insure that all programs are built from the most recent source versions. You can also use make and SCCS to build an entire software project, and to maintain the source files and directories from which that project is built.

Consistency Control

The Source Code Control System, or SCCS, provides facilities for version control over source files. These include file locking, audit trails, commentary, and other useful features. Refer to Chapter 7 of this manual for an introduction to SCCS.

make provides facilities for consistency control over the object files or other files derived from those sources. It rebuilds the files, in a modular and consistent fashion, when the source files they derive from have changed.

make reads a file that you create, called a *makefile*, which contains information about what files to build and how to build them. Once you write and test the makefile, you can forget about the processing details; make takes care of them. This gives you more time to concentrate on debugging and correcting your code; the repetitive portion of the maintenance cycle is reduced to:

think — edit — **make** — test . . .

Dependency Checking: make vs. Shell Scripts

While it is possible to use a shell script to assure consistency in trivial cases, scripts are often inadequate in actual practice. On the one hand, you don't want to wait for a simple-minded script to compile every single program or object module when only one of them has changed. On the other hand, having to edit the script for each iteration can defeat the objective of consistent compilation.

Although it is possible to write a script of sufficient complexity to process only those modules that require it, such scripts can often develop maintenance problems of their own. In any case, make eliminates the need for you to do so.

make assumes that only it will make changes to files being processed during the current make run. If a source file changes in the middle of the run, the files make produces may be in an inconsistent state.

make allows you to write a simple, structured listing of what to build and how to build it. It uses the mechanism of *dependency checking* to compare each module with the source files or intermediate files it derives from. make only rebuilds a module if one or more of these prerequisite files, called *dependency files*, has changed since the module was last built. To determine whether a derived file is out of date with respect to its sources, make compares the modification time of the module with that of the source file. If the module is missing, or if it is older than the source file, it is considered to be out of date; make issues the commands necessary to rebuild it. Optionally, a target can be treated as out of date if the commands used to build it have changed.

Because make does a complete dependency scan, changes to a source file are consistently propagated through any number of intermediate files or processing steps. This lets you specify a hierarchy of processing steps in a top-down fashion.

make Basics

You can think of a makefile as a type of recipe. make reads the recipe, decides which steps need to be performed, and executes only those steps that are required to produce the finished product. Each file to build, or step to perform, is called a *target*. The makefile entry for a target contains its name, and a list of commands for building it called a *rule*, along with a list of dependencies. make treats dependencies as prerequisite targets, and updates them if necessary, before processing the target that depends on them.

The file for which the target is named is also referred to as a *target file*. Each file from which a target is derived (or that the target depends on) is called a *dependency file* with respect to that target.

Basic Use of Implicit Rules

In addition to any makefile(s) that you supply, make reads in the default makefile, /usr/include/make/default.mk, which contains target entries for *implicit rules*, as well as other information.[8] When there is no target entry in the makefile for a specified target, make attempts to select an implicit rule for building it. When it finds a rule for the target's class, it applies the commands listed in the implicit rule's target entry to build the specific target.

There are two types of implicit rules. ''Suffix'' rules specify a set of command for building a file with one suffix from another file with the same basename but a different suffix. ''Pattern-matching'' rules select a rule based on a target and dependency pair matching a certain wild-card pattern. The default set of implicit rules provided by make are of the former type, namely, suffix rules.

[8] Implicit rules were hard-coded in earlier versions of make.

In some cases, the use of suffix rules can eliminate the need for writing a makefile entirely. For instance, to build an object file named go.o from a single C source file named go.c, you could use the command:

```
make go.o
```

as shown:

```
tutorial% make go.o cc   -sun4 -c go.c -o go.o
```

This would work equally well for building the object file nonesuch.o from the source file nonesuch.c.

To build an executable file named go (with a null suffix) from go.c, you need only type the command:

```
make go
```

as shown:

```
tutorial% make go
cc   -sun4  -o go go.c
```

The rule for building a .o file from a .c file is called the .c.o (pronounced "dot-c-dot-o") suffix rule. The rule for building an executable program from a .c file is called the .c (dot-c) rule. The complete set of default suffix rules is listed in Table 3-1.

Writing a Simple Makefile

The basic format for a makefile target entry is:

Figure 8-1 *Makefile Target Entry Format*

If there is no rule for a target entry, make looks for an implicit rule to use.

```
target ... :  [ dependency ... ]
              [ command ]
              ...
```

If the dependency list is terminated with a semicolon and followed by a command, that command is included in the rule. However, makefiles tend to read better if you avoid this.

In the first line, the target name (or list) is followed by a colon, which is required. This, in turn, is followed by the dependency list if there is one. Several target names separated by white space can precede the colon; this indicates a list of independent targets that are built using the same dependency list and rule.

Command lines in a rule start with a TAB; leading spaces are no substitute as far as make is concerned.

Subsequent lines that start with a TAB are taken as the commands lines that comprise the target's rule. make is awfully fussy about those leading TAB's, SPACE characters *simply won't do.*

Lines that start with a # are treated as comments up until the next (unescaped) NEWLINE, and do not terminate the target entry. The target entry is terminated by the next nonempty line that begins with a character other than TAB or #, or by the end of the file.

A trivial makefile might consist of just one target:

Figure 8-2 *A Trivial Makefile*

```
test:
        ls test
        touch test
#       'test' is now present
        ls test
```

The convention is to use the name `Makefile`, since filenames starting with a capital are listed first by `ls`; this highlights the fact that a makefile is present.

When you run `make` with no arguments, it searches first for a file named `makefile`, or if there is no file by that name, `Makefile`. If either of these files is under SCCS control, `make` extracts the current version and uses it.

If `make` finds a makefile, it begins the dependency check with the first target entry in that file. Otherwise you must list the targets to build as arguments on the command line. `make` displays each command it runs while building its targets.

```
tutorial% make
ls test
test not found
touch test
ls test
test
```

Because the file `test` was not present (and therefore out of date), `make` performed the rule in its target entry. If you run `make` a second time, it issues a message indicating that the target is now up to date:

```
tutorial% make
'test' is up to date.
```

and doesn't perform the rule.

`make` invokes a Bourne shell to process a command line if that line contains any shell metacharacters, such as a semicolon (`;`), redirection symbol (<, >, >>, . . .) or pipe symbol (`|`), etc. If a shell isn't required to parse the command line, etc. *make* invokes the command directly for better performance.

Line breaks within a rule are significant in that each command line is performed by a separate process or shell.

This means that a rule such as:

```
test:
        cd /tmp
        pwd
```

behaves differently than you might expect, as shown below.

```
tutorial% make test
cd /tmp
pwd
/usr/tutorial/waite/arcana/minor/pentangles
```

You can use semicolons to specify a sequence of commands to perform in a single shell invocation:

```
test:
        cd /tmp ; pwd
```

Or, you can continue the input line onto the next line in the makefile by escaping the NEWLINE with a backslash (\):

The backslash must be the last character on the line.

```
test:
        cd /tmp ; \
        pwd
```

Here is an example of a simple target entry to compile a C program from a single source file:

Figure 8-3 *Simple Target Entry for Compiling a* C *Program*

This entry performs the same function with respect to go as in the second example of implicit rules shown above; it compiles an executable program from a C source file.

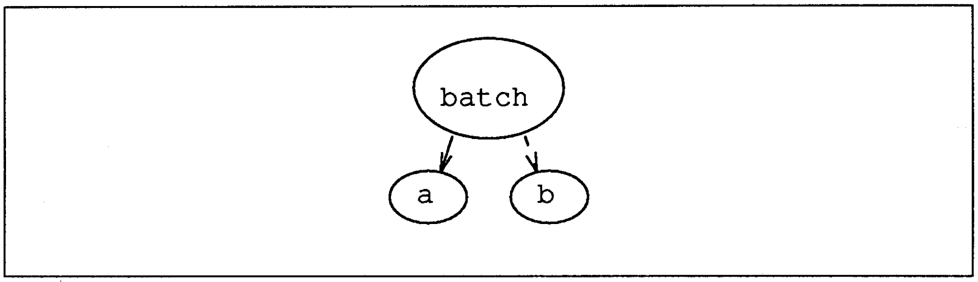
```
go: go.c
        cc -sun4 -o go go.c
```

Processing Dependencies

Once make begins, it processes targets as it encounters them in its depth-first dependency scan. For example, with the following makefile:

```
batch: a b
        touch batch
b:
        touch b
a:
        touch a
c:
        echo "you won't see me"
```

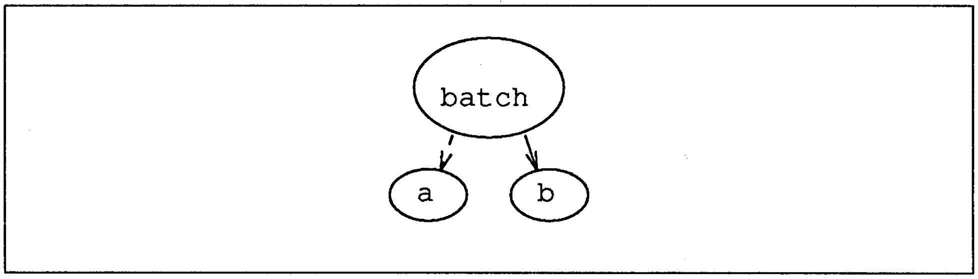
make starts with the target batch. Since batch has some dependencies that haven't been checked yet, namely a and b, make defers checking batch until after it has checked each of them against any dependencies they might have.



Since a has no dependencies, make processes it; if the file is not present make performs its rule.

```
tutorial% make
touch a
. . .
```

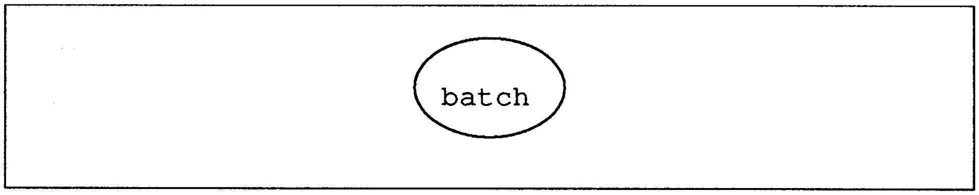
Next, make works its way back up to the parent target batch. Since there is still an unchecked dependency b, make descends to b and checks it.



b also has no dependencies, so make processes it:

```
. . .
touch b
. . .
```

Finally, now that all of the dependencies for batch have been checked and built if needed, make checks it against those dependency files:



Since both a and b were built just now, and are therefore newer than batch, make builds it:

```
. . .
touch batch
```

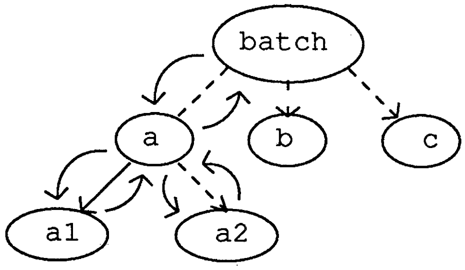
Although there is a target entry for c in the makefile, make does not encounter it while performing its dependency scan. Target entries that aren't encountered in the dependency scan are omitted from processing. You can select a starting target like c by entering it as an argument to the make command:

```
tutorial% make c
echo "you won't see me"
you won't see me
```

In the next example, batch is used to group a set of targets.

```
batch:  a b c

a:    a1  a2
        touch a
b:
        touch b
c:
        touch c
a1:
        touch a1
a2:
        touch a2
```

In this case, the targets are checked and processed as shown in the following diagram:



1. make checks batch, for dependencies and notes that there are three, so it defers processing it.
2. make checks a, the first dependency, and notes that it has two dependencies of its own. So, continuing in the same fashion, make:
   - Checks a1, and if necessary, rebuilds it.
   - Checks a2, and rebuilds it if necessary.
   - Determines whether to build a.
3. make checks b and rebuilds it if need be.

4. Checks and rebuilds c if needed.

5. After processing all of these nested dependencies, make checks and processes the topmost target, batch.

Missing Targets and Dependencies

If a target entry contains no rule, make attempts to select an implicit rule to build it. If make cannot find an appropriate rule to apply and there is no SCCS file to extract it from, make presumes that the target has an *empty* rule, and continues processing subsequent targets. With this makefile:

```
void:
```

make produces:

```
tutorial% make void
tutorial%
```

make stops processing and issues an error message if the target was named either on the command line or in a dependency list but it:

- is missing,
- has no target entry,
- no implicit rule can be used to build it, and
- there is no SCCS file to extract it from.

The following command produces:

```
tutorial% make believe
make: Fatal error: Don't know how to make target 'believe'.
```

On the other hand, if the target entry has no rule, and make encounters the target in a dependency list, it does *not* produce an error, either when processing the dependency, or when processing the target for which it is a dependency. This holds true, even if the dependency file is *absent*.

make *finds* a target entry for the dependency. It *executes* the (null) rule for that dependency without encountering errors. So, make concludes that the dependency has been updated successfully, at the time that the (null) rule is performed. The dependency is therefore considered *newer* than the target, even though no dependency *file* exists. In a case such as this, make simply goes on to rebuild the parent target (after processing any remaining dependencies). With this makefile:

You can use a dependency with a null rule to force the target's rule to be executed. The conventional name for such a dependency is FORCE.

```
haste: FORCE
        echo "haste makes waste"
FORCE:
```

make performs the rule for making haste, even if a file by that name is up to date:

```
tutorial% touch haste
tutorial% make haste
echo "haste makes waste"
haste makes waste
```

Running Commands Silently

You can inhibit the display of a given command line by inserting an @ as the first non-TAB character on that line. For example, the following target:

```
quiet:
        @ echo you only see me once
```

produces:

```
tutorial% make quiet
you only see me once
```

If you want to inhibit the display during a particular make run, you can use the -s option. If you want to inhibit the display of all command lines in every run, add the special target .SILENT to your makefile:

```
.SILENT:
quiet:
        echo you only see me once
```

Special-function targets begin with a dot (.). Target names that begin with a dot are never used as the starting target, unless specifically requested as an argument on the command line.

Ignoring a Command's Exit Status

make normally issues an error message and stops when a command returns a nonzero exit code. For example, if you have the target:

```
rmxyz:
        rm xyz
```

and there is no file named xyz, make halts after rm returns its exit status.

```
tutorial% ls xyz
xyz not found
tutorial% make rmxyz
rm xyz
rm: xyz: No such file or directory
*** Error code 1
make: Fatal error: Command failed for target 'rmxyz'
```

If - and @ are the first two such characters, both take effect.

To continue processing regardless of the command's exit code, use a dash character (-) as the first non-TAB character:

```
rmxyz:
        -rm xyz
```

In this case you get a warning message indicating the exit code make received:

```
tutorial% make rmxyz
rm xyz
rm: xyz: No such file or directory
*** Error code 1 (ignored)
```

Unless you are testing a makefile, it is usually a bad idea to ignore non-zero error codes on a global basis. Specific commands that return non-zero status can be ignored in certain circumstances. But, in general, a non-zero exit code indicates trouble. It is best for make to stop so that you can diagnose the problem right away.

Although it is generally ill-advised to do so, you can have make ignore error codes entirely within a run with the -i option. You can also have make ignore exit codes when processing a given makefile, by adding the special target .IGNORE to your makefile, although this too should normally be avoided.

```
.IGNORE:
rmxyz:
        rm xyz
```

If you are processing a list of targets, and you want make to continue with the next target on the list, rather than stopping entirely after encountering an non-zero return code, use the -k option.

Automatic Extraction of SCCS Files

When source files are named in the dependency list, make treats them just like any other target file. Because the source file is presumed to be present in the directory, there is no need to add an entry for it to the makefile. When a target has no dependencies, but is present in the directory, make assumes that that file is up to date. If, however, a source file is under SCCS control, make does some additional checking to assure that the source file is the version most recently checked in. If the file is missing, or if there is a new version has been checked in, make automatically issues an

"sccs get -s -G*filename filename*"

command to extract the most recent version:[9] If, however, the source file is writable by anyone, make does not extract it.

[9] With other versions of make automatic SCCS extraction was a feature only of certain implicit rules. Also, unlike earlier versions, make only looks for history (s.) files in the SCCS subdirectory; s. files in the current directory are ignored.

```
tutorial% ls SCCS/*
SCCS/s.go.c
tutorial% rm -f go.c
tutorial% make go
sccs get -s go.c -Ggo.c
cc -sun4 -o go go.c
```

This makes it unnecessary to add SCCS commands for extracting current versions of source files; make handles this for you automatically.

Suppressing SCCS Extraction

The command for extracting SCCS files is specified in the rule for the .SCCS_GET special target in the default makefile. To suppress automatic extraction, simply add an entry for this target, without any rule, to your makefile:

```
#   Suppress sccs extraction.
.SCCS_GET:
```

Passing Parameters: Simple make Macros

make's macro substitution comes in handy when you want to pass parameters to commands lines within a makefile. Suppose that you sometimes wish to compile an optimized version of the program go using cc's -O option. You can lend this sort of flexibility to your makefile by adding a *macro reference*, such as the one below, to the target for go:

```
go: go.c
        cc -sun4 $(CFLAGS) -o go go.c
```

The macro reference acts as a placeholder for a value that you define, either in the makefile itself, or as an argument to the make command. If you then supply make with a *definition* for the CFLAGS macro, make replaces the macro reference with the value you have defined.

There is a reference to the CFLAGS macro in both the .c and the .c.o implicit rules.

```
tutorial% rm go
tutorial% make go "CFLAGS= -O"
cc -sun4 -O -o go go.c
```

If a macro is undefined, make replaces references to it with an empty string:

```
tutorial% rm go
tutorial% make go
cc -sun4 -o go go.c
```

You can also include macro definitions in the makefile itself. A typical use is to set CFLAGS to -O so that make produces optimized object code by default, as shown below.

```
CFLAGS= -O
go: go.c
        cc -sun4 $(CFLAGS) -o go go.c
```

With no arguments, the make command produces:

```
tutorial make
cc -sun4 -O -o go go.c
```

A macro definition supplied as an argument to make overrides all other definitions for that macro found in that make run. For instance, to compile go for debugging with dbx or dbxtool, you can define the value of CFLAGS to be -g in the make command:

```
tutorial% rm go
tutorial% make CFLAGS=-g
cc -sun4 -g -o go go.c
```

To compile a profiling version for use with gprof, supply both -O and -pg in the value for CFLAGS:

```
tutorial% rm go
tutorial% make "CFLAGS= -O -pg"
cc -sun4 -O -pg -o go go.c
```

A macro reference must include parentheses when the name of the macro is longer than one character. If the macro name is only one character, the parentheses can be omitted. Also, you can use curly braces, { and }, instead of parentheses. For example:

```
S= echo now and forever
.SILENT:
when:
        $S
        $(S)
        ${S}
```

are all three equivalent:

```
tutorial% make when
now and forever
now and forever
now and forever
```

Command Dependency Checking and .KEEP_STATE

In addition to the normal dependency checking, you can use the special target .KEEP_STATE to activate *command dependency* checking.[10] When activated, make not only checks each target file against its dependency files, it compares each command line in the rule with the corresponding command line it ran the last time it built the target. (This information is stored in a state file in the current directory.) If the command line has changed, make rebuilds the target. So, if .KEEP_STATE were in effect for the previous few examples, you wouldn't have had to type in all those rm go commands.

With the makefile:

```
CFLAGS= -O
.KEEP_STATE:
go: go.c
        cc -sun4 -o go go.c
```

the following commands work as shown:

```
tutorial% make
cc -sun4 -O -o go go.c
tutorial% make CFLAGS=-g
cc -sun4 -g -o go go.c
tutorial% make "CFLAGS= -O -pg"
cc -sun4 -O -pg -o go go.c
```

This assures you that make compiles a program with the options you want, even if a different variant of the file is present and newer than its dependencies.

The first make run with .KEEP_STATE in effect recompiles all targets. This insures that they have, in fact, been built by the command line reported in the state file.

Suppressing or Forcing Command Dependency Checking for Selected Lines

To inhibit command dependency checking for a given command line, insert a question mark as the first character after the TAB. For instance, without the question mark, this makefile:

```
ARG= redone or not
.KEEP_STATE:
x:
        echo $(ARG) | tee x
```

reprocesses x when you define ARG on the command line, as shown below.

[10] This feature is not available in earlier versions of make.

```
tutorial% make x
echo redone or not | tee x
redone or not
tutorial% make x "ARG= redone this time"
echo  redone this time | tee x
redone this time
```

Adding a ? as the first character after the TAB suppresses command dependency checking.

```
ARG= is it redone
.KEEP_STATE:
x:
        ? echo $(ARG) | tee x
```

With it, x is not reprocessed as a result of changing ARG, as shown:

```
tutorial% make x
echo is it redone | tee x
is it redone
tutorial% make x "ARG= still not redone"
'x' is up to date.
```

Command dependency checking is automatically suppressed for lines containing the dynamic macro $?, This macro stands for the list of dependencies that are newer than the current target, and can be expected to differ between any two make runs. (See *Implicit Rules and Dynamic Macros* for more information.) To force make to perform command dependency checking on a line containing this macro, prefix the command line with a ! character (following the TAB).

The State File

When the .KEEP_STATE special target is in effect, make writes out a state file named .make.state, in the current directory. This file lists all targets that have ever been processed while .KEEP_STATE has been in effect, in a format similar to a makefile. In order to assure that this state file is maintained consistently, once you have added the .KEEP_STATE special target to a makefile, we recommend that you leave it in effect.[11]

Hidden Dependencies and .KEEP_STATE

When a source file contains #include directives for interpolating header files, the target depends just as much on those header files as it does on the sources that include them. Because such header files may not be listed explicitly as sources in the compilation command line, they are called *hidden dependencies*. When .KEEP_STATE is in effect, make receives a report from the various compilers and compilation preprocessors indicating which hidden dependency files were

[11] Since this target is ignored in earlier versions of make, it does not introduce any compatibility problems. Other versions simply treat it as a superfluous target that no targets depend on, with an empty rule and no dependencies of its own. Since it starts with a dot, it is not used as the starting target.

interpolated for each target.[12] It adds this information to the dependency list in the state file. In subsequent runs, these additional dependencies are processed just like regular dependencies. This feature maintains the hidden dependency list for each target automatically; this insures that the dependency list for each target is always accurate and up to date. It also eliminates the need for the complicated schemes found in some earlier makefiles to generate complete dependency lists.

A slight inconvenience can arise the first time make processes a target with hidden dependencies, because there is as yet no record of them in the state file. If a header file is missing, and make has no record of it, make won't know that it needs to extract it from SCCS, before compiling the target. So, even though there is an SCCS history file, the current version won't be extracted because it doesn't yet appear in a dependency list or the state file. So, when the C preprocessor attempts to interpolate the header, it won't find it; the compilation fails.

Supposing that an #include directive for interpolating the header file hidden.h is added to go.c, and that the file hidden.h is somehow removed before the subsequent make run. The results would be:

```
tutorial% make go
cc -sun4 -O -o go go.c
go.c: 2: Can't find include file hidden.h
make: Fatal error: Command failed for target 'go'
. . .
```

The workaround is simple. Just make sure that the new header file is present in the directory before you run make. Or, if the compilation should fail (and assuming the header file is under SCCS), extract it from SCCS manually:

```
tutorial% sccs get hidden.h
1.1
10 lines
tutorial% make go
cc -sun4 -O -o go go,c
```

In future cases, should the header file turn up missing, make will know to build or extract it for you, because it will be listed in the state file as a hidden dependency:

```
tutorial% rm go hidden.h
tutorial% make go
sccs get -s hidden.h -Ghidden.h
cc -sun4 -O -o go go.c
```

Note that with hidden dependency checking, the $? macro includes the names of hidden dependency files. This may cause unexpected behavior in existing makefiles that rely on $?.

[12] Also unavailable with earlier versions of make.

Displaying Information About a make Run

Running make with the -n option displays the commands make is to perform, without executing them. This comes in handy when verifying that the macros in a makefile are expanded as expected. With the following makefile:

There is an exception to this however. make executes any command line containing a reference to the MAKE macro (i.e., $(MAKE) or ${MAKE}), regardless of -n. So, it would be a very bad idea to include a line like: "$(MAKE) ; rm -f *" in your makefile.

```
CFLAGS= -O
CPPFLAGS=
LDFLAGS=

.KEEP_STATE:

program: main.o data.o
        $(LINK.c) -o program main.o data.o
```

make -n displays:

```
tutorial% make -n
cc -O -sun4 -c main.c
cc -O sun4 -c data.c
cc -O  -sun4 -o program main.o data.o
```

make has some other options that you can use to keep abreast of what it's doing and why:

Setting an environment variable named MAKEFLAGS can lead to complications, since make adds its value to the list of options. To prevent puzzling surprises, avoid setting this variable.

-d Displays the criteria by which make determines that a target is be out-of-date. Unlike -n, it *does* process targets, as shown below. This options also displays the value imported from the environment (null by default) for the MAKEFLAGS macro, which is described in detail in a later section.

```
tutorial% make -d
MAKEFLAGS value:
   Building main.o using suffix rule for .c.o because it is out of date relative to main.c
cc -O  -sun4 -c  main.c
  Building program because it is out of date relative to main.o
   Building data.o using suffix rule for .c.o because it is out of date relative to data.c
cc -O  -sun4 -c  data.c
  Building program because it is out of date relative to data.o
cc -O   -sun4 -o program main.o data.o
```

This option displays all dependencies make checks in vast detail.

-D Displays the text of the makefile as it is read.

-DD Displays the makefile and the default makefile, the state file, and hidden dependency reports for the current make run.

Several -f options indicate the concatenation of the named makefiles.

-f *makefile*
make uses the named *makefile* (instead of makefile or Makefile).

-p Displays the complete set of macro definitions and target entries.

-P Displays the complete dependency tree for each target encountered.

Due to its potentially troublesome side effects, we recommend against using the -t (touch) option for make.

There is an option that can be used to shortcut make processing, the -t option. When run with -t, make does not perform the rule for building a target. Instead it uses touch to alter the modification time for each target that it encounters in the dependency scan. It also updates the state file to show reflect what it built. This often creates more problems than it supposedly solves, and so we recommend that you exercise extreme caution if you do use it. Note that if there is no file corresponding to a target entry touch creates it.

The following is one example of how *not* to use make -t. Suppose you have a target named clean that performed housekeeping in the directory by removing target files produced by make:

```
clean:
        rm program main.o data.o
```

If you give the erroneous command:

```
tutorial% make -t clean
touch clean
tutorial% make clean
'clean' is up to date.
```

you then have to remove the file clean before your housekeeping target can work once again.

For a complete listing of all make options, refer to make(1) in the *SunOS Reference Manual.*

8.2. Compiling Programs with make

Compilation Strategies

In previous examples you have seen how to compile a simple C program from a single source file, using both explicit target entries and implicit rules. Most C programs, however, are compiled from several source files. Many include library routines, either from one of the standard system libraries or from a local library. Although it may be easier to recompile and link a single-source program using a single cc command, it is usually more convenient to compile programs with multiple sources in stages—first, by compiling each source file into a separate object (.o) file, and then by linking the object files to form an executable program (an a.out format file). This method requires more disk space, but subsequent (repetitive) recompilations need be performed only on those object files for which the sources have changed. The time saved is usually worth the extra space required, since the remaining, up-to-date, object files are simply relinked as is into a newly produced executable program.

The makefile that follows compiles an executable program from two C source files. In subsequent examples, this makefile will be refined and enhanced to take advantage of make's predefined macros and implicit rules. Subsequent sections describe the mechanics of implicit rules, including how to add new ones of your own.

Then, additional features are introduced that are useful in makefiles for maintaining C object libraries. Later sections expand upon these examples to create sophisticated templates that are easily modified to handle a variety of programs or libraries.[13]

Further examples illustrate template makefiles for more complex operations, such as linking programs with with user-supplied object libraries (from other directories), linking C programs with assembly language routines, and compiling programs from lex and yacc sources.

A Simple Makefile

The makefile below is not very flexible or elegant, but it does the job.

Figure 8-4 *Simple Makefile for Compiling* C *Sources: Everything Explicit*

```
#   Simple makefile for compiling a program from
#   two C source files.

.KEEP_STATE:

program: main.o data.o
        cc -sun4 -o program main.o data.o

main.o: main.c
        cc -sun4 -O -c main.c

data.o: data.c
        cc -sun4 -O -c data.c

clean:
        rm program main.o data.o
```

In this example, the command:

make

produces the object files main.o and data.o, and the executable file program.

[13] Makefiles for programs and libraries written in other compiled languages, such as FORTRAN 77, Pascal, and Modula-2, are analogous.

Conventions have evolved for the use of certain target names, such as `all`, `clean` and `install`, among others. There may be other conventions in your organization. In general, it is a good idea to avoid creating files by any such name in your source directories.

The last target, `clean`, removes these files. This is a common addition to simplify housekeeping chores. The name `clean` is a convention for targets that removes derived files.

Using `make`'s Predefined Macros

The next example performs exactly the same function, but demonstrates the use of `make`'s predefined macros for the indicated compilation commands. Using predefined macros eliminates the need to edit makefiles when the underlying compilation environment changes. They also provide access to the `CFLAGS` macro (and other `FLAGS` macros) for supplying compiler options from the command line. Predefined macros are also used extensively within `make`'s implicit rules. The predefined macros in the following makefile are listed below.[14] They are generally useful for compiling C programs.

Macro names that end in the string `FLAGS` are used to pass options to a related compiler-command macro. It is good practice to use these macros for consistency and portability. It is also good practice to note the desired default values for them in the makefile.

The complete list of all predefined macros is shown in Table 1.2, below.

`COMPILE.c` The complete `cc` command line; composed of the values of `CC`, `CFLAGS`, `CPPFLAGS`, and `TARGET_ARCH`, as follows, along with the `-c` option.

```
COMPILE.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
```

The root of the macro name, `COMPILE`, is a convention used to indicate that the macro stands for an entire compilation command line. The `.c` suffix is a mnemonic device to indicate that the command line applies to `.c` (C source) files.

`LINK.c` The complete `cc` command line to link object files, like `COMPILE.c`, but without the `-c` option and with a reference to the `LDFLAGS` macro:

```
LINK.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
```

`CC` The value `cc`. (You can redefine the value to be the pathname of an alternate C compiler.)

`CFLAGS` Options for the `cc` command; none by default.

`CPPFLAGS` Options for `cpp`; none by default.

`LDFLAGS` Options for the link editor, `ld`; none by default.

[14] Predefined macros are used more extensively than in earlier versions of `make`. Not all of the predefined macros shown here are available with earlier versions.

`AR` The `ar` command, which is used for maintaining library archives.

`ARFLAGS` Flags for `ar`. The default value is

```
rv
```

`TARGET_ARCH` The target-architecture argument to `cc` used for cross-compiling. The default is set by `make` to the value returned by the `arch` command.

`TARGET_MACH` The target machine-type argument to `cc` that is used for cross-compiling. The default is set by `make` to the value returned by the `mach` command. Refer to *Cross-Compilation on the Sun Workstation* for details.

Figure 8-5 *Makefile for Compiling* C *Sources Using Predefined Macros*

```
#   Makefile for compiling two C sources
#   using predefined macros.

CFLAGS= -O
CPPFLAGS=
LDFLAGS=

.KEEP_STATE:

program: main.o data.o
        $(LINK.c) -o program main.o data.o

main.o: main.c
        $(COMPILE.c) main.c

data.o: data.c
        $(COMPILE.c) data.c

clean:
        rm program main.o data.o
```

Using Implicit Rules to Simplify a Makefile: Suffix Rules

Since the command lines for compiling `main.o` and `data.o` from their respective `.c` files are now functionally equivalent to the `.c.o` suffix rule, their target entries are, in a sense, redundant; `make` performs the same compilation whether they appear in the makefile or not. This next version of the makefile eliminates them, relying on the `.c.o` rule to compile the individual object files.

Figure 8-6 *Makefile for Compiling* C *Sources Using Suffix Rules*

```
#   Makefile for a program from two C sources
#   using suffix rules.

CFLAGS= -O
CPPFLAGS=
LDFLAGS=

.KEEP_STATE:

program: main.o data.o
        $(LINK.c) -o program main.o data.o

clean:
        rm program main.o data.o
```

A complete list of suffix rules appears in Table 3-1.

As make processes the dependencies main.o and data.o, it finds no target entries for them. So, it checks for an appropriate implicit rule to apply. In this case, make selects the .c.o rule for building a .o file from a dependency file that has the same basename and a .c suffix.

make uses the order of appearance in the suffixes list to determine which dependency file and suffix rule to use. For instance, if there were both main.c and main.s files in the directory, make would use the .c.o rule, since .c is ahead of .s in the list.

First, make scans its suffixes list to see if the suffix for the target file appears. In the case of main.o, the string .o appears in the list. Next, make checks for an suffix rule to build it with, and a dependency file to build it from. The dependency file has the same basename as the target, but a different suffix. In this case, while checking the .c.o rule, make finds a dependency file named main.c, so it selects that rule. The target entry for the suffix rule is named for the dependency suffix and the target suffix; the name is composed of the two suffixes, in this case the target name becomes .c.o, make applies the rule given in the target entry by that name (in the default makefile).

The suffixes list is a special-function target named .SUFFIXES. The various suffixes are included in the definition for the SUFFIXES macro; the dependency list for .SUFFIXES is given as a reference to this macro:

Figure 8-7 *The Standard Suffixes List*

```
SUFFIXES= .o .c .c~ .s .s~ .S .S~ .ln .f .f~ .F .F~ .l .l~ \
        .mod .mod~ .sym .def .def~ .p .p~ .r .r~ .y .y~ .h .h~ .sh .sh~

.SUFFIXES:  $(SUFFIXES)
```

The following example shows a makefile for compiling a whole set of executable programs, each having just one source file. Each executable is to be built from a source file that has the same basename, and the .c suffix appended. For instance demo_1 is built from demo_1.c.

Like clean, all is a target name used by convention. It builds "all" the targets in its dependency list. Normally, all is the first target; make and make all are usually equivalent.

```
#   Makefile for a set of C programs, one source
#   per program.  The source file names have ".c"
#   appended.

CFLAGS= -O
CPPFLAGS=
LDFLAGS=

.KEEP_STATE:

all: demo_1 demo_2 demo_3 demo_4 demo_5
```

In this case, make does not find a suffix match for any of the targets (demo_1 through demo_5). So, it treats each as if it had a null suffix. It then searches for an suffix rule and dependency file with a valid suffix. In the case of demo_2, it would find a file named demo_2.c. Since there is a target entry for a .c(null) rule, namely the .c rule, along with a corresponding .c file make uses the rule in the .c target entry to build demo_2 from demo_2.c.

There is no transitive closure for suffix rules. If you had a suffix rule for building, say, a .y file from a .x file, and another for building a .z file from a .y file, make would not combine their rules to build a .z file from a .x file. You must specify the intermediate steps as targets, as in the next example.

```
tutorial% ls mcp.[xyz]
mcp.x
tutorial% make mcp.z
Don't know how to make mcp.z.  Stop.
tutorial% make mcp.y mcp.z
cp mcp.x mcp.y
cp mcp.y mcp.z
```

When to Use Explicit Target Entries vs. Implicit Rules

Whenever you build a target from multiple dependency files, you must provide make with an explicit target entry that contains a rule for doing so. When building a target from a single dependency file, it is often convenient to use an implicit rule.

As the previous examples show, make is happy to compile a single source file into a corresponding object file or executable. However, it has no built-in knowledge whatsoever about how to collate several files into one. For instance, it has no idea of the order in which to link a list of object files into an executable program. Also, make only compiles those object files that it encounters in its dependency scan. It needs a starting point—a target for which each object file in the list (and ultimately, each source file) is a dependency.

So, for a target built from multiple dependency files, make needs an explicit rule that provides a collating order, and a dependency list that accounts for all of its dependency files. On the other hand, if each of those dependency files is built from just one source, you could use an implicit rule to build them.

Implicit Rules and Dynamic Macros

make maintains a set of macros dynamically, on a target-by-target basis. These macros are used quite extensively, especially in the definitions of implicit rules. So, it is important to understand what they mean.

Because they aren't explicitly defined in a makefile, the convention is to document dynamic macros with the $-sign prefix attached (in other words, by showing the macro reference).

They are:

`$@` The name of the current target.

`$?` The list of dependencies newer than the target.

`$<` The name of the dependency file, as if selected by make for use with an implicit rule.

`$*` The basename of the current target (the target name stripped of its suffix).

`$%` For libraries, the name of the member being processed. See *Building Object Libraries,* below, for more information.

Implicit rules make use of these dynamic macros in order to supply the name of a target or dependency file to a command line within the rule itself. For instance, in the `.c.o` rule, shown in the next example.

```
   .c.o:
           $(COMPILE.c) $< $(OUTPUT_OPTION)
```

The macro `OUTPUT_OPTION` has an empty value by default. While similar to `CFLAGS` in function, it is provided as a separate macro, intended for passing in the `-o` *filename* compiler option, as needed, to force compiler output to a given filename.

`$<` is replaced by the name of the dependency file (in this case the `.c` file) for the current target.

In the `.c` rule:

```
   .c:
           $(LINK.c) $< -o $@
```

`$@` is replaced with the name of the current target.

Because values for the `$<` and `$*` macros depend upon both the order of suffixes in the suffixes list, you may get surprising results when you use them in an explicit target entry. See *Suffix Replacement in Macro References* for a strictly deterministic method for deriving a filename from a related filename.

Dynamic Macro Modifiers

Dynamic macros can be modified by including `F` and `D` in the reference. If the target being processed is in the form of a pathname, `$(@F)` indicates the filename part, while `$(@D)` indicates the directory part. If there are no `/` characters in the target name, then `$(@D)` is assigned the dot character `(.)` as its value. For example, with the target named `/tmp/test`, `$(@D)` has the value `/tmp`; `$(@F)` has the value `test`.

Dynamic Macros and the Dependency List: Delayed Macro References

Dynamic macros are assigned while processing any and all targets. They can be used within the target's rule as is, or in the dependency list by prepending an additional `$` character to the reference. A reference beginning with `$$` is called a *delayed* reference to a macro. For instance, the entry:

```
x.o y.o z.o: $$@.BAK
        cp $@.BAK $@
```

could be used to copy `x.o` from a backup copy named `x.o.BAK`, and so forth for `y.o` and `z.o`.

How `make` Evaluates Dependencies

This technique works because `make` reads the dependency list twice, once as it starts up, and again as it encounters each target while following the dependency scan. Each time it does so, it resolves any macro references contained in the dependency list. Before processing any dependencies, the dynamic macros aren't defined. Unless the references are delayed until the second pass, `make` would resolve them to an empty value. The string `$$` is a reference to the predefined macro '`$`'. This macro, conveniently enough, has the value '`$`'; when `make` resolves it in the first (parsing) pass, the string `$$*` is resolved to `$*`. Then, in the second pass, the `$*` macro reference has a value dynamically assigned to it, so `make` resolves the reference to that value.

Note that `make` only evaluate the target-name portion of a target entry in the first pass. A delayed macro reference as a target name will produce incorrect results. The makefile:

```
NONE= none
all: $(NONE)

$$(NONE):
        @: this target's name isn't 'none'
```

produces:

```
tutorial% make
make: Fatal error: Don't know how to make target 'none'
```

However, the `$$` notation can be used, as described under *Delayed References to a Shell Variable* below, to pass a shell variable reference to the shell interpreting the command line.

Also note that `make` evaluates the rule portion of a target entry only once, at the time that the rule is executed. Here again, a delayed reference to a `make` macro will produce incorrect results.

Adding Suffix Rules

Pattern matching rules, which are described in the previous section, are often easier to use than. The procedure for adding implicit rules is given here for compatibility with previous versions of make.

Although make supplies you with a number of useful suffix rules, you can also add new ones of your own design. However, pattern matching rules,[15] which are described in the next section, are to be preferred when adding new implicit rules. Unless you need to write implicit rules that are compatible with earlier versions of make, you may safely skip the remainder of this section, which describes the traditional method of adding implicit rules to makefiles.

Adding a suffix rule is a two-step process. First, you must add the suffixes of both target and dependency file to the suffixes list by providing them as dependencies to the .SUFFIXES special target. Because dependency lists accumulate, you can add suffixes to the list simply by adding another entry for this target, for example:

```
.SUFFIXES: .ms .tr
```

Second, you must add a target entry for the suffix rule:

```
.ms.tr:
        troff -t -ms $< > $@
```

A makefile with these entries can be used to format document source files containing ms macros (.ms files) into troff output files (.tr files):

```
tutorial% make doc.tr
troff -t -ms doc.ms > doc.tr
```

Entries in the suffixes list are contained in the SUFFIXES[16] macro. To insert suffixes at the head of the list, first clear its value by supplying an entry for the .SUFFIXES target that has no dependencies. This is an exception to the rule that dependency lists accumulate. You can clear a previous definition for any target with a name starting with the character '.' by supplying a target entry for that target with no dependencies and no rule,[17] like this:

```
.SUFFIXES:
```

When you do, both the previous rule, and the previous dependency list are erased. You can then add another entry containing the new suffixes, followed by a reference to the SUFFIXES macro, as shown below.

[15] Not available with earlier versions of make.

[16] Note that there is no leading dot.

[17] You can only clear the dependency list for the .SUFFIXES target in previous versions of make.

```
.SUFFIXES:
.SUFFIXES: .ms .tr $(SUFFIXES)
```

Pattern Matching Rules: an Alternative to Suffix Rules

A *pattern matching rule* is similar to an implicit rule in function. Pattern matching rules are easier to write, and more powerful, because you can specify a relationship between a target and a dependency based on prefixes and suffixes both. A pattern matching rule is a target entry of the form:

tp%*ts*: *dp*%*ds*
        rule

where *tp* and *ts* are the optional prefix and suffix in the target name, respectively, *dp* and *ds* are the (optional) prefix and suffix in the dependency name, and % is a wild card that stands for a basename common to both.

make checks for pattern matching rules ahead of suffix rules. While this allows you to override the standard implicit rules, doing so is not recommended.

If there is no rule for building a target, make searches for a pattern matching rule, *before* checking for a suffix rule. If make can use a pattern matching rule, it does so.

If the target pattern matches the target name, there is a dependency file matching the dependency pattern, and the target is out of date with respect to that dependency file, make rebuilds the target. If the target is up to date with respect to the dependency, make does not rebuild it, and continues processing with the next target in the dependency hierarchy.

If the target entry for a pattern matching rule contains no rule, make processes the target file as if it had an explicit target entry with no rule; it therefore searches for a suffix rule, attempts to extract a version of the target file from SCCS, and finally, treats the target as having a null rule (flagging the target as updated, which forces any parent target to be rebuilt).

A pattern matching rule for formatting a troff source file into a troff output file looks like:

```
%.tr: %.ms
        troff -t -ms $< > $@
```

This is much easier to write, and much simpler to follow than the equivalent suffix rule would be.

make's Default Suffix Rules and Predefined Macros

The following tables list the standard set of suffix rules and predefined macros supplied with make.

Table 8-1 make's *Standard Suffix Rules*

Use	*Suffix Rule Name*	*Command Line(s)*
Assembly *Files*	.s.o	$(COMPILE.s) –o $@ $<
	.s.a	$(COMPILE.s) –o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $%
	.S.o	$(COMPILE.S) –o $@ $<
	.S.a	$(COMPILE.S) –o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $%
C *Files*	.c	$(LINK.c) –o $@ $< $(LDLIBS)
	.c.ln	$(LINT.c) $(OUTPUT_OPTION) –i $<
	.c.o	$(COMPILE.c) $(OUTPUT_OPTION) $<
	.c.a	$(COMPILE.c) –o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $%
FORTRAN 77 *Files*	.f	$(LINK.f) –o $@ $< $(LDLIBS)
	.f.o	$(COMPILE.f) $(OUTPUT_OPTION) $<
	.f.a	$(COMPILE.f) –o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $%
	.F	$(LINK.F) –o $@ $< $(LDLIBS)
	.F.o	$(COMPILE.F) $(OUTPUT_OPTION) $<
	.F.a	$(COMPILE.F) –o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $%
lex *Files*	.l	$(RM) $*.c $(LEX.l) $< > $*.c $(LINK.c) –o $@ $*.c $(LDLIBS) $(RM) $*.c
	.l.c	$(RM) $@ $(LEX.l) $< > $@
	.l.ln	$(RM) $*.c $(LEX.l) $< > $*.c $(LINT.c) –o $@ –i $*.c $(RM) $*.c
	.l.o	$(RM) $*.c $(LEX.l) $< > $*.c $(COMPILE.c) –o $@ $*.c $(RM) $*.c
Modula 2 *Files*	.mod .mod.o .def.sym	$(COMPILE.mod) –o $@ –e $@ $< $(COMPILE.mod) –o $@ $< $(COMPILE.def) –o $@ $<
NeWS	.cps.h	cps $*.cps
Pascal *Files*	.p	$(LINK.p) –o $@ $< $(LDLIBS)
	.p.o	$(COMPILE.p) $(OUTPUT_OPTION) $<
Ratfor *Files*	.r	$(LINK.r) –o $@ $< $(LDLIBS)
	.r.o	$(COMPILE.r) $(OUTPUT_OPTION) $<
	.r.a	$(COMPILE.r) –o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $%

Table 8-1 make's *Standard Suffix Rules— Continued*

Use	*Suffix Rule Name*	*Command Line(s)*
Shell Scripts	.sh	cat $< >$@ chmod +x $@
yacc *Files*	.y	$(YACC.y) $< $(LINK.c) -o $@ y.tab.c $(LDLIBS) $(RM) y.tab.c
	.y.c	$(YACC.y) $< mv y.tab.c $@
	.y.ln	$(YACC.y) $< $(LINT.c) -o $@ -i y.tab.c $(RM) y.tab.c
	.y.o	$(YACC.y) $< $(COMPILE.c) -o $@ y.tab.c $(RM) y.tab.c

Table 8-2 make's *Predefined and Dynamic Macros*

Use	*Macro*	*Default Value*
Library Archives	AR	ar
	ARFLAGS	rv
Assembler Commands	AS	as
	ASFLAGS	
	COMPILE.s	$(AS) $(ASFLAGS) $(TARGET_ARCH)
	COMPILE.S	$(CC) $(ASFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
C Compiler Commands	CC	cc
	CFLAGS	
	CPPFLAGS	
	COMPILE.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
FORTRAN 77 Compiler Commands	FC	f77
	FFLAGS	
	COMPILE.f	$(FC) $(FFLAGS) $(TARGET_ARCH) -c
	LINK.f	$(FC) $(FFLAGS) $(TARGET_ARCH) $(LDFLAGS)
	COMPILE.F	$(FC) $(FFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.F	$(FC) $(FFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
Link Editor Command	LD	ld
	LDFLAGS	
lex *Command*	LEX	lex
	LFLAGS	
	LEX.l	$(LEX) $(LFLAGS) -t
lint *Command*	LINT	lint
	LINTFLAGS	
	LINT.c	$(LINT) $(LINTFLAGS) $(CPPFLAGS) $(TARGET_ARCH)
Modula 2 Commands	M2C	m2c
	M2FLAGS	
	MODFLAGS	
	DEFFLAGS	
	COMPILE.def	$(M2C) $(M2FLAGS) $(DEFFLAGS) $(TARGET_ARCH)
	COMPILE.mod	$(M2C) $(M2FLAGS) $(MODFLAGS) $(TARGET_ARCH)
Pascal Compiler Commands	PC	pc
	PFLAGS	
	COMPILE.p	$(PC) $(PFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.p	$(PC) $(PFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
Ratfor Compilation Commands	RFLAGS	
	COMPILE.r	$(FC) $(FFLAGS) $(RFLAGS) $(TARGET_ARCH) -c
	LINK.r	$(FC) $(FFLAGS) $(RFLAGS) $(TARGET_ARCH) $(LDFLAGS)

Table 8-2 make*'s Predefined and Dynamic Macros— Continued*

Use	*Macro*	*Default Value*
rm *Command*	RM	rm -f
yacc *Command*	YACC YFLAGS YACC.y	yacc $(YACC) $(YFLAGS)
Suffixes *List*	SUFFIXES	.o .c .c~ .s .s~ .S .S~ .ln .f .f~ .F .F~ .l .l~ .mod .mod~ .sym .def .def~ .p .p~ .r .r~ .y .y~ .h .h~ .sh .sh~ .cps .cps~

8.3. Building Object Libraries

Libraries, Members and Symbols

An object library is a set of object files contained in an ar library archive.[18]

Various languages make use of object libraries to store compiled functions of general utility, such as those in the C library.

ar reads in a set of one or more files to create a library. Each member contains the text of one file, preceded by a header. This header contains information taken from the file's directory entry when the text is read in, including the modification time. make can treat the library member as a separate entity for dependency checking using this header.

When you compile a program that uses functions from an object library (specifying the proper library either by filename, or with the -l option to cc), the link editor selects and links with the library member that contains a needed function or symbol.

You can use ranlib to generate a symbol table for a library of object files. ld uses this table for random access to symbols within the library—to locate and link object files in which functions are defined. You can also use lorder and tsort ahead of time to put members in calling order within the library. (See lorder(1) for details.) For very large libraries, it is a good idea to do both.

Library Members and Dependency Checking

make recognizes a target or dependency of the form

lib.a (*member* . . .)

as a reference to a library member, or a space-separated list of members.[19] For example, the following target entry indicates that the library named librpn.a is built from members named stacks.o and fifos.o. The pattern matching rule indicates that each member depends on a corresponding object file, and that object file is built from its corresponding source file using an implicit rule.

[18] See ar(1), ar(5), lorder(1), and ranlib(1) in the *Commands Reference Manual* for details about library archive files.

[19] Earlier versions make recognize this notation. However, only the first item in a parenthesized list of members was processed. In this version of make, all members in a parenthesized list are processed.

```
librpn.a: librpn.a(stacks.o fifos.o)
        ar rv $@ $?
        ranlib $@

lib.a(%.o): %.o
```

When used with library-member notation, the dynamic macro `$?` contains the list of files that are newer than their corresponding members:

```
tutorial% make
cc   -sun4 -c stacks.c
cc   -sun4 -c fifos.c
ar rv librpn.a stacks.o fifos.o
a - stacks.o
a - fifos.o
ranlib librpn.a
```

Library Member Name-Length Limit

The name of an `ar` library member cannot exceed 15 characters. If a filename is longer than that, `ar` truncates the name of its corresponding member to the first 15 characters. If a library depends upon a member whose corresponding filename is too long, `make` attempts to match the name of the member to the first 15 characters of a file in the directory. *make* uses the first filename that matches as the file from which to build the member.

`.PRECIOUS`: Preserving Libraries Against Removal Due to Interrupts

Normally, if you interrupt `make` in the middle of a target, the target file is removed. For individual files this is a good thing, otherwise incomplete files with brand new modification times might be left in the directory. For libraries, which consist of several members, the story is different. It is often better to leave the library intact, even if one of the members is still out of date. This is especially true for large libraries, especially since a subsequent `make` run will pick up where the previous one left off—by processing the object file or member whose processing was interrupted.

`.PRECIOUS` is a special target that is used to indicate which files should be preserved against removal on interrupts; `make` does not remove targets that are listed as its dependencies. If you add the line:

```
.PRECIOUS: librpn.a
```

to the makefile shown above, run `make`, and interrupt the processing of `librpn.a`, the library is preserved.

The `$%` dynamic macro is provided specifically for use with libraries. When a library member is the target, the member name is assigned to the `$%` macro. For instance, this makefile below produces the results that follow.

```
libx.a(demo.o):
        @echo $%
```

```
tutorial% make
demo.o
```

8.4. Maintaining Programs and Libraries With make

In previous sections you have learned how make can help compile simple programs and build simple libraries. The focus of this section is on developing makefiles for more complex compilations. To eliminate possible sources of confusion, it is often a good idea to put each module into a separate directory of its own. This makes clear which source files pertain to which programs or libraries, and allows you to create makefiles that operate consistently between various parts of a software project. Subsequent sections describe how to maintain, as a single entity, a project that spans several directories.

Using Macros for Added Flexibility

You have seen how to use predefined and dynamic macros within rules, and for passing parameters from the command line. make also allows you to define your own macros within a makefile. Macros allow you to simplify makefiles while making them more flexible (for use with other modules, or other projects; makefiles for this version of make are not necessarily portable to other versions of Withmake). use of macros, you can develop *template* makefiles that can be re-used, with only minor edits, for any number of similar compilation procedures. The examples to follow illustrate how to use macros to develop template makefiles for C programs and libraries.

Macro definitions can appear on any line in a makefile; macros can be used to abbreviate long target lists or expressions, or as shorthand to replace long strings that would otherwise have to be repeated. Macro names are allocated as the makefile is read in; the value a particular macro reference takes depends upon the most recent value assigned.[20] For instance, in the following makefile, the macro TEST evaluates to false.

```
TEST= true
TEST= false

all:
        #echo $(TEST)
```

[20] Actually, macro evaluation is a bit more complicated than this. Refer to *Passing Parameters to Nested* make *Commands* for more information.

Embedded Macro References

Macro references can embedded within other references, like this:[21]

```
$(OUTER$(INNER))
```

In which case they are expanded from innermost to outermost:

```
OUTER= out
INNNER= in
outin= something completely different

all:
        @echo $(OUTER$(INNER))
```

produces:

```
tutorial% make
something completely different
```

A More Flexible Makefile

The makefile for compiling a C program that used implicit rules can be generalized to accommodate other programs using macros. By replacing key words with macros, and by editing the definitions of those macros, altering the makefile for use with yet another program becomes a simple matter.

```
#       Flexible makefile for a C program.

SOURCES= main.c data.c
OBJECTS= main.o data.o
PROGRAM= program

CFLAGS= -O CPPFLAGS= LDFLAGS=

.KEEP_STATE:

$(PROGRAM): $(OBJECTS) $(LINK.c) -o $@ $(OBJECTS)

clean:  rm $(PROGRAM) $(OBJECTS)
```

In this case, you need only edit the SOURCES, OBJECTS and PROGRAM macros and you can compile a different program entirely, albeit in the same way.

Although in a simple case like this the changes to the makefile might not seem worth the extra trouble, the added flexibility becomes increasingly important as you apply more powerful techniques. With judicious use of macros you can avoid having to puzzle over which specific changes you can, or should (or even dare), add to a makefile.

[21] Not supported in previous versions of make.

Makefiles as Specifications

A makefile performs an important function by documenting what files get built from which sources, and what compilation options are used, by default, to build them. Specifying this information with a set of macro definitions at the top of the makefile is a great aid the reader, especially when makefiles are similar in format, or at all complicated.

No one should have to scan an entire makefile just to puzzle out what it builds.

Suffix Replacement in Macro References

In the flexible makefile shown above, the value of `OBJECTS` is a bit redundant. It would be better to derive the names of the object files from the names of the source files. In fact, there are any number of filenames that can be derived from the names of source files, simply by altering their suffix. For this reason, `make` provides a mechanism for temporarily replacing suffixes of words in a macro's value, when the reference to that macro is of the form:[22]

`$(`*macro*`:`*old-suffix*`=`*new-suffix*`)`

This *suffix replacement* macro reference allows you to express the list of object files in terms of the list of sources:

```
OBJECTS= $(SOURCES:.c=.o)
```

It replaces all occurrences of the `.c` suffix in words within the value with the `.o` suffix. The substitution is not applied to words for that do not end in the suffix given. The following makefile:

```
OLD= main.c data.c moon
NEW= $(OLD:.c=.o)
all:
        @echo $(NEW)
```

illustrates this very simply:

```
tutorial% make
main.o data.o moon
```

Using `lint` with `make`

`lint`, the C program verifier,[23] is an important tool for forestalling the kinds of bugs that are most difficult and tedious to track down. These include uninitialized pointers, parameter-count mismatches in function calls, and nonportable uses of C constructs. As with the `clean` target, `lint` is a target name used by convention; it is usually a good practice to include it in makefiles that build C

We encourage you to `lint` your C programs for easier debugging and maintenance. `lint` also checks for C constructs that are not considered portable across machine architectures. It can be a real help in writing portable C programs.

[22] Although conventional suffixes start with dots, a suffix may consist of any string of characters.

[23] See *Using* `lint` in `lint` — a Program Verifier for C for more information.

programs. `lint` produces output files that have been preprocessed through `cpp` and its own first (parsing) pass. These files characteristically end in the `.ln` suffix,[24] and can also be derived from the list of sources through suffix replacement:

```
LINTFILES= $(SOURCES:.c=.ln)
```

The `lint` target entry appears as follows:

```
lint: $(LINTFILES)

$(LINTFILES):
        $(LINT.c) $(LINTFILES)
```

There is an implicit rule for building each `.ln` file from its corresponding `.c` file, so there is no need for target entries for the `.ln` files. As sources change, the `.ln` files are updated whenever you run

`make lint`

Since the `LINT.c` predefined macro includes a reference to the `LINTFLAGS` macro, it is a good idea to specify the `lint` options to use by default (none in this case). Since `lint` entails the use of `cpp`, it is a good idea to use `CPPFLAGS`, rather than `CFLAGS` for compilation preprocessing options (such as `-I`). The `LINT.c` macro does not include a reference to `CFLAGS`.

Also, when you run `make clean` you will want to get rid of any `.ln` files produced by this target. It is a simple enough matter to add another such macro reference to the `clean` target:

```
clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

With these changes, the new version of the makefile appears as follows.

[24] This is true for the Sun implementation, it may not be true for other versions of `lint`.

Figure 8-8 *Makefile with ''Suffix-Replacement'' Macro References*

```
#   Makefile for a C program with an entry for lint.
SOURCES= main.c data.c
PROGRAM= program
CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=
OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)
.KEEP_STATE:
$(PROGRAM): $(OBJECTS)
        $(LINK.c) -o $@ $(OBJECTS)
lint: $(LINTFILES)
$(LINTFILES):
        $(LINT.c) $(LINTFILES)
clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

Linking With System-Supplied Libraries

This makefile is easily altered to compile a program that uses system-supplied library packages. The next example shows a makefile that compiles a program that uses the `curses` and `termlib` library packages for screen-oriented cursor motion.

You can also link with a library by specifying its pathname name as an argument to `cc`.

A makefile link with user-supplied libraries appears later on.

Figure 8-9 *Makefile for a C Program With System-Supplied Libraries*

```
#   @(#) sample.1.mk
#
#   Makefile for a C program with curses and termlib.

SOURCES= main.c data.c
LIBS= -lcurses -ltermlib
PROGRAM= program

CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)

.KEEP_STATE:

$(PROGRAM): $(OBJECTS)
        $(LINK.c) -o $@ $(OBJECTS) $(LIBS)

lint: $(LINTFILES)

$(LINTFILES):
        $(LINT.c) $(LINTFILES)

clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

Since the link editor resolves undefined symbols as they are encountered, it is normally a good idea to place library references at the end of the list of files to link.

This makefile produces:

```
tutorial% make
cc -O  -sun4 -c main.c
cc -O  -sun4 -c data.c
cc -O  -sun4 -o program main.o data.o -lcurses -ltermlib
```

Compiling Programs for Debugging and Profiling

Compiling programs for debugging or profiling introduces a new twist to the procedure, and to the makefile. These variants are produced from the same source code, but are built with different options to the C compiler. The `cc` option to produce object code that is suitable for debugging is `-g`, and it is important to omit the `-O` option in this case. The `cc` options that produce code for profiling are `-O` and `-pg`.

Since the compilation procedure is the same otherwise, you *could* give make a definition for CFLAGS on the command line. Since this definition overrides the definition in the makefile, and .KEEP_STATE assures any command lines affected by the change are performed, the command:

```
make "CFLAGS= -O -pg"
```

produces the following results.

```
tutorial% make "CFLAGS= -O -pg"
cc  -O -pg  -sun4 -c main.c
cc  -O -pg  -sun4 -c data.c
cc  -O -pg  -sun4 -o program main.o data.o -lcurses -ltermlib
```

Of course, you may not want to memorize these options or type a complicated command like this, especially when you can put this information in the makefile. What is needed is a way to tell make how to produce a debugging or profiling variant, and some instructions in the makefile that tell it how. One way to do this might be to add two new target entries, one named debug, and the other named profile, with the proper compiler options hard-coded into the command line.

A better way would be to add these targets, but rather than hard-coding their rules, include instructions to alter the definition of CFLAGS depending upon which target it starts with. Then, by making each one depend on the existing target for program make could simply make use of its rule, along with the specified options.

Instead of saying "**make "CFLAGS= -g**", you could say "**make debug**" to compile a variant for debugging. The question is, how do you tell make that you want a macro defined one way for one target (and its dependencies), and another way for a different target?

Conditional Macro Definitions

A conditional macro definition[25]

is a line of the form:

target-list := *macro* = *value*

make must know which targets the definition applies to, so you can't use a conditional macro definition to alter a target name.

which assigns the given *value* to the indicated *macro* while make is processing the target named *target-name* and its dependencies. The following lines give CFLAGS an appropriate value for processing each program variant.

```
debug := CFLAGS= -g
profile := CFLAGS= -pg -O
```

[25] Not available with previous versions of make.

Note that when you use a reference to a condition macro in the dependency list that reference must be delayed (by prepending a second $). Otherwise, make may expand the reference before the correct value has been assigned. When it encounters a (possibly) incorrect reference of this sort, make issues a warning.

Compiling Debugging and Profiling Variants

The following makefile produces optimized, debugging, or profiling variants of a C program, depending on which target you specify (the default is the optimized variant). Command dependency checking guarantees that the program and its object files will be recompiled whenever you switch between variants.

Figure 8-10 *Makefile for a* C *Program with Alternate Debugging and Profiling Variants*

```
#   @(#) sample.2.mk
#
#   Makefile for a C program with alternate
#   debugging and profiling variants.

SOURCES= main.c data.c
LIBS= -lcurses -ltermlib
PROGRAM= program

CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)

.KEEP_STATE:

all debug profile: $(PROGRAM)

debug := CFLAGS= -g
profile := CFLAGS= -pg -O

$(PROGRAM): $(OBJECTS)
        $(LINK.c) -o $@ $(OBJECTS) $(LIBS)

lint: $(LINTFILES)

$(LINTFILES):
        $(LINT.c) $(LINTFILES)

clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

Going through the makefile, all of the lines above .KEEP_STATE seem familiar. The subsequent target entry specifies three targets, with all appearing first.

all is a conventional target for building "all" final, or "finished" targets. Debugging and profiling variants aren't normally considered part of a finished program.

all traditionally appears as the first target in makefiles with alternate starting targets (or those that process a list of targets). It's dependencies are "all" targets that go into the final build, whatever that may be. In this case, the final target is the optimized program variant. This entry also indicates that debug and profile depend on program (the value of $(PROGRAM)).

The next two lines contain conditional macro definitions for CFLAGS, when it appears in profile or debug, or their dependencies:

```
debug := CFLAGS= -g
profile := CFLAGS= -pg -O
```

Next comes the familiar target entry that starts with $(PROGRAM). Finally, the remainder of the makefile looks familiar.

With this makefile,

make

or

make all

produces:

```
tutorial% make
cc -O  -sun4 -c main.c
cc -O  -sun4 -c data.c
cc -O  -sun4 -o program main.o data.o -lcurses -ltermlib
```

make debug

produces:

```
tutorial% make debug
cc -g  -sun4 -c main.c
cc -g  -sun4 -c data.c
cc -g  -sun4 -o program main.o data.o -lcurses -ltermlib
```

and

make profile

produces:

```
tutorial% make profile
cc -pg -O   -sun4 -c main.c
cc -pg -O   -sun4 -c data.c
cc -pg -O   -sun4 -o program main.o data.o -lcurses -ltermlib
```

The next example applies similar techniques to maintaining a C object library.

Figure 8-11 *Makefile for a* C *Library with Alternate Variants*

```
#   @(#) sample.3.mk
#
#   Makefile for a C library with alternate
#   variants.

SOURCES= calc.c map.c draw.c
LIBRARY= libpkg.a

CFLAGS= -O
CPPFLAGS=
LINTFLAGS=

MEMBERS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)

all debug profile: $(LIBRARY)

debug := CFLAGS= -g
profile := CFLAGS= -pg -O

.KEEP_STATE:
.PRECIOUS: $(LIBRARY)

$(LIBRARY): $(LIBRARY)($(MEMBERS))
        ar rv $@ $?
        ranlib $@

$(LIBRARY)(%.o): %.o

lint: $(LINTFILES)

$(LINTFILES):
        $(LINT.c) $(LINTFILES)

clean:
        rm  -f $(LIBRARY) $(MEMBERS) $(LINTFILES)
```

Maintaining Separate Program and Library Variants

The previous two examples are adequate when development, debugging and profiling are done in distinct phases. However they suffer from the drawback that all object files are recompiled whenever you switch between variants, which can result in unnecessary delays. The next two examples illustrate how all three variants can be maintained as separate entities.

To avoid the confusion that might result from having three variants of each object file in the same directory as the program sources, it makes sense to place the debugging and profiling objects and executables in their own subdirectories. However, in order to do this we need a technique for adding a the name of the subdirectory as a prefix to each entry in the list of object files.

Pattern Replacement Macro References

A pattern replacement macro reference is similar in form an function to a suffix replacement reference.[26] You can use a pattern replacement reference to add or alter a prefix, suffix, or both, to matching words in the value of a macro. A pattern replacement reference takes the form:

$ (*macro*:*p* %*s* =*np* %*ns*)

where *p* is the existing prefix to replace (if any), *s* is the existing suffix to replace (if any), *np* and *ns* are the new prefix and suffix, respectively, and % is a wild card character that matches zero or more characters in each word. The pattern replacement is applied to all words in the value that match. For instance, this makefile:

```
OLD= old.main.c old.data.c moon
NEW= $(OLD:old.%.c=new.%.o)

all:
        @echo $(NEW)
```

produces:

```
tutorial% make
new.main.o new.data.o moon
```

Please note, however, that pattern replacement macro references should not appear on the dependency line of a pattern matching rule's target entry. This produces unexpected results. With the makefile:

```
OBJECT= .o

x:
x.Z:
        @echo correct

%: %.$(OBJECT:%o=%Z)
```

it looks as if make should attempt to build x from x.Z. However, the pattern matching rule is not recognized; make cannot determine which of the % characters in the dependency line to use in the pattern matching rule; consequently, the target entry for x.Z is never reached.

```
tutorial% make
tutorial%
```

[26] As with pattern matching rules, pattern matching macro references aren't available in earlier versions of make.

Makefile for a Program with Separate Variants

make performs the rule in the .INIT target just after the makefile is read.

The following example shows a makefile for a C program with separately-maintained variants. First, the .INIT special target, creates the debug and profile subdirectories (if they don't already exist), which will contain the debugging and profiling object files and executables.

Next, the macros DEBUG and PROFILE are assigned the program name, prefixed with either debug/ or profile/, as appropriate. Pattern replacement macro references to the PROGRAM macro are used to accomplish this. Next, the debug and profile targets are set to depend on them so that when you type **make debug**, instead of recompiling program, with different compiler options, make builds the file debug/program.

These variant executables are made to depend on the object files listed in the VARIANTS.o macro. This macro is given the value of OBJECTS by default; later on it may be reassigned using a conditional macro definition, at which time either the debug/ or profile/ prefix is added, as appropriate, to each entry in the list of object files; executables in the subdirectories depend on the object files that are built in those same directories.

Next, pattern matching rules are added to indicate that the object files in both subdirectories depend upon source (.c) files in the working directory. This is the key step needed to allow all three variants to be built and maintained from a single set of source files.

Here is an exception to the advice that a makefile should only maintain files in the current working directory. Still, target files should only be built in a subdirectory if they depend on source files in the working directory.

Finally, the clean target has been updated to recursively remove the debug and profile subdirectories and their contents, which should be regarded as temporary. This helps to impose the practice of keeping all files that are critical to the program in the same directory as its source files, and not in the subdirectories for the variants.

Figure 8-12 *Makefile for Separate Debugging and Profiling Program Variants*

```
#   @(#) sample.4.mk
#
#   Makefile for maintaining separate debugging and
#   profiling program variants.

SOURCES= main.c data.c
LIBS= -lcurses -ltermlib
PROGRAM= program

CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)
DEBUG= $(PROGRAM:%=debug/%)
PROFILE= $(PROGRAM:%=profile/%)
VARIANTS.o= $(OBJECTS)

.KEEP_STATE:
.INIT:
        -mkdir profile debug

all: $(PROGRAM)
debug: $(DEBUG)
profile: $(PROFILE)
variants: debug profile

$(DEBUG) := CFLAGS= -g
$(PROFILE) := CFLAGS= -pg -O
$(DEBUG) := VARIANTS.o= $(OBJECTS:%=debug/%)
$(PROFILE) := VARIANTS.o= $(OBJECTS:%=profile/%)

$(PROGRAM) $(DEBUG) $(PROFILE): $$(VARIANTS.o)
        $(LINK.c) -o $@ $(VARIANTS.o) $(LIBS)

profile/%.o debug/%.o: %.c
        $(COMPILE.c) -o $@ $<

lint: $(LINTFILES)

$(LINTFILES):
        $(LINT.c) $(LINTFILES)

clean:
        rm -rf $(PROGRAM) $(OBJECTS) $(LINTFILES) debug profile
```

Notice that the `all` target has not been made to depend on the debugging and profiling variants. This is because they are not normally part of final production build, so they aren't included in the conventional meaning for `all`. However, if you want to build all three variants it is a simple matter to give the command:

`make all variants`

The modifications for separate library variants are quite similar. First, the new macros `DEBUG` and `PROFILE` are assigned the library name with the proper subdirectory prefix. `VARIANTS.o` is assigned the value of `MEMBERS` by default, and conditionally defined for the debugging and profiling targets. Then, the `.INIT` target is given so that the subdirectories are created (if not already present). Then, the target entry for the library is altered to include all three variants. Next, pattern matching rules are added to specify the dependence of the variant libraries in the respective subdirectories, the variant object files in those same directories. Other pattern matching rules specify the dependence of those object files on source files in the current working directory.

Finally, the `clean` target is modified to recursively remove the variant subdirectories.

Makefile for a Library with Separate Variants

Figure 8-13 *Makefile for Separate Debugging and Profiling Library Variants*

```
#   @(#) sample.5.mk
#
#   Makefile for maintaining separate library
#   variants.

SOURCES= calc.c map.c draw.c
LIBRARY= libpkg.a

CFLAGS= -O
CPPFLAGS=
LINTFLAGS=

MEMBERS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)
DEBUG= $(LIBRARY:%=debug/%)
PROFILE= $(LIBRARY:%=profile/%)
VARIANTS.o= $(MEMBERS)

.KEEP_STATE:
.PRECIOUS: $(LIBRARY)
.INIT:
        -mkdir profile debug

all: $(LIBRARY)
debug: $(DEBUG)
profile: $(PROFILE)
variants: debug profile

debug := CFLAGS= -g
profile := CFLAGS= -pg -O
$(DEBUG) := VARIANTS.o = $(MEMBERS:%=debug/%)
$(PROFILE) := VARIANTS.o = $(MEMBERS:%=profile/%)

$(LIBRARY) $(DEBUG) $(PROFILE): $$(VARIANTS.o)
        ar rv $@ $?
        ranlib $@
        rm -f $?

$(LIBRARY)(%.o): %.o
$(DEBUG)(debug/%.o): debug/%.o
$(PROFILE)(profile/%.o): profile/%.o
profile/%.o debug/%.o: %.c
        $(COMPILE.c) -o $@ $<

lint: $(LINTFILES)

$(LINTFILES):
        $(LINT.c) $(LINTFILES)

clean:
        rm -rf $(LIBRARY) $(MEMBERS) $(LINTFILES) debug profile
```

Here the command:

```
make all variants
```

produces:

```
tutorial% make all variants
cc -O  -sun4 -c  calc.c
cc -O  -sun4 -c  map.c
cc -O  -sun4 -c  draw.c
ar rv libpkg.a calc.o map.o draw.o
a - calc.o
a - map.o
a - draw.o
ar: creating libpkg.a
ranlib libpkg.a
rm -f calc.o map.o draw.o
cc -g  -sun4 -c -o debug/calc.o calc.c
cc -g  -sun4 -c -o debug/map.o map.c
cc -g  -sun4 -c -o debug/draw.o draw.c
ar rv debug/libpkg.a debug/calc.o debug/map.o debug/draw.o
a - debug/calc.o
a - debug/map.o
a - debug/draw.o
ar: creating debug/libpkg.a
ranlib debug/libpkg.a
rm -f debug/calc.o debug/map.o debug/draw.o
cc -pg -O  -sun4 -c -o profile/calc.o calc.c
cc -pg -O  -sun4 -c -o profile/map.o map.c
cc -pg -O  -sun4 -c -o profile/draw.o draw.c
ar rv profile/libpkg.a profile/calc.o profile/map.o profile/draw.o
a - profile/calc.o
a - profile/map.o
a - profile/draw.o
ar: creating profile/libpkg.a
ranlib profile/libpkg.a
rm -f profile/calc.o profile/map.o profile/draw.o
```

While an interesting and useful compilation technique, this method for maintaining separate variants is a bit complicated. For clarity's sake it is omitted from subsequent examples.

Maintaining a Directory of Header Files

The makefile for maintaining an `include` directory of header files is really quite simple. Since header files consist of plain text, all that is needed is a target, `all`, that lists them all as dependencies. Automatic SCCS extraction takes care of the rest. If you use a macro for the list of header files, this same list can be used in other target entries, which may be added later for project management purposes.

```
#   Makefile for maintaining an include directory.
FILES.h= calc.h map.h draw.h
all: $(FILES.h)
clean:
        rm -f $(FILES.h)
```

This same technique can be applied to other files that do not require compilation or other such processing (such as man command document source files).

Compiling and Linking With Your Own Libraries

When preparing your own library packages, it often makes sense to treat each library as a separate entity from programs that use it, as well as the header files used by both. Separating programs, libraries and header files into distinct directories often makes it easier to prepare makefiles for each type of module. And, it clarifies the structure of a software project.

It is not a good idea to have things pop up all over the file system as a result of running make.

A courteous and necessary convention of makefiles is that they only build files in the working directory, or in temporary subdirectories. Unless you are using make specifically to install files into a specific directory on an agreed-upon file system, it is regarded as very poor form for a makefile to produce output in another directory.

Building programs that rely on user-supplied libraries in other directories adds several new wrinkles to the makefile. Up until now, everything needed has been in the directory, or else in one of the standard directories that are presumed to be stable. This is not true for user-supplied libraries that are part of a project under development, especially when their contents are subject to change.

More importantly, since these libraries aren't built automatically (there is no equivalent to automatic SCCS extraction for them), there must be an explicit target entry to build them. So, a problem arises until such time as the library has been completed tested and can be presumed to be stable.

On the one hand, you need to assure the libraries you link with are up to date. On the other hand, you need to observe the convention that a makefile should only maintain files in the local directory. In addition, the makefile should not contain duplicate information that could get out of sync with a makefile in another directory. The whole purpose of make, after all, is to provide consistent, modular processing.

Nested make Commands

The solution is to use a nested make command, running in the directory the library resides in, to rebuild it (according to the target entry in the makefile there).

The MAKE macro, which is set to the value "make" in the default file, overrides the -n option. Any command line in which it is referred to is executed, even though -n may be in effect. Since this macro is used to invoke make, and since the make it invokes inherits -n from the special MAKEFLAGS macro, make can trace a hierarchy of nested make commands with the -n option.

```
#   First cut entry for target in another
#   directory.

../lib/libpkg.a:
        cd ../lib ; $(MAKE) libpkg.a
```

The library is specified with a pathname relative to the current directory. In general, it is better to use relative pathnames. If the project is moved to a new root directory or machine, so long as its structure remains the same relative to that new root directory, all the target entries will still point to the proper files.

Within the nested make command line, the dynamic macro modifiers F and D come in handy, as does the MAKE predefined macro. If the target being processed is in the form of a pathname, $(@F) indicates the filename part, while $(@D) indicates the directory part. If there are no / characters in the target name, then $(@D) is assigned the dot character (.) as its value.

The target entry can be rewritten as:

```
#   Second cut.

../lib/libpkg.a:
        cd $(@D); $(MAKE) $(@F)
```

Forcing A Nested make Command to Run

Because it has no dependencies, this target will only run when the file named ../lib/libpkg.a is missing. If the file is a library archive protected by .PRECIOUS, this could be a rare occurrence. The current make invocation neither knows nor cares about what that file depends on, nor should it. It is the nested invocation that decides whether and how to rebuild that file. After all, just because a file is present in the file system doesn't mean that it is up to date. This means that you have to force the nested make to run, regardless of the file's presence, by making it depend on a target with a null rule:

```
#   Reliable target entry for a nested make
#   command.

../lib/libpkg.a: FORCE
        cd $(@D); $(MAKE) $(@F)
FORCE:
```

In this way, make reliably cd's to the directory ../lib and builds libpkg.a if necessary, using instructions from the makefile found in that directory as../lib), These lines are produced by the nested make run.

```
tutorial% make ../lib/libpkg.a
cd ../lib; make libpkg.a
make libpkg.a
'libpkg.a' is up to date.
```

The following makefile uses a nested make command to process local libraries that a program depends on.

Figure 8-14 *Makefile for* C *Program With User-Supplied Libraries*

```
#   @(#) sample.6.mk
#
#   Makefile for a C program with user-supplied
#   libraries and nested make commands.

SOURCES= main.c data.c
ULIBS= ../lib/libpkg.a
SLIBS= -lcurses -ltermlib
PROGRAM= program

CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)

.KEEP_STATE:

all debug profile: $(PROGRAM)

debug := CFLAGS= -g
profile := CFLAGS= -pg -O

$(PROGRAM): $(OBJECTS) $(ULIBS)
        $(LINK.c) -o $@ $(OBJECTS) $(ULIBS) $(SLIBS)

$(ULIBS): FORCE
        cd $(@D); $(MAKE) $(@F)
FORCE:

lint: $(LINTFILES)

$(LINTFILES):
        $(LINT.c) $(LINTFILES)

clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

When ../lib/libpkg.a is up to date, this makefile produces:

```
tutorial% make
cc -O  -sun4 -c main.c
cc -O  -sun4 -c data.c
cd ../lib; make libpkg.a
'libpkg.a' is up to date.
cc -O  -sun4 -o program main.o data.o ../lib/libpkg.a -lcurses -l termlib
```

The MAKEFLAGS Macro

Do not define MAKEFLAGS in your makefiles.

Like the MAKE macro, MAKEFLAGS is also a special case. As its name suggests, it contains flags (that is, single-character options) for the make command. Unlike other FLAGS macros, the MAKEFLAGS value is a concatenation of flags, without a leading '-'. For instance the string, eiknp would be a recognized value for MAKEFLAGS, while, '-f x.mk' or 'macro=value' would not.

If the MAKEFLAGS environment variable is set, make runs with the combination of flags given on the command line and contained in that variable.

The value of MAKEFLAGS is always exported, whether set in the environment or not, and the options it contains are passed to any nested make commands (whether invoked by $(MAKE), make or /usr/bin/make). This insures you that nested make commands are always passed the options that the parent make was invoked with. Because MAKEFLAGS is maintained automatically, defining it in the makefile would only be misleading.

Macro Definitions and Environment Variables: Passing Parameters to Nested **make** Commands

With the exception of MAKEFLAGS,[27] make imports variables from the environment and treats them as if they were defined macros. In turn, make propagates those environment variables and their values to commands it invokes, including nested make commands. Macros can also be defined as command line arguments, as well as the makefile. This can lead to name-value conflicts when a macro is defined in more than one place, and so, make has a fairly complicated precedence rule for resolving them.

First of all, conditional macro definitions always take effect within the targets (and their dependencies) for which they are defined.

If make is invoked with a macro-definition argument, that definition takes precedence over definitions given either within the makefile, or imported from the environment. (This does not necessarily hold true for nested make commands, however.) Otherwise, if you define (or redefine) a macro within the makefile, the most recent definition applies. The latest definition normally overrides the environment. Lastly, if the macro is defined in the default file and nowhere else, that value is used.

The -e option alters this scheme. With -e, macros defined in the environment override any and all makefile definitions (but not the command line).

[27] and SHELL. The SHELL environment variable is neither imported nor exported in this version of make. See make(1) in the *SunOS Reference Manual*, for more information about the SHELL macro.

With nested make commands, definitions made in the makefile normally override the environment, but only for the makefile in which each definition occurs; the value of the corresponding environment variable is propagated regardless. Command-line definitions override both environment and makefile definitions, but only for the topmost make run. Although values from the command line are propagated to nested make commands, they are overridden both by definitions in the nested makefiles, and by environment variables imported by the nested make commands.

The -e option behaves more consistently. The environment overrides macro definitions made in any makefile, and command-line definitions are always used ahead of definitions in the makefile and the environment. One drawback to -e is that it introduces a situation in which information that is *not contained in the makefile* can be critical to the success or failure of a build.

This is an awful lot to remember, so a good rule of thumb when passing parameters to nested make commands is: supply them as command-line definitions, and use -e. However, before you run make with the -e option, it is important to eliminate all extraneous or improperly defined environment variables, since make -e will propagate whatever is in the environment to the entire hierarchy of nested make commands:

```
make -e CFLAGS=-E
```

Environment variables don't go away when you're done with them (i.e, they stay around to haunt you, especially when you attempt to build something else with make later on). One way to avoid lingering environment variables is to invoke make within a subshell. When you set environment variables and run make in the subshell, their values are isolated within that subshell and any processes it spawns (like the one for make):

```
( setenv CFLAGS -E ; make -e )
```

This next example illustrates the difference in parameters between the top make run and the nested make runs, using the two makefiles shown below.

```
# top.mk

MACRO= "Correct if unexpected."

top:
        @echo "------------------------------ top"
        echo $(MACRO)
        @echo "-------------------------------"
        $(MAKE) -f nested.mk

        @echo "------------------------------ clean"
clean:
        rm nested
```

and:

```
# nested.mk

MACRO=nested

nested:
        @echo "------------------------------ nested"
        touch nested
        echo $(MACRO)
        $(MAKE) -f top.mk
        $(MAKE) -f top.mk clean
```

With these makefiles, the command:

make -f top.mk MACRO=top

produces the results that follow.

```
tutorial% make -f top.mk MACRO=top
------------------------------ top
echo top
top
------------------------------
make -f nested.mk
------------------------------ nested
touch nested
echo nested
nested
make -f top.mk
------------------------------ top
echo "Correct, if unexpected."
Correct, if unexpected.
------------------------------
make -f nested.mk
`nested' is up to date.
make -f top.mk clean
------------------------------ clean
rm nested
```

This pair of makefiles can be helpful if you decide to review the various cases yourself.

Table 8-3 *Summary of Macro Assignment Order*

Without -e	*With* -e *in effect*
top-level make *command:*	
conditional definitions	conditional definitions
make command line	make command line
latest makefile definition	environment value
environment value	latest makefile definition
predefined value, if any	predefined value, if any
nested make *commands:*	
conditional definitions	conditional definitions
make command line	make command line
latest makefile definition	parent make cmd. line
environment variable	environment value
predefined value, if any	latest makefile definition
parent make cmd. line	predefined value, if any

Compiling Other Source Files

The following examples illustrate the use of make to maintain C programs that contain assembly routines, and programs produced with lex and yacc.

Compiling and Linking a C Program with Assembly Language Routines

The makefile in the next example maintains a program with C source files linked with assembly language routines.[28] There are two varieties of assembly source files, those that contain cpp preprocessor directives, and those that don't. By convention, assembly source files without preprocessor directives have the .s suffix. Assembly sources that require preprocessing have the .S suffix.

Assembly sources are assembled to form object files in a fashion similar to that used to compile C sources. The object files can then be linked into a C program. make has implicit rules for transforming .s and .S files into object files, so at a minimum, a target entry for a C program with assembly routines need only specify how to link the object files. You can use the familiar cc command to link object files produced by the assembler:

ASFLAGS passes options for as to the .s.o and .S.o implicit rules.

```
CFLAGS= -O
ASFLAGS= -O

.KEEP_STATE:

driver:  c_driver.o s_routines.o S_routines.o
        cc -o driver c_driver.o s_routines.o S_routines.o
```

The next example shows a more flexible makefile for this sort of compilation.

[28] Refer to the *Assembly Reference Manual* for more information about assembly language source files.

Figure 8-15 *Makefile for a C Program with Assembly Routines*

```
#   @(#) sample.7.mk
#
#   Makefile for a C program linked with assembly routines.

SOURCES.c= c_driver.c
SOURCES.s= s_routines.s
SOURCES.S= S_routines.S
ULIBS=
SLIBS=
PROGRAM= driver

ASFLAGS=
CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=
OBJECTS= $(SOURCES.c:.c=.o) $(SOURCES.s:.s=.o) $(SOURCES.S:.S=.o)
LINTFILES= $(SOURCES.c:.c=.ln)  # not for assembly sources

.KEEP_STATE:

all debug profile: $(PROGRAM)

debug := CFLAGS= -g
profile := CFLAGS= -pg -O

$(PROGRAM): $(OBJECTS) $(ULIBS)
        $(LINK.c) -o $@ $(OBJECTS) $(ULIBS) $(SLIBS)

$(ULIBS): FORCE
        cd $(@D); $(MAKE) $(@F)
FORCE:

lint: $(LINTFILES)

$(LINTFILES):
        $(LINT.c) $(LINTFILES)

clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

This makefile compiles the executable program `driver` as shown:

```
tutorial% make
cc -O  -sun4 -c c_driver.c
as  -sun4 -o s_routines.o s_routines.s
cc   -sun4 -o S_routines.o -c S_routines.S
cc -O  -sun4 -o driver c_driver.o s_routines.o S_routines.o
```

Note that the `.S` files are processed using the `cc` command, which invokes the C preprocessor `cpp`, and invokes the assembler implicitly.

Compiling lex and yacc Sources

lex and yacc produce C source files as output. Source files for lex end in the suffix .l, while those for yacc end in .y. When used separately, the compilation process for each is similar to that used to produce programs from C sources alone. There are implicit rules for compiling the lex or yacc sources into .c files; from there the files are further processed with the implicit rules for compiling object files from C sources. When these source files contain no #include statements, there is no need to keep the c file, which in this simple case serves as an intermediate file. In this case one could use .l.o rule, or the .y.o rule, respectively, to produce the object files, and remove the (derived) .c files. For example, the makefile:

```
CFLAGS= -O
.KEEP_STATE:

all: scanner parser

scanner: scanner.o

parser: parser.o
```

produces:

```
tutorial% make
rm -f scanner.c
lex  -t scanner.l > scanner.c
cc -O  -sun4 -c scanner.c -o scanner.o
rm -f scanner.c
rm -f scanner.c
lex  -t scanner.l > scanner.c
cc -O  -sun4 scanner.c -o scanner
rm -f scanner.c
yacc  parser.y
cc -O  -sun4 -c y.tab.c -o parser.o
rm -f y.tab.c
yacc  parser.y
cc -O  -sun4 y.tab.c -o parser
rm -f y.tab.c
```

Things get to be a bit more complicated when you use lex and yacc in combination. In order for the object files to work together properly, the C code from lex must include a header file produced by yacc. So, it may be necessary to recompile the C source file produced by lex when the yacc source file changes. In this case, it is better to retain the .c (intermediate) files produces by lex, as well as the additional .h file that yacc provides, so as to avoid running lex whenever the yacc source changes.

The following makefile maintains a program built from a lex source, a yacc source, and a C source file.

yacc produces output files named y.tab.c and y.tab.h. If you want the output files to have the same basename as the source file, you must rename them.

```
CFLAGS= -O
.KEEP_STATE:

a2z: c_functions.o scanner.o parser.o
        cc -o $@ c_functions.o scanner.o parser.o

scanner.c:

parser.c + parser.h: parser.y
        yacc -d parser.y
        mv y.tab.c parser.c
        mv y.tab.h parser.h
```

Since there is no transitive closure for implicit rules, you must supply a target entry for scanner.c. This entry bridges the gap between the .l.c implicit rule and the .c.o implicit rule, so that the dependency list for scanner.o extends to scanner.l. Since there is no rule in the target entry, scanner.c is built using the .l.c implicit rule.

The next target entry describes how to produce the yacc intermediate files. Because there is no implicit rule for producing both the header file and the C source file using yacc -d, a target entry must be supplied that includes a rule for doing so.

Specifying Target Groups With the + Sign

In the target entry for parser.c and parser.h, the + sign separating the target names indicates that the entry is for a *target group*.[29] A target group is a set of files, all of which are produced when the rule is performed. Taken as a group, the set of files is what comprises the target. Without the + sign, each item listed would comprise a separate target. With a target group, make checks the modification dates separately against each target file, but performs the target's rule only once, if necessary, per make run.

The next example shows a makefile for the more general case of a lex source, a yacc source, and any number of C source files.

[29] Not available with earlier versions of make.

Figure 8-16 *Makefile for Compiling C Programs With* `lex` *and* `yacc` *Sources*

```
#   @(#) sample.8.mk
#
#   Makefile to compile a C program with lex and yacc sources.

SOURCES.c= c_functions.c
LEXFILE.l= scanner.l
YACCFILE.y= parser.y
ULIBS=
SLIBS=
PROGRAM= a2z

LFLAGS=
YFLAGS=
CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

LEXFILE.c= $(LEXFILE.l:.l=.c)
YACCFILE.c= $(YACCFILE.y:.y=.c)
YACCFILE.h= $(YACCFILE.y:.y=.h)
SOURCES= $(SOURCES.c) $(LEXFILE.c) $(YACCFILE.c)
OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)

.KEEP_STATE:

all debug profile: $(PROGRAM)

debug := CFLAGS= -g
profile := CFLAGS= -pg -O

$(PROGRAM): $(OBJECTS) $(ULIBS)
        $(LINK.c) -o $@ $(OBJECTS) $(ULIBS) $(SLIBS)

$(LEXFILE.c): $(YACCFILE.h)

$(YACCFILE.c) + $(YACCFILE.h): $(YACCFILE.y)
        $(YACC.y) -d $(YACCFILE.y)
        mv y.tab.c $(YACCFILE.c)
        mv y.tab.h $(YACCFILE.h)

$(ULIBS): FORCE
        cd $(@D); $(MAKE) $(@F)
FORCE:

lint: $(LINTFILES)

$(LINTFILES):
        $(LINT.c) $(LINTFILES)

clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

```
tutorial% make all
cc -O  -sun4 -c c_functions.c
yacc  -d parser.y
mv y.tab.c parser.c
mv y.tab.h parser.h
rm -f scanner.c
lex  -t scanner.l > scanner.c
cc -O  -sun4 -c scanner.c
cc -O  -sun4 -c parser.c
cc -O  -sun4   -o a2z c_functions.o scanner.o parser.o
tutorial%
```

Maintaining Shell Scripts with make and SCCS

Although a shell script is a plain text file, it must be executable in order to run. Since SCCS removes execute permission for files under its control and a shell script must have execute permission in order to run, a distinction must be drawn between a shell script and it's ''source'' file under SCCS control. make has an implicit rule for deriving a script from its ''source'' file under SCCS. The suffix for a shell script source file is .sh. Even though the contents of the script and the .sh file are the same, the script has execute permissions, while the .sh file does not. make's implicit rule for scripts ''derives'' the script from its source file, making a copy of the .sh file (extracting it first, if necessary) and changing the mode of the resulting script file to allow execution. For example:

```
tutorial% file script.sh
script.sh:      ascii text
tutorial% make script
cat script.sh > script
chmod +x script
tutorial% file script
script:         commands text
```

Running Tests with make

Shell scripts often come in handy for running tests, and performing other routine tasks that are either interactive, or don't require make's dependency checking. Test suites, in particular, often entail providing a program with specific, repeatable input that a program might expect to receive from a terminal.

In the case of a library, a set of programs that exercise its various functions may be written in C, and then executed in a specific order, with specific inputs from a script. In the case of a utility program, there may be a set of benchmark programs that exercise and time its functions. In each of these cases, the commands to run each test can be incorporated into a shell script for repeatability and easy maintenance.

Once you have developed a test script that suits your needs, including a target to run it is easy. Although make's dependency checking may not be needed within the script itself, you *can* use it to make sure that the program or library is updated before running those tests.

In the following target entry for running tests, test depends on the library named as a dependency to all. If the library is out of date, make rebuilds it and proceeds with the test. This insures that you always test with an up to date version:

```
test: all testscript
        set -x ; testscript > /tmp/test.$$$$

testscript: testscript.sh test_1 test_2 test_3

test_1 test_2 test_3: $$@.c $(LIBRARY)
        $(LINK.c) -o $@ $< $(LIBRARY) $(SLIBS)
```

test also depends on testscript, which in turn depends on the three test programs. This assures that they too are up to date before make initiates the test procedure. all is built according to its target entry in the makefile; testscript is built using the .sh implicit rule; and the test programs are built using the rule in the last target entry, assuming that there is just one source file for each test program. (The .c implicit rule doesn't apply to these programs, because they must link with the proper libraries in addition to their respective .c files).

Delayed References to a Shell Variable

The string $$$$, in the rule for test is, in fact, a pair of references to make's $ macro (each written as $$). make resolves each such reference into a single $, and the command line is passed to the shell as:

```
set -x ; testscript > /usr/tmp/test.$$
```

In this way, the variable reference is delayed from final expansion until it reaches the shell, which interprets it as a reference to $$, the value of which is the process number of the shell. This number is appended to the output filename so that the results of each successive test is written to a unique filename with a standard format. The set -x command forces the shell to display the command on the terminal. This allows you to see the actual filename containing the test results.

This makefile produces:

```
tutorial make
cp testscript.sh testscript
chmod +x testscript
cc   -sun4 -o test_1 test_1.c
cc   -sun4 -o test_2 test_2.c
cc   -sun4 -o test_3 test_3.c
testscript > /tmp/test.$$
+ testscript > /tmp/test.26500
```

A more flexible set of entries for testing a library looks like:

```
TESTSCRIPT= testscript
TESTPROGS= test_1 test_2 test_3

test: all $(TESTSCRIPT)
        set -x ; $(TESTSCRIPT) > /tmp/test.$$$$

$(TESTSCRIPT): $$@.sh $(TESTPROGS)

$(TESTPROGS): $$@.c $(LIBRARY)
        $(LINK.c) -o $@ $< $(LIBRARY) $(SLIBS)
```

In the case of a program, testing routines written in C may not be necessary; leaving TESTPROGS undefined will mean the target entry for test programs is omitted from the dependency scan. TESTSCRIPT depends only upon its corresponding .sh file. If there *are* test programs that don't depend on a library (the LIBRARY macro is undefined) this method is still applicable; it is the equivalent of the .c implicit rule. If, there is a test program that depends on the same libraries as the program does, you can either replace references to the LIBRARY macro with references to ULIBS:

```
$(TESTPROGS): $$@.c $(ULIBS)
        $(LINK.c) -o $@ $< $(ULIBS) $(SLIBS)
```

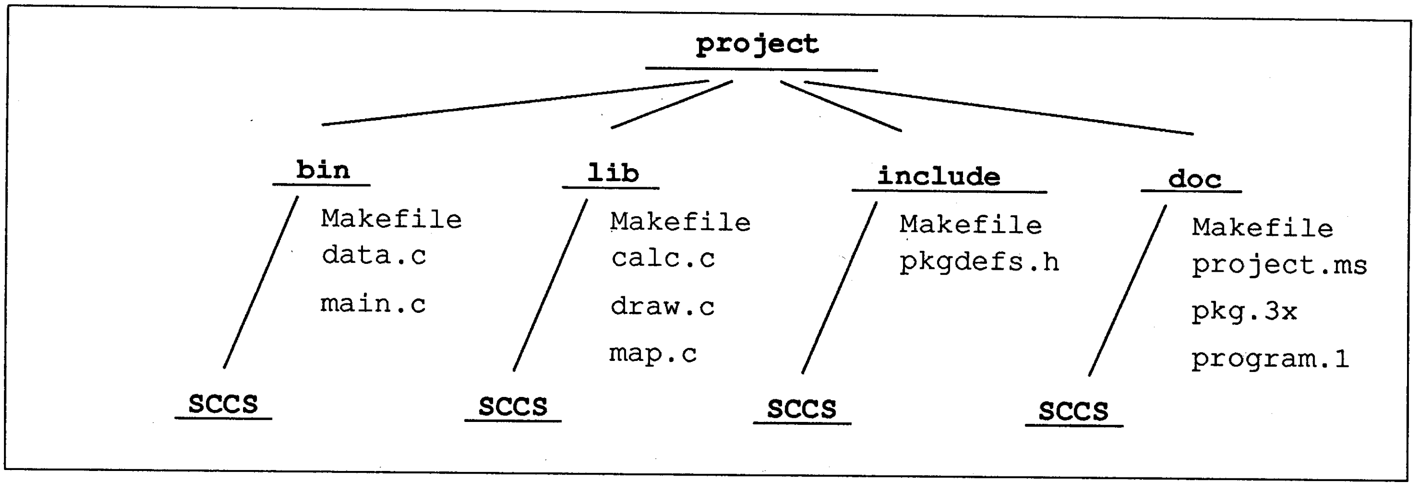
8.5. Maintaining Software Projects

make is especially useful when a software project consists of a system of programs and libraries. By taking advantage of nested make commands, you can use it to maintain object files, executables, and libraries in a whole hierarchy of directories. You can use make in conjunction with SCCS, to assure that sources are maintained in a controlled manner, and that programs built from them are consistent. This means that you can provide other programmers with duplicates of the directory hierarchy for simultaneous development and testing if you wish (although there are tradeoffs to consider).

You can use make to build the entire project and install final copies of various modules onto another filesystem for integration and distribution.

Organizing A Project for Ease of Maintenance

As mentioned earlier, one good way to organize a project is to segregate each major piece into its own directory. A project broken out this way usually resides within a single file-system or directory hierarchy. Header files could reside in one subdirectory, libraries in another, and programs in still another. Documentation, such as Reference Pages, may also be kept on hand in another subdirectory. Suppose that a project is composed of one executable program, one library that you supply, a set of header files for the library routines, and some documentation, structured as shown.



The makefiles in each subdirectory can be borrowed from examples in earlier sections, but something more is needed to manage the project as a whole. A carefully structured makefile in the root directory, the *root makefile* for the project, provides target entries for managing the project as a single entity.

As a project grows, the need for consistent, easy-to-use makefiles also grows. Macros and target names should have the same meanings no matter which makefile you are reading. Conditional macro definitions and compilation options for output variants should be consistent across the entire project.

Where feasible, a *template* approach to writing makefiles makes sense. This makes it easy for you keep track of how the project gets built. All you have to do to add a new type of module is to make a new directory for it, copy an appropriate makefile into that directory, and make a few minor edits to change macro values. (Of course, you also need to add the new module to the list of things to build in the root makefile, but that comes later.)

Although a makefile should document exactly what it builds, it does not necessarily have to contain an explanation of every step. After all, the idea is to spend time working on the code, not the makefiles.

Conventions for macro names, such as those for the various source files in the above examples, should be instituted and observed throughout the project. Mnemonic macro names mean that although you may not remember the exact value of the macro, you'll know the type of value it represents (and that's usually more valuable when deciphering a makefile anyway).

Using include Makefiles

One method of simplifying makefiles, while providing a consistent compilation environment, is to use make's

```
    include filename
```

directive to read in the contents of a named makefile; if the named file is not present, make checks for a file by that name in /usr/include/make.

For instance, there is no need to duplicate the pattern-matching rule for processing troff sources in each makefile, when you can include it's target entry, as shown below.

```
SOURCES= main.c data.c
...
clean: $(PROGRAM) $(OBJECTS) $(LINTFILES)
include ../pm.rules.mk
```

Here, make reads in the contents of the ../pm.rules.mk file, shown here:

```
#   pm.rules.mk
#
#   Simple "include" makefile for pattern matching
#   rules.

%.tr: %.ms
        troff -t -ms $< > $@
%.nr: %.ms
        nroff -ms $< > $@
```

While it may seem silly to propagate simple rules like these, but the include facility does allow you to define rules of any degree of complexity just once, and maintain them in just one location.

Installing Finished Programs and Libraries

When a program is ready to be released for outside testing or general use, you can use make to install it. Adding a new target and new macro definition to do so is easy:

```
DESTDIR= /proto/project/bin

install: $(PROGRAM)
        -mkdir $(DESTDIR)
        cp $(PROGRAM) $(DESTDIR)
```

A similar target entry can be used for installing a library under the macro naming scheme used in this manual:

```
DESTDIR= /proto/project/lib

install: $(LIBRARY)
        -mkdir $(DESTDIR)
        cp $(LIBRARY) $(DESTDIR)
```

A list of header files might appear as:

```
DESTDIR= /proto/project/include
install: $(LIST)
        -mkdir $(DESTDIR)
        cp $(LIST) $(DESTDIR)
```

Finally, a list of Reference Manual Pages, which are typically distributed in source form, are installed just like header files (these may comprise a subset of the items in the doc subdirectory).

Building the Entire Project

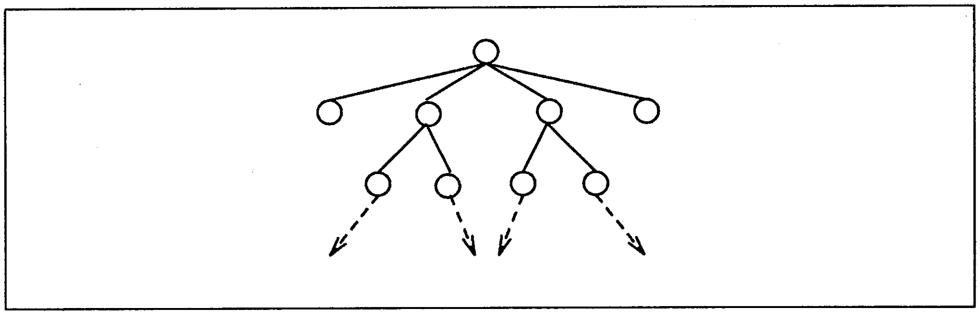
From time to time it is necessary to take a snapshot of the sources, and the object files that they produce. This can either be done as a checkpoint in the development process, or as an intermediate or final build for release to users. Building an entire project is simply a matter of invoking make successively in each subdirectory to build and install each module.

Subsequent examples show how to incorporate these make commands in the root makefile, which should also allow you to build debugging and profiling variants of the project, clean the directories, and install completed modules. The following example show how to use nested make commands to build a simple project.

```
#       Simple root makefile for a project.
TARGETS= all debug profile lint clean test install
SUBDIRS= bin include lib doc
$(TARGETS):
        $(MAKE) $(SUBDIRS) TARGET=$@
$(SUBDIRS): FORCE
        cd $@: $(MAKE) $(TARGET)
FORCE:
```

Maintaining Directory Hierarchies With Recursive Makefiles

If you extend your project hierarchy to include more layers:



chances are that not only will the makefile in each intermediate directory have to produce target files, but it will also have to invoke nested make commands for subdirectories of its own. Files in the current directory can sometimes depend on files in subdirectories, and their target entries need to depend on their counterparts in the subdirectories.

This means that the nested make command for each subdirectory should run before the command in the local directory does. One way to assure that the commands run in the proper order is to make a separate entry for the nested part, and another for the local part. If you add these new targets to the dependency list for the original target, its action will encompass them both.

Targets that encompass equivalent actions in both the local directory and in subdirectories are referred to as *recursive* targets.[30] A makefile with recursive targets is referred to as a *recursive* makefile.

In the case of all, the the nested dependency can be named all.nested; the local dependency, all.local. Note that this example conditionally defines the TARGET macro, rather than using $@, to pass the proper argument to the make command in what is now the all.nested dependency.

```
...
  all := TARGET all

all: all.nested all.local

all.nested:
        $(MAKE) $(SUBDIRS) TARGET=$(TARGET)

$(SUBDIRS): FORCE
        cd $@; $(MAKE) $(TARGET)

all.local: $(PROGRAM)
...
```

[30] Strictly speaking, any target that calls make, with its name as an argument, is recursive. However, here the term is reserved for the narrower case of targets that have both nested and local actions. Targets that only have nested actions are referred to as "nested" targets.

Note that the "nested" target invokes make with the all target as an argument, *not* all.nested. The nested make must also be recursive, unless it is at the bottom of the hierarchy. Either way, it should be invoked with the same name as that used in the parent directory. In the makefile for a leaf directory (one with no subdirectories to descend into), you can simply comment out the rule for the nested target, which will halt any further descent.

Recursive install Targets

This same principle can be extended to all of the generic targets. The install target, however, is something of a special case. If the destination is a parallel directory hierarchy (such as when you are installing completed source code), the parent directories must be created before the destination subdirectories can be. This often means that the make install target in the current directory (which creates the destination directory if needed) must be performed before that in any subdirectory can succeed. So, install.local must appear ahead of install.nested in the dependency list for install.[31]

This next example shows a recursive makefile in a directory with a C program and subdirectories.

Figure 8-17 *Recursive Makefile for Building a* C *Program and Subdirectories*

```
#   @(#) sample.9.mk
#
#   Recursive makefile for a C program and subdirectories.
#   Also includes test and install targets.

SOURCES= main.c data.c
ULIBS= ../lib/libpkg.a
SLIBS= -lcurses -ltermlib
PROGRAM= program

SUBDIRS= sun2 sun3 sun4
TESTSCRIPT= testscript
TESTPROGS= test_1 test_2 test_3
DESTDIR= /proto/project/bin

CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

TARGETS= all debug profile lint clean test
TARGETS.nested= $(TARGETS:%=%.nested)
TARGETS.local= $(TARGETS:%=%.local)

OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)

.KEEP_STATE:
```

[31] If the local target depends on files within a subdirectory, this may force make to descend into that subdirectory twice during a make install run.

```
debug := CFLAGS= -g
profile := CFLAGS= -pg -O
debug.local := CFLAGS= -g
profile.local := CFLAGS= -pg -O

#   Recursive targets:

all := TARGET = all
debug := TARGET = debug
profile := TARGET = profile
lint := TARGET = lint
clean := TARGET = clean
test := TARGET = test
install := TARGET = install

$(TARGETS): $$@.nested $$@.local
install: $$@.local $$@.nested

#   Nested targets:

$(TARGETS.nested) install.nested:
        $(MAKE) $(SUBDIRS) TARGET=$(TARGET)

$(SUBDIRS): FORCE
        cd $@ ; $(MAKE) $(TARGET)

#   Local target entries:

all.local debug.local profile.local: $(PROGRAM)

$(PROGRAM): $(OBJECTS) $(ULIBS)
        $(LINK.c) -o $@ $(OBJECTS) $(ULIBS) $(SLIBS)

$(ULIBS): FORCE
        cd $(@D); $(MAKE) $(@F)
FORCE:

lint.local: $(LINTFILES)
        $(LINT.c) $(LINTFILES)

clean.local:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES) $(TESTSCRIPT) $(TESTPROGS)

test.local: all $(TESTSCRIPT)
        set -x ; $(TESTSCRIPT) > /tmp/test.$$$$
$(TESTSCRIPT): $(TESTSCRIPT).sh $(TESTPROGS)
$(TESTPROGS): $$@.c $(ULIBS)
        $(LINK.c) -o $@ $< $(ULIBS) $(SLIBS)

install.local: $(PROGRAM)
        -mkdir $(DESTDIR)
        -cp $(PROGRAM) $(DESTDIR)
```

Notice that you can still use `make` to build a local target, simply by appending the `.local` suffix to the target name that you're used to. The command

```
make all.local
```

does exactly what you'd expect. However, we recommend against making a habit of this practice, especially where local targets rely on modules in nested targets. If the files in the subdirectories are up to date, it doesn't take very long for make to check them. If they *aren't* up to date, and you build the local target without a full dependency check, there is a strong possibility that the target file you produce will be inconsistent with those lower-level files, at least until it is clean'ed and remade.

Maintaining A Large Library as a Hierarchy of Subsidiaries

When maintaining a very large library, it is sometimes easier to break it up into smaller, subsidiary libraries, and use make to combine them into a complete package. Although you cannot combine libraries directly with ar, you can extract the member files from each subsidiary library, and then archive those files in another step:

```
tutorial% ar xv libx.a
x - x1.o
x - x2.o
x - x3.o
tutorial% ar xv liby.a
x - y1.o
x - y2.o
tutorial% ar rv libz.a *.o
a - x1.o
a - x2.o
a - x3.o
a - y1.o
a - y2.o
ar: creating libz.a
```

A subsidiary library is maintained using a makefile in its own directory, along with the (object) files it is built from. The makefile for the complete library typically makes a symbolic link to each subsidiary archive, extracts their contents into a temporary subdirectory, and archives the resulting files to form the complete package.

The next example updates the subsidiary libraries, creates a temporary directory in which to extracted the files, and extracts them. It uses the * (shell) wild card within that temporary directory to generate the collated list of files. While filename substitutions are generally frowned upon, this use of the wild card is acceptable because the directory is created afresh whenever the target is built. This guarantees that it will contain only files extracted during the *current* make run.

In general, use of shell filename wildcards is considered to be bad form in a makefile. If you *do* use it, you need to take steps to insure that it excludes spurious files, perhaps by isolating affected files in a temporary subdirectory.

The example relies on a naming convention for directories. The name of the directory is taken from the basename of the library it contains. For instance, if libx.a is a subsidiary library, the directory that contains it is named libx. It makes use of suffix replacements in dynamic-macro references to derive the directory name for each specific subdirectory. (You can verify yourself that this is necessary.)

It uses a shell for loop to successively extract each library, and a shell command substitution to collate the object files into proper sequence for linking (using lorder and tsort) as it archives them into the package. Finally, it removes the temporary directory and its contents.

```
#   Simple makefile for collating a library from
#   subsidiaries.

LIBRARY= libz.a
LIBS= libx.a liby.a

ARFLAGS=
CFLAGS= -O
CPPFLAGS=

.KEEP_STATE:
.PRECIOUS: libz.a

all: $(LIBRARY)

$(LIBRARY): $(LIBS)
	-rm -rf tmp
	-mkdir tmp
	set -x ; for i in $(LIBS) ; \
		do ( cd tmp ; ar x ../$$i ) ; done
	( cd tmp ; rm -f __.SYMDEF ; ar cr ../$@ `lorder * | tsort` )
	-ranlib $@
	-rm -rf tmp $(LIBS)

$(LIBS): FORCE
	-cd $(@:.a=) ; $(MAKE) $@
	-ln -s $(@:.a=)/$@ $@
FORCE:
```

For the sake of clarity, this example omits support for alternate variants, as well as the targets for `clean`, `install`, and `test` (`lint` does not apply since the source files are in the subdirectories). This material is added in later examples.

The `rm -f __.SYMDEF` command embedded in the collating line prevents a symbol table in a subsidiary (produced by running `ranlib` on that library) from being archived in this library.

Since the nested `make` commands build the subsidiary libraries before the currently library is processed, it is a simple matter to extend this makefile to account for libraries built from both subsidiaries and object files in the current directory. You need only add the list of object files to the dependency list for the library, and a command to copy them into the temporary subdirectory for collation with object files extracted from subsidiary libraries.

```
#   Simple makefile for collating a library from
#   subsidiaries and local object files.

LIBRARY= libz.a
LIBS= libx.a liby.a
SOURCES= map.o calc.o draw.o
ULIBS= $(LIBRARY)

ARFLAGS=
CFLAGS= -O
CPPFLAGS=

OBJECTS= $(SOURCES.c=.o)

.KEEP_STATE:
.PRECIOUS: libz.a

all: $(LIBRARY)

$(LIBRARY): $(LIBS) $(OBJECTS)
        -rm -rf tmp
        -mkdir tmp
        -cp $(OBJECTS) tmp
        set -x ; for i in $(LIBS) ; \
        do ( cd tmp ; ar x ../$$i ) ; done
        ( cd tmp ; rm -f __.SYMDEF ; ar cr ../$@ `lorder * | tsort` )
        -ranlib $@
        -rm -rf tmp $(LIBS)

$(LIBS): FORCE
        -cd $(@:.a=) ; $(MAKE) $@
        -ln -s $(@:.a=)/$@ $@
FORCE:
```

The next example includes support for debugging and profiling variants, along with recursive targets for `clean`, `lint`, `test`, and `install`.

Figure 8-18 *Makefile for a Hierarchy of Subsidiary Libraries with Variants*

```
#   @(#) sample.10.mk
#
#   Makefile for collating a library from local object files and
#   subsidiary libraries.  Supports alternate variants, and maintains
#   subdirectories recursively.

LIBRARY= libz.a
LIBS= libx.a liby.a
SOURCES= map.c calc.c draw.c
ULIBS= $(LIBRARY)
SLIBS= -lcurses -ltermlib

SUBDIRS= $(LIBS:.a=)
TESTSCRIPT= testscript
TESTPROGS= test_1 test_2 test_3
DESTDIR= /proto/project/lib

ARFLAGS=
CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

TARGETS= lint clean test
TARGETS.nested=$(TARGETS:%=%.nested)
TARGETS.local=$(TARGETS:%=%.local)

OBJECTS= $(SOURCES.c:.c=.o)
LINTFILES= $(SOURCES.c:.c=.ln)

.KEEP_STATE:
.PRECIOUS: libz.a

all profile debug: $(LIBRARY)

debug := CFLAGS= -g
profile := CFLAGS= -O -pg
debug := TARGET= debug
profile := TARGET= profile

$(LIBRARY): $(LIBS) $(OBJECTS)
    -rm -rf tmp
    -mkdir tmp
        -cp $(OBJECTS) tmp
    -set -x ; for i in $(LIBS) ; \
        do ( cd tmp ; ar x ../$$i ) ; done
    ( cd tmp ; rm -f __.SYMDEF ; ar cr ../$@ `lorder * | tsort` )
    -ranlib $@
    -rm -rf tmp $(LIBS)

$(LIBS): FORCE
    -cd $(@:.a=) ; $(MAKE) $(TARGET)
    -ln -s $(@:.a=)/$@ $@
FORCE:
```

```
#   Recursive targets:

  lint := TARGET = lint
  clean := TARGET = clean
  test := TARGET = test
  install := TARGET = install

lint clean test: $$@.nested $$@.local
install: $$@.local $$@.nested

#   Nested targets:

$(TARGETS.nested) install.nested:
        $(MAKE) $(SUBDIRS) TARGET=$(TARGET)

$(SUBDIRS): FORCE
        cd $@ ; $(MAKE) $(TARGET)

#   Local target entries:

lint.local: $(LINTFILES)
    $(LINT.c) $(LINTFILES)

clean.local:
    rm -f $(LIBRARY) $(OBJECTS) $(LINTFILES) $(TESTSCRIPT) $(TESTPROGS)

test.local: all $(TESTSCRIPT)
        set -x ; $(TESTSCRIPT) > /tmp/test.$$$$
$(TESTSCRIPT): $(TESTSCRIPT).sh $(TESTPROGS)
$(TESTPROGS): $$@.c $(ULIBS)
        $(LINK.c) -o $@ $< $(ULIBS) $(SLIBS)

install.local: $(LIBRARY)
        -mkdir $(DESTDIR)
        -cp $(PROGRAM) $(DESTDIR)
```

Closing Remarks about make

make has evolved into a powerful and flexible tool for consistently processing files that stand in a hierarchical relationship to one another. The methods and examples shown in this manual are intended to provide you with an exposure to the kinds of problems that lend themselves to solution with make. There is a large body of folklore about make; strong and varied opinions about its ''best'' use abound. This manual does not make the claim that any one approach or example is necessarily the best available. Compromises between clarity and functionality were made in many of the examples.

Also, there is considerable opinion both pro and against makefiles that use macros extensively. Some experts prefer to tailor makefiles for specific situations. Others prefer that all makefiles look the same and work the same way.

This manual takes the latter approach. The examples are intended to be useful, just as they are, in a wide variety of not-too-complicated settings. As procedures become more complicated, so do the makefiles that implement them. The trick is to know which approach will yield a reasonable makefile that works in a given situation. The examples are intended to give you a flavor for common situations, and some fairly straightforward methods to simplify them using make.

If a template approach is used in a project from the outset, chances are that custom makefiles that evolve from the templates will be more familiar, and therefore easier to understand, to integrate, to maintain, and more importantly, to *re-use*. After all, the less time you spend tinkering with the makefiles, the more time you have to develop your program or project.

9

m4 — a Macro Processor

9

m4 — a Macro Processor

m4 is a macro processor whose primary use has been as a front end for Ratfor in those cases where parameterless macros are not powerful enough. It has also been used for languages as disparate as C and COBOL. m4 is particularly suited for higher-level languages like FORTRAN, PL/I and C since macros are specified in a functional notation.

m4 provides features seldom found even in much larger macro processors, including

- arguments
- condition testing
- arithmetic capabilities
- string and substring functions
- file manipulation

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to 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, that is, replacement of text by other text.

The basic operation of m4 is to act as a filter between its input and its output. As the input is read, each alphanumeric ''token'' (that is, string of letters and digits) is checked. If it is the name of a macro, then it macro is replaced by the text that has been assigned to it (*defining text*), and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the text before it is rescanned.

m4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-in macros and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

9.1. Using the m4 Command

The basic m4 command line looks like this:

```
m4  [filename ...]
```

Each argument file is processed in order; if there are no arguments, or if an argument is '–', the standard input is read at that point. The processed text is written to the standard output, which may be captured for subsequent processing using redirection:

```
m4 [filename ...] > outputfile
```

9.2. Defining Macros

The primary built-in function of m4 is `define`, which is used to define new macros. The input

```
define( name , value )
```

defines the string *name* as *value*. All subsequent occurrences of *name* will be replaced by *value*, unless *name* is redefined, or its definition is removed. Note that *name* must be alphanumeric, and must begin with a letter; the underscore character, _ is taken as a letter. The *value* argument is any text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example might be:

```
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 '(', it is assumed to have no arguments. This is the situation for N above; it is actually a macro with no arguments, and thus when it is used there need be no parenthesis following it.

m4 divides its input into tokens, so a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
...
if (NNN > 100)
```

the variable NNN is absolutely unrelated to the defined macro N, even though it contains several N's.

Macros can be defined in terms of other macros. 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 translated to `100` as is is scanned, so changing `N` does not change `M`.

This behavior arises because `m4` expands macro names into their defining text immediately. Here, that means that when the string `N` is seen while 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 alternatives. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now `M` is defined to be the string `N`, so when you ask for `M` later, you'll always get the value of `N` at that time (because the `M` will be replaced by `N` which will be replaced in turn by its value).

9.3. Quoting and Comments

The more general solution is to delay the expansion of the arguments of `define` by *quoting* them. Any text enclosed within the single-quote marks ` and ´ is not expanded immediately, but merely has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N')
```

the quotes 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`, rather than the value of the `N` macro.

The general rule is that `m4` always strips off one level of single quotes 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)
```

While this statement is ignored by m4, since you can only define macros with names that start with an alphabetical character or underscore, it obviously doesn't have the effect you wanted. To redefine N, you must delay the evaluation by quoting it:

```
define(N, 100)
...
define(`N', 200)
```

If the ` and ´ characters are not convenient for some reason, the quote and end-quote characters can be changed with the built-in changequote function. For instance:

```
changequote([, ])
```

the left and right brackets the new quote and end-quote characters. You can restore the original characters with just **changequote**. There are two additional built-ins related to define. undefine removes the definition of some macro or built-in:

```
undefine(`N')
```

removes the definition of *N*. (Why are the quotes absolutely necessary?) 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. In particular, m4 pre-defines the name unix.

ifdef actually permits three arguments; if the name is undefined, the value of ifdef is then the third argument, as in

```
ifdef(`unix', on SunOS, not on SunOS)
```

Don't forget the quotes around the argument.

Comments in m4 are introduced by the # (sharp) character. All text from the # to the end of the line is taken as a comment and otherwise ignored.

9.4. Macros with Arguments

So far we have discussed the simplest form of macro processing — replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its `define`) any occurrence of *$n* 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)
```

evaluates to

```
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*, although that is less commonly used. Arguments that are not supplied are replaced by null strings, so we can define a macro *cat* which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

$4 through *$9* are null, since no corresponding arguments were provided.

Leading unquoted SPACE's, TAB's, or NEWLINE's 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 commas can be nested inside parentheses. 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.

9.5. Arithmetic Built-ins

m4 provides two built-in functions for doing arithmetic on integers (only). 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)')
```

which defines N1 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. `eval` provides the operators (in decreasing order of precedence), as shown in the table below.

Table 9-1 *Operators to the eval built in in m4*

Operator	*Meaning*
`unary + and -`	add and subtract
`**` *or* `^`	exponentiation
`*  /  %`	multiply, divide, and modulus
`+  -`	binary add and subtract
`==  !=  <  <=  >  >=`	equal, not equal, less than, less than or equal, greater than, greater than or equal
`!`	logical not
`&` *or* `&&`	logical and)
`|` *or* `||`	(logical or)

Parentheses may be used to group operations where needed. All the operands of an expression given to `eval` must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in `eval` is 32 bits.

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.

9.6. File Manipulation

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, the alternate form `sinclude` can be used; `sinclude` (``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. m4 maintains nine of these diversions, numbered 1 through 9. If you say

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as n. Diverting to this file is stopped by another divert command; in particular, divert or divert(0) resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and undivert with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of undivert is *not* the diverted text. 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.

9.7. Running SunOS Commands

You can run any SunOS command using 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 process ID (*pid*) of the current process.

9.8. Conditionals

There is a built-in called ifelse which enables you to perform arbitrary conditional testing. In its simplest form,

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If these are identical, ifelse returns the string *c*; otherwise it returns *d*. Thus we might define a macro called *compare* which compares two strings and returns ''yes'' or ''no'' according to whether they are the same or different.

```
define(compare, `ifelse($1, $2, yes, no)')
```

Note the quotes, which prevent too-early evaluation of ifelse.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

```
ifelse(a, b, c, d, e, f, g)
```

if the string a matches the string b, the result is c. Otherwise, if d is the same as e, the result is f. Otherwise the result is g. If the final argument is omitted, the result is null, so

```
ifelse(a, b, c)
```

is c if a matches b, and null otherwise.

9.9. String Manipulation

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. substr(s, i, n) returns the substring of s that starts at the ith position (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)
```

evaluates to

```
ow is the time
```

If either i or n is out of range, various sensible things happen.

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 in t. That is,

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If t is shorter than f, characters which don't have an entry in t are deleted; as a limiting case, if t is not present at all, characters in f are deleted from s. So

```
translit(s, aeiou)
```

deletes vowels from s.

There is also a built-in called `dnl` which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up `m4` output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add `dnl` to each of these lines, the newlines will disappear.

Another way to achieve this[32] is:

```
divert(-1)
        define(. . .)
        . . .
divert
```

9.10. Printing

The built-in `errprint` writes its arguments to the standard error file. Thus you can say

```
errprint('fatal error')
```

`dumpdef` is a debugging aid which 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 to quote the names!

9.11. Summary of Built-in `m4` Macros

Table 9-2 *Summary of Built-in m4 Macros*

Built In	*Description*
`changequote(L, R)`	Change left quote to L, right quote to R
`define(`*name*`, `*replacement*`)`	define *name* as *replacement*
`divert(`*number*`)`	Divert output to stream *number*
`divnum`	Return number of currently active diversions
`dnl`	Delete up to and including newline

[32] Thanks to J. E. Weythman.

Table 9-2 *Summary of Built-in m4 Macros— Continued*

Built In	*Description*
`dumpdef` (\`*name*´, \`*name*´, . . .)	Dump specified definitions
`errprint` (*s*, *s*, . . .)	Write arguments *s* to standard error
`eval` (*numeric expression*)	Evaluate *numeric expression*
`ifdef` (\`*name*´, *true string*, *false string*)	Return *true string* if *name* is defined, *false string* if *name* is not defined
`ifelse` (*a*, *b*, *c*, *d*)	If *a* and *b* are equal, return *c*, else return *d*
`include` (*file*)	Include contents of *file*
`incr` (*number*)	Increment *number* by 1
`index` (*s1*, *s2*)	Return position in *s1* where *s2* occurs, or −1 if no occurrence
`len` (*string*)	Return length of *string*
`maketemp` (. . .XXXXX. . .)	Make a temporary file
`sinclude` (*file*)	Include contents of *file* — ignored and continue if *file* not found.
`substr` (*string*, *position*, *number*)	Return substring of *string* starting at *position* and *number* characters long
`syscmd` (*command*)	Run *command* in the system
`translit` (*string*, *from*, *to*)	Transliterate characters in *string* from the set specified by *from* to the set specified by *to*
`undefine` (\`*name*´)	Remove *name* from the list of definitions
`undivert` (*number*, *number*, . . .)	Append diversion *number* to the current diversion

10

lex — a Lexical Analyzer Generator

10

lex — a Lexical Analyzer Generator

lex is a program generator designed for lexical processing of character input streams. lex accepts a high-level, problem-oriented specification for character string matching, and produces a program in a general-purpose language which recognizes regular expressions. The regular expressions are specified by the programmer in the source specifications given to lex. The lex written 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 programmer 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 programmer supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general-purpose programming language employed for the programmer's program fragments. Thus, a high-level expression language is provided to write the string expressions to be matched while the programmer's freedom to write actions is unimpaired. This avoids forcing the programmer who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by lex. The program fragments written by the programmer are executed in the order in which the corresponding regular expressions occur in the input stream.

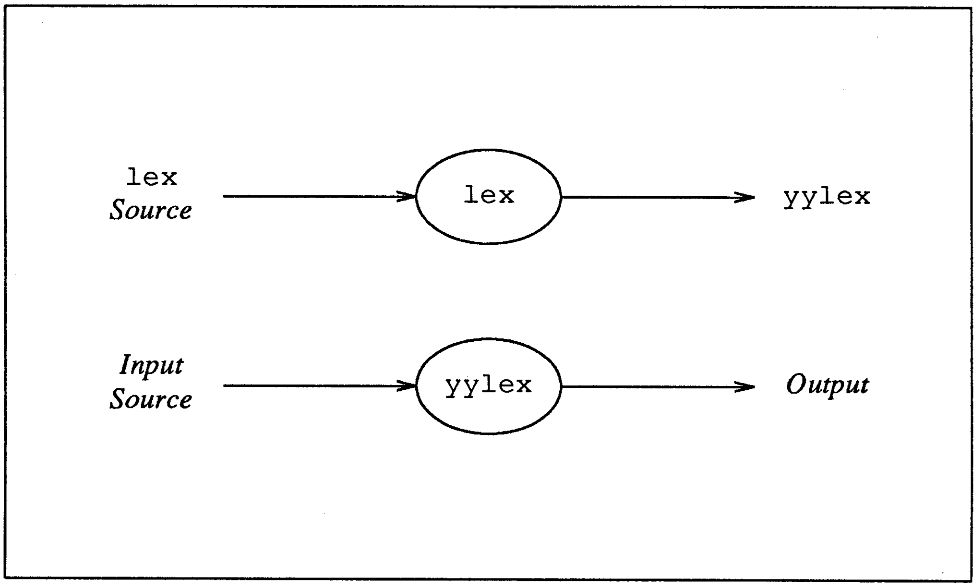
The lexical analysis programs written with lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream is then backed up to the end of the current partition, so that the programmer has general freedom to manipulate it.

lex is designed to simplify interfacing with yacc, which is described in the next chapter.

lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called 'host languages.' Just as general-purpose languages can produce code to run on different computer hardware, lex can write code in different host languages. The host language is used for the output code generated by lex and also for the program fragments added by the programmer. Compatible run-time libraries for the different host languages are also provided. This makes lex adaptable to different environments and different programmer. Each application may be directed to the combination of hardware and host language appropriate to the task, the programmer's background, and the properties of local implementations.

lex turns the programmer's expressions and actions (called source in this document) into the host general-purpose language; the generated program is named yylex. The yylex program recognizes expressions in a stream (called input in this document) and performs the specified actions for each expression as it is detected — see Figure 10-1 below.

Figure 10-1 *An overview of* lex



For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

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

is 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 which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C 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 ...'; and the $ indicates 'end-of-line'. No action is specified, so the program generated by lex (yylex) ignores these characters. Everything else is copied to the output stream. To change any remaining string of blanks or tabs to a single blank, add another rule:

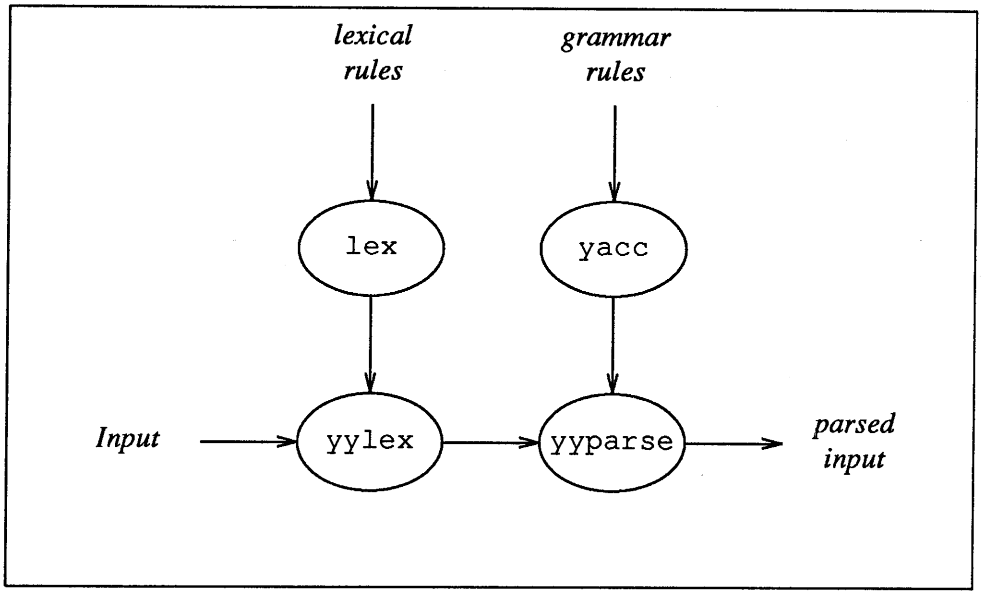
```
%%
[ \t]+$ ;
[ \t]+  printf(" ");
```

The finite automaton generated for this source scans for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the ends of lines, and the second rule all remaining strings of blanks or tabs.

lex can also be used with a parser generator to perform the lexical analysis phase.

lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface lex and yacc lex programs recognize only regular expressions; yacc writes parsers that accept a large class of context-free grammars, but require a lower-level analyzer to recognize input tokens. Thus, a combination of lex and yacc is often appropriate. When used as a preprocessor for a later parser generator, lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 10-2. Additional programs, written by other generators or by hand, can be added easily to programs written by lex.

Figure 10-2 lex *with* yacc



yacc programmers will realize that the name yylex is what yacc expects its lexical analyzer to be named, so that the use of this name by lex simplifies interfacing.

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

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

lex is not limited to source which 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 leave the input pointer just before "cd..." Such backup is more costly than processing simpler languages.

10.1. lex Source

The general format of lex source is:

```
{definitions}
%%
{rules}
%%
{programmer subroutines}
```

where the definitions and the programmer 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 thus

```
%%
```

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

In the outline of lex programs shown above, the *rules* represent the programmer's control decisions; they are a table, in which the left column contains *regular expressions* (see section 10.2) and the right column contains *actions*, program fragments to be executed when the expressions

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

to look for the string integer in the input stream and print the message 'found keyword INT' whenever it appears. In this example the host procedural language is C and the C library function printf() is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just 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. `lex` 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 this is described later.

10.2. `lex` Regular Expressions

The definitions of regular expressions are very similar to those in the editors *ex*(1) and *vi*(1). A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which 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`.

Operators

The operator characters are

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

and if they are to be used as text characters, an escape must be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

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

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the programmer can avoid remembering the list above of current operator characters, and is safe should further extensions to `lex` lengthen the list.

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

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [ ] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \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.

Character Classes

Classes of characters can be specified using the operator pair [ ]. The construction `[abc]` matches a single character, which may be `a`, `b`, or `c`. Within square brackets, most operator meanings are ignored. Only three characters are special: \, –, and ^. The – character indicates ranges. For example,

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

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using – between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation-dependent and generates a warning message. For example, `[0-z]` in ASCII is many more characters than it is in EBCDIC. If it is desired to include the character – in a character class, it should be first or last, thus:

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

matches all the digits and the two signs.

In character classes, the ^ 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 system's character set. Thus

```
[^abc]
```

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

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

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary Character

To match almost any character, the operator character

(period) is the class of all characters except newline. Escaping into octal is possible although non-portable:

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

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

Optional Expressions

The operator ? indicates an optional element of an expression. Thus

```
ab?c
```

matches either ac or abc.

Repeated Expressions

Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive a characters, including zero; while

```
a+
```

is one or more instances of a. For example,

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

is all strings of lower case letters. And

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

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping

The operator | indicates alternation:

```
(ab | cd)
```

matches either ab or cd. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab | cd
```

would have sufficed. Parentheses can be used for more complex expressions:

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

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

Context Sensitivity

lex recognizes a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression is only be matched at the beginning of a line This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [ ] operators. If the very last character is $, the expression is only be matched at the end of a line (when immediately followed by newline).

The latter operator is a special case of the / operator character, which indicates trailing context. The expression

```
ab/cd
```

matches the string ab, but only if it is followed by cd. Thus

```
ab$
```

is the same as

```
ab/\n.
```

Left context is handled in lex by *start conditions* as explained in section 10.9 — *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 prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered 'being at the beginning of a line' to be start condition ONE, then the ^ operator would be equivalent to

```
<ONE>.
```

Start conditions are explained more fully below.

Repetitions and Definitions

The operators { } 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. 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 a.

Finally, initial % is special, being the separator for lex source segments.

10.3. lex Actions

When an expression written as above is matched, lex executes the corresponding action. This section describes some features of lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the lex programmer who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When lex is being used with yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

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

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

which ignores the three spacing characters (blank, tab, and newline).

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

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

with the same result. The quotes around \n and \t are not required.

Actual Text that Matched

In more complex actions, the programmer often wants to know the actual text that matched some expression like `[a-z]+`. `lex` leaves this text in an external character array named `yytext`.
Thus, to print the name found, a rule like

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

prints the string in `yytext`. The C function `printf` accepts a format argument and data to be printed; in this case, the format is 'print string' (% indicating data conversion, and s indicating 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 may be written as `ECHO`:

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

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches `read()` it normally matches 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.

Length of Matched Text

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

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

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

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

yymore *and* yyless

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 would overwrite the current entry in yytext. Second, yyless(n) may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument n indicates the number of characters to be retained in yytext. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write:

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

which, when faced with a string such as abc\def" first matches the five characters "abc\ ; then the call to yymore() tacks the next part of the string, "def , onto the end. Note that the final quote 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 of resolving (in old-style C) the ambiguity of ‘=–a’. Suppose it is desired to treat this as ‘=– a’ but 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:

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

performs the other interpretation. Note that the expressions for the two cases might more easily be written:

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

in the first case and

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

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of '=-3', however, makes

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

a still better rule.

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

1. `input()` which returns the next input character;
2. `output(c)` which writes the character `c` on the output; and
3. `unput(c)` pushes the character `c` back onto the input stream to be read later by `input()`.

By default these routines are provided as macro definitions, but the programmer can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to transmit input or output 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 `lex` lookahead will not work. `lex` does not look ahead at all if it does not have to, but every rule ending in `+ * ?` or `$` or containing `/` implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See section 10.10 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 the programmer will 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 programmer should provide a `yywrap` which arranges for new input and returns 0. This instructs `lex` to continue processing. The default `yywrap` always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which 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.

10.4. Ambiguous Source Rules

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

1. The longest match is preferred.
2. Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

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

to be given in that order. 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`) will not match the expression `integer`, and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like `.*` dangerous. For example,

```
'.*'
```

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

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

the above expression matches

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

which is probably not what was wanted. A better rule 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 `.` operator does not match newline. Thus expressions like `.*` stop on the current line. Don't try to defeat this with expressions like `[.\n]+` or equivalents; the `lex` generated program will try to read the entire input file, causing internal buffer overflows.

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

Sometimes the programmer would like to override this choice. The action REJECT means 'go do the next alternative.' It executes whatever rule was second choice after the current rule. The position of the input pointer is adjusted accordingly. Suppose the programmer 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 is then counted. In this example, of course, the programmer 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 *a priori* to tell which input characters were in both classes.

Consider the two rules

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

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

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

```
%%
[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.

10.5. lex Source Definitions

Remember the format of the lex source:

```
  {definitions}
  %%
  {rules}
  %%
  {programmer routines}
```

So far only the rules have been described. The programmer needs additional options, though, to define variables for use in his 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 which is not part of a lex rule or action which begins with a blank or tab is copied into the lex-generated program. Such source input prior to the first %% delimiter is 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 which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the lex source or the generated code. The comments should follow the host language convention.

2. Anything included between lines containing only the delimiters %{ 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 formats, etc., 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 associates the string given as a translation with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be invoked by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

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

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

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

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

The definitions section may also contain other commands, including the selection of a host language, 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 below under section 10.11 — *Summary of Source Format.*

10.6. Using `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 on a file named `lex.yy.c`. The I/O library is defined in terms of the C standard library in section 3 of the *SunOS Reference Manual.*

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

```
tutorial% lex source
tutorial% 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 below. 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. `lex` has several options which are described in the `lex(1)` manual page.

10.7. lex and yacc

If you want to use lex with yacc, note that what lex writes is a program named yylex(), the name required by yacc for its analyzer. Normally, the default main program in 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);
```

to return the appropriate token value.

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 command sequence can just be:

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

The lex and yacc programs can be generated in either order.

10.8. Examples

As a trivial problem, consider copying an input file while adding 3 to every non-negative number divisible by 7. Here is a suitable lex source program

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

to do just that. The rule [0-9]+ recognizes strings of digits; atoi() converts the digits to binary and stores the result in k.
The operator % (remainder) is used to check whether k is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as 49.63 or X7. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as shown below.

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

Numerical strings containing a '.' 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 which constructs a histogram of the lengths of words, where a word is defined as a string of letters.

```
        int lengs[100];
%%
[a-z]+  lengs[yyleng]++;
.       |
\n      ;
%%
1 s.
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 by N. L. Schryer to convert double-precision FORTRAN to single-precision FORTRAN. Because FORTRAN does not distinguish upper and lower case 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 (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"      "[^ 0]   ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as 'beginning of line, then five blanks, then anything but blank or zero.' Note the two different meanings of ^. There follow some rules to change double-precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+        |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+       |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+       {
     /* convert constants */
     for(p=yytext; *p != 0; p++)
          {
          if (*p == 'd' || *p == 'D')
               *p=+ 'e'- 'd';
          ECHO;
          }
```

After the floating point constant is recognized, it is scanned by the `for` loop to find the letter `d` or `D`. The program then adds 'e'–'d', which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which 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;
```

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

10.9. 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 ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ 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: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the programmer's action code; such a flag is the simplest way of dealing with the problem, since lex is not involved at all. It may be more convenient, however, to have lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It is only be recognized when lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

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

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

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

should be adequate.

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

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

where the conditions may be named in any order. The word `Start` may be abbreviated to `s` or `S`. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when `lex` is in the start condition `name1`. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to `name1`. To resume the normal state,

```
BEGIN 0;
```

which resets to the initial condition of the `lex` automaton interpreter. A rule may be active in several start conditions:

```
<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 programmer's code.

10.10. Character Set

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 in the same form as the character constant `'a'`. If this interpretation is changed, by providing I/O routines which translate the characters, `lex` must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by two lines containing only '%T'. The table contains lines of the form

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

which indicate the value associated with each character. Thus the next example

Figure 10-3 *Sample character table.*

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

maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and – into 28 and 29, and the digits into 30 through 39. Note 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 may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

10.11. Summary of Source Format

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

```
{definitions}
%%
{rules}
%%
{programmer 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 condition declarations, 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:

Table 10-1 *Changing Internal Array Sizes in lex*

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 may be continued on succeeding lines by using braces to delimit it.

Regular expressions in lex use the following operators:

Table 10-2 *Regular Expression Operators in lex*

Operator	*Meaning*
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

10.12. Caveats and Bugs

There are pathological expressions which produce exponential growth of the tables when converted to deterministic automata; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT is executed, the programmer must not have used unput to change the characters forthcoming from the input stream. This is the only restriction on the programmer's ability to manipulate the not-yet-processed input.

11

yacc — Yet Another Compiler-Compiler

11

yacc — Yet Another Compiler-Compiler

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an 'input language' which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

yacc provides a general tool for describing the input to a computer program. The yacc programmer specifies the structure of the input, together with code to be invoked as each item is recognized. yacc 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 programmer's application handled by this subroutine.

The input subroutine produced by yacc calls a programmer-supplied routine to return the next basic input item. Thus, the programmer can specify his input in terms of individual input characters, or in terms of higher-level constructs such as names and numbers. The programmer-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

The class of specifications that yacc accepts is a very general one: LALR(1) grammars with disambiguating rules.

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

yacc provides a general tool for imposing structure on the input to a computer program. The yacc programmer 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. yacc then generates a function to control the input process. This function, called a *parser*, calls the programmer-supplied low-level input routine (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 programmer 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.

yacc generates its actions and output subroutines in C. Moreover, many of the syntactic conventions of yacc follow C.

The heart of the yacc 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 quotes — implying 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 routine reads the input stream, recognizing the lower-level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol.* To avoid confusion, terminal symbols are 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 ',' must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively 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 may be self-contradictory, or they may require a more powerful recognition mechanism than that available to yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for yacc to handle are also frequently difficult for 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 basic process of preparing a yacc specification; Section 11.1 describes the preparation of grammar rules, Section 11.2 the preparation of the programmer-supplied actions associated with these rules, and Section 11.3 the preparation of lexical analyzers. Section 11.4 describes the operation of the parser. Section 11.5 discusses various reasons why yacc may be unable to produce a parser from a specification, and what to do about it. Section 11.6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 11.7 discusses error detection and recovery. Section 11.8 discusses the operating environment and special features of the parsers yacc produces. Section 11.9 gives some suggestions which should improve the style and efficiency of the specifications. Section 11.10 discusses some advanced topics. Section 11.11 has a brief example, and section 11.12 gives a summary of the yacc input syntax. Section 11.13 gives an example using some of the more advanced features of yacc, and, finally, section 11.14 describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of yacc.

11.1. Basic Specifications

Names refer to either tokens or nonterminal symbols. yacc requires token names to be declared as such. In addition, for reasons discussed in Section 11.3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar) rules*, and *programs*. The sections are separated by double percent %% marks. The percent % 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 may be omitted also; thus, the smallest legal yacc specification is

```
%%
rules
```

Spaces (also called blanks), tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal — they are enclosed in /* . . . */, as in C and PL/I.

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

```
A  :  BODY  ;
```

A represents a nonterminal name, and *BODY* represents a sequence of zero or more names and literals. The colon and the semicolon are yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot '.', underscore '_', and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes ‘’. As in C, the backslash ‘\’ is an escape character within literals, and all the C escapes are recognized:

```
'\n'    newline
'\r'    return
'\''    single quote '
'\\'    backslash '\'
'\t'    tab
'\b'    backspace
'\f'    form feed
'\xxx'  'xxx' in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, 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. See Sections 3 , 5, and 6 for much more discussion. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. 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 which matches 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 programmer-supplied lexical analyzer to return the endmarker when appropriate — see Section 11.3, below. Usually the endmarker represents some reasonably obvious I/O status, such as 'end-of-file' or 'end-of-record'.

11.2. Actions

With each grammar rule, the programmer may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, 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 symbol '$' is used as a signal to yacc in this context.

To return a value, the action normally sets the pseudo-variable '$$' to some value. For example, an action that does nothing but return the value 1 is

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

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A       :       B  C  D   ;
```

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 $1 (the first element in it). 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. yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the 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. yacc actually treats the above example as if it had been written:

```
$ACT     :          /* empty */
                          {   $$ = 1;   }
         ;

A        :          B  $ACT  C
                          {    x = $2;    y = $3;   }
         ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function node, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. The parse tree can be built by supplying actions such as:

```
expr    :       expr  ´+´  expr
                         {  $$ = node( ´+´, $1, $3 );   }
```

in the specification.

The programmer may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks '%{' and '%}'. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{   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'; the programmer should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 11.10.

11.3. Lexical Analysis

The programmer 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, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable yylval().

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by yacc, or chosen by the programmer. 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 the 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 `if` or `while` as token names 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 (see Section 11.7.

As mentioned above, the token numbers may be chosen by `yacc` or by the programmer. 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 programmer; thus, 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 the *lex* program developed by Mike Lesk[8] and described in chapter NumberOf(Lex_Lexical_Analyzer) , TitleOf(Lex_Lexical_Analyzer) . These lexical analyzers are designed to work in close harmony with `yacc` parsers. The specifications use regular expressions instead of grammar rules. `lex` can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

11.4. How the Parser Works

yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by yacc consists of a finite-state machine with a stack. The parser can read and remember 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 may result in states being pushed onto the stack, or popped off the stack, and in the lookahead token being processed or left alone.

shift Action

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
            IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

reduce Action

The *reduce* action keeps the stack from growing without bound. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a '.') is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
            . 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 which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
            A       goto 20
```

which pushes state 20 onto the stack, and becomes 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 the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of programmer-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 programmer's 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.

`accept` and `error` Actions

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 11.7.

It is time for an example! Consider the specification

```
%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
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING  DONG  DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, DING, is read, becoming the lookahead token. The action in state 0 on DING is 'shift 3', so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, DONG, is read, becoming the lookahead token. The action in state 3 on the token DONG is 'shift 6', so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
        sound   :   DING  DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off 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, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by '$end' in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as DING DONG DONG, DING DONG, DING DONG DELL DELL, and so on. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

11.5. Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
    expr    :       expr   '-'   expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not unambiguously 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*.

yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given 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 happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict.* Note that there are never any 'shift/shift' conflicts.

When there are shift/reduce or reduce/reduce conflicts, yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule.*

yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.

2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the programmer rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, `yacc` always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, `yacc` will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an 'if-then-else' construction:

```
stat    :       IF  '('  cond  ')'  stat
        |       IF  '('  cond  ')'  stat  ELSE  stat
        ;
```

In these rules, `IF` and `ELSE` are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if rule*, and the second the *if-else rule.*

These two rules form an ambiguous construction, since input of the form:

```
IF ( condition -1 )  IF ( condition -2 ) statement -1  ELSE statement -2
```

can be structured according to these rules in two ways:

```
IF   (   condition -1   )   {
         IF   (   condition -2   )   statement -1
}
ELSE   statement -2
```

or

```
IF   (   condition -1   )   {
         IF   (   condition -2   )   statement -1
         ELSE   statement -2
}
```

The second interpretation is the one given in most programming languages having this construct. Each `ELSE` is associated with the last preceding 'un-*ELSE*'d' `IF`. In this example, consider the situation where the parser has seen

```
IF   (   condition -1   )   IF   (   condition -2   )   statement -1
```

and is looking at the `ELSE`. It can immediately reduce by the simple-if rule to get

```
IF  (  condition -1  )  stat
```

and then read the remaining input,

```
ELSE  statement -2
```

and reduce

```
IF  (  condition -1  )  stat  ELSE  statement -2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the ELSE may be shifted, *statement*-2 read, and then the right hand portion of

```
IF  (  condition -1  )  IF  (  condition -2  )  statement -1  ELSE  statement -2
```

can be reduced by the if-else rule to get

```
IF  (  condition -1  )  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, which 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  (  condition -1  )  IF  (  condition -2  )  statement -1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of yacc are best understood by examining the verbose (–v) option output file. 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 which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF  (  cond  )  stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is ELSE, it is possible to shift into state 45. State 45 will have, 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 which can be reduced. In most states, there will be at most one reduce action possible in the state, and this will be the default command. Programmers who encounter unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the programmer 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 cited in Chapter 1 might be consulted.

11.6. Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr  :  expr  OP  expr
```

and

```
expr  :  UNARY  expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the programmer 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 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, which are also left-associative. The keyword `%right` is used to describe right-associative operators, and the keyword `%nonassoc` is used to describe operators, like the `.LT.` operator 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 may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword `%prec` changes the precedence level associated with a particular grammar rule. `%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 changes 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 may have no precedence and associativity associated with them.

3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left-associative implies reduce, right-associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by `yacc`. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially 'cookbook' fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

11.7. Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser 'restarted' after an error. A general class of algorithms to do 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 programmer 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.

In order 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 quietly 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 in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat    :       error   ´;´
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ´;´. All tokens after the error and before the next ´;´ cannot be shifted, and are discarded. When the ´;´ is seen, this rule will be reduced, and any 'cleanup' action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
  input   :       error  '\n'  {  printf( "Reenter last line: " );  }  input
                          {       $$  =  $4;  }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
  input   :       error  '\n'
                          {       yyerrok;
                                  printf( "Reenter last line: " );   }
                  input
                          {       $$  =  $4;  }
          ;
```

As mentioned above, the token seen immediately after the 'error' symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the programmer, 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 programmer can get control to deal with the error actions required by other portions of the program.

11.8. The yacc Environment

When the programmer inputs a specification to yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the name may differ from installation to installation). yacc produces an integer-valued function called yyparse(). When yyparse() is called, it in turn repeatedly calls yylex() — the lexical analyzer supplied by the programmer (see Section 11.3) 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 programmer 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.

The programmer must supply these two routines in one form or another. They can be as simple as the following example, or they can be as complex as needed.

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to yyerror() is a string containing an error message, usually the string 'syntax error'. The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable yychar contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics.

The external integer variable yydebug is normally set to 0. If it is set to a nonzero value, the parser generates a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

11.9. Hints for 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. The following style hints owe much to Brian Kernighan.

1. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of 'knowing who to blame when things go wrong.'
2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
3. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be added easily.
5. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in section 11.11 is written following this style, as are the examples in the text of this paper (where space permits). The programmer must make up his own mind 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 will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right-recursive rules, such as

```
seq     :       item
        |       item  seq
        ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the programmer 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 would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if yacc is asked to decide which empty sequence it has seen, when it 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 might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}

        other declarations

%%

prog    :       decls  stats
        ;

decls   :       /* empty */
                        {       dflag = 1;  }
        |       decls  declaration
        ;

stats   :       /* empty */
                        {       dflag = 0;  }
        |       stats  statement
        ;

        other rules
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except* for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single-token exception does not affect the lexical scan.

This kind of 'backdoor' approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the programmer 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 programmer can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

11.10. Advanced Topics

This section discusses a number of advanced features of `yacc`.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT makes `yyparse` return the value 0; YYERROR makes the parser behave as if the current input symbol results in a syntax error; `yyerror()` is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent    :   adj  noun  verb  adj  noun
            {  look at the sentence  . . .  }
    ;

adj :   THE     {   $$ = THE;  }
    |   YOUNG   {   $$ = YOUNG;  }
    . . .
    ;

noun    :   DOG
            {   $$ = DOG;  }
    |   CRONE
            {   if( $0 == YOUNG ){
                    printf( "what?\n" );
                    }
                $$ = CRONE;
                }
    ;
    . . .
```

In the action following the word `CRONE`, a check is made that the preceding token shifted was not `YOUNG`. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. yacc can also support values of other types, including structures. In addition, yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The yacc value stack (see Section 11.4) is declared to be a union of the various types of values desired. The programmer declares the union, and associates a union member name to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, yacc automatically inserts the appropriate union name, so that no unwanted conversions will take place. In addition, type-checking commands such as lint(1) will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the programmer 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 programmer includes in the declaration section:

```
%union  {
        body of union ...
        }
```

This declares the yacc value stack, and the external variables yylval and yyval, to have type equal to this union. If yacc was invoked with the **–d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the 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>  '+'  '-'
```

will tag any reference to values returned by these two tokens with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type  <nodetype>  expr  stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left-context values (such as $0 — see the previous subsection) leaves yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule    :   aaa  {  $<intval>$  =  3;  } bbb
                 {   fun( $<intval>2, $<other>0 );  }
      ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in 11.13. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the yacc value stack is used to hold int's, as was true historically. This paper is reprinted in this manual.

11.11. A Simple Example

This example gives the complete yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled 'a' through 'z', and accepts arithmetic expressions made up of the operators +, –, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is 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    /* supplies 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 lower case letter, yylval=0 thru 25 */
/* return DIGIT for digit, yylval=0 thru 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);
}
```

11.12. yacc Input Syntax

This section describes the yacc input syntax, as a yacc specification. Context dependencies, etc., are not considered. Ironically, the yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide 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 IDENTIFIERs, but never as part of C_IDENTIFIERs.

```
        /* grammar for the input to yacc */

        /* basic entities */
%token  IDENTIFIER      /* includes identifiers and literals */
%token  C_IDENTIFIER    /* identifier (not literal) followed by : */
%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    {    In this action, 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       /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER   /* NOTE: 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  ';'
        ;
```

11.13. An Advanced Example

This section gives an example of a grammar using some of the advanced features discussed in Section 11.10. The desk calculator example in section 11.11 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 may also be used. The usage is similar to that in section 11.11 — assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of `yacc` and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, `INTERVAL`, by using `typedef`.
The `yacc` value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of `YYERROR` to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of `yacc` is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval-value. This causes a large number of conflicts when the grammar is run through `yacc`: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

```
2.5 + ( 3.5 - 4. )
```

and

```
2.5 + ( 3.5 , 4. )
```

Notice that the 2.5 is to be used in an interval-valued expression in the second example, but this fact is not known until the ',' is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. 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 evaded by having two rules for each binary interval-valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar-valued expressions scalar-valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* 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  of  order\n"  );
                                    YYERROR;
                                    }
                           }
         |        VREG
                           {        $$  =  vreg[$1];    }
         |        vexp  ´+´  vexp
                           {        $$.hi  =  $1.hi  +  $3.hi;
                                    $$.lo  =  $1.lo  +  $3.lo;    }
         |        dexp  ´+´  vexp
                           {        $$.hi  =  $1  +  $3.hi;
                                    $$.lo  =  $1  +  $3.lo;    }
         |        vexp  ´-´  vexp
                           {        $$.hi  =  $1.hi  -  $3.lo;
                                    $$.lo  =  $1.lo  -  $3.hi;    }
         |        dexp  ´-´  vexp
                           {        $$.hi  =  $1  -  $3.lo;
                                    $$.lo  =  $1  -  $3.hi;    }
         |        vexp  ´*´  vexp
                           {        $$  =  vmul(  $1.lo,  $1.hi,  $3  );  }
         |        dexp  ´*´  vexp
                           {        $$  =  vmul(  $1,  $1,  $3  );  }
         |        vexp  ´/´  vexp
                           {        if(  dcheck(  $3  )  )  YYERROR;
                                    $$  =  vdiv(  $1.lo,  $1.hi,  $3  );  }
         |        dexp  ´/´  vexp
                           {        if(  dcheck(  $3  )  )  YYERROR;
                                    $$  =  vdiv(  $1,  $1,  $3  );  }
         |        ´-´  vexp         %prec  UMINUS
                           {        $$.hi  =  -$2.lo;    $$.lo  =  -$2.hi;    }
         |        ´(´  vexp  ´)´
                           {        $$  =  $2;  }
         ;
```

```
%%
# define BSZ 50  /* buffer size for floating point 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( '.' );
                                        /* will cause syntax error */
                                continue;
                                }
                        if(  c  ==  'e'  ){
                                if( exp++ ) return( 'e' );
                                        /* will cause syntax error */
                                continue;
                                }
                        /* end of number */
                        break;
                        }
                *cp  =  '\0';
                if(  (cp-buf)  >=  BSZ  )
                        printf(  "constant  too  long:  truncated\n"  );
                else ungetc( c, stdin );  /* push back last char read */
                yylval.dval  =  atof(  buf  );
                return(  CONST  );
                }
        return(  c  );
        }
INTERVAL  hilo(  a,  b,  c,  d  )  double  a,  b,  c,  d;  {
        /* returns the smallest interval containing a, b, c, and d */
        /* used by *, / routines */
        INTERVAL  v;
        if(  a>b  )  {  v.hi  =  a;   v.lo  =  b;  }
        else  {  v.hi  =  b;   v.lo  =  a;  }
        if(  c>d  )  {
                if(  c>v.hi  )  v.hi  =  c;
                if(  d<v.lo  )  v.lo  =  d;
                }
        else  {
                if(  d>v.hi  )  v.hi  =  d;
                if(  c<v.lo  )  v.lo  =  c;
                }
        return(  v  );
        }

INTERVAL  vmul(  a,  b,  v  )  double  a,  b;    INTERVAL  v;  {
        return(  hilo(  a*v.hi,  a*v.lo,  b*v.hi,  b*v.lo  )  );
        }

dcheck(  v  )  INTERVAL  v;  {
        if(  v.hi  >=  0.  &&  v.lo  <=  0.  ){
                printf(  "divisor  interval  contains  0.\n"  );
                return(  1  );
                }
        return(  0  );
        }

INTERVAL  vdiv(  a,  b,  v  )  double  a,  b;    INTERVAL  v;  {
        return(  hilo(  a/v.hi,  a/v.lo,  b/v.hi,  b/v.lo  )  );
        }
```

11.14. Old Features Supported but not Encouraged

This section mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes "".

2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with yacc, since it suggests that yacc is doing a job which must be actually done by the lexical analyzer.

3. Most places where % is legal, backslash '\' may be used. In particular, \\ is the same as %%, \left the same as %left, etc.

4. There are a number of other synonyms:

```
%< is the same as %left
%> is the same as %right
%binary and %2 are the same as %nonassoc
%0 and %term are the same as %token
%= is the same as %prec
```

5. Actions may also have the form

   ```
   ={ . . . }
   ```

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

12

The curses Library: Screen-Oriented Cursor Motions

12

The curses Library: Screen-Oriented Cursor Motions

curses is a Library Package for:

- Updating a screen with reasonable optimization,
- Getting input from the terminal in a screen-oriented fashion, and
- Moving the cursor from one point to another, independent of the two previous functions.

These routines all use the termcap database to describe the capabilities of the terminal.

Overview

In making available the generalized terminal descriptions in termcap, much information was made available to the programmer, but little work was taken out of one's hands. curses helps the programmer perform the required functions, those of movement optimization and optimal screen updating, without doing any of the dirty work, and (hopefully) with nearly as much ease as is necessary to simply print or read things.

The curses package is split into three parts:

1. Screen updating without user input;
2. Screen updating with user input; and
3. Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the termcap database itself.

Terminology

In this chapter, the terminology illustrated in the table below is used with reasonable consistency.

Table 12-1 *Description of Terms*

Term	*Description*
window	An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen. Note that the term *window* is used elsewhere in the Sun system manuals when describing the window management packages for driving the bitmapped screens. curses windows bear little, if any, resemblance to the window system concepts.
terminal	Sometimes called *terminal screen*. The package's idea of what the terminal's screen currently looks like, that is, what the user sees now. This is a special screen:
screen	This is a subset of windows which are as large as the terminal screen, that is, they start at the upper left hand corner and encompass the lower right hand corner. One of these, stdscr, is automatically provided for the programmer.

Cursor Addressing Conventions

The curses library routines address positions on a screen with the y coordinate first and the x coordinate second. This follows the convention of most terminals that address the screen in row, column order. The reader should note this convention.

Compiling Things

To use the curses library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

```
#include <curses.h>
```

at the top of the program source.[33]

Also, compilations should have the following form:

```
tutorial% cc [ C-compiler options ] filename ... -lcurses -ltermcap
```

[33] The header file <curses.h> needs to include <sgtty.h>, so one should not do so oneself. The screen package also uses the Standard I/O library, so <curses.h> includes <stdio.h>. It is redundant (but harmless) to include it again.

Screen Updating

To update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named `window()` is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) coordinates of the upper left hand corner) and its size. One of these (called `curscr` for *current screen*) is a screen image of what the terminal currently looks like. Another screen (called `stdscr`, for *standard screen*) is provided by default to make changes on.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When one has a window which describes what some part the terminal should look like, the routine `refresh()` (or `wrefresh()` if the window is not `stdscr`) is called. `refresh()` makes the terminal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window *does not change the terminal.* Actual updates to the terminal screen are made only by calling `refresh()` or `wrefresh()`. This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say 'make it look like this,' and let the package worry about the best way to do this.

Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: `curscr`, which knows what the terminal looks like, and `stdscr`, which is what the programmer wants the terminal to look like next. The user should never really access `curscr` directly. Changes should be made to the appropriate screen, and then the routine `refresh()` (or `wrefresh()`) should be called.

Many functions are set up to deal with `stdscr` as a default screen. For example, to add a character to `stdscr`, one calls `addch()` with the desired character. If a different window is to be used, the routine `waddch()` (for ''window-specific'' `addch()`) is provided[34]. This convention of prepending function names with a w when they are to be applied to specific windows is consistent. The only routines which do *not* do this are those to which a window must always be specified.

[34] Actually, `addch()` is really a macro with arguments, as are most of the "functions" which deal with `stdscr` as a default.

To move the current (y, x) coordinates from one point to another, the routines move() and wmove() are provided. However, it is often desirable to first move and then perform some I/O operation. To avoid clumsiness, most I/O routines can be preceded by the prefix mv and the desired (y, x) coordinates then can be added to the arguments to the function. For example, the calls:

```
    move(y, x);
    addch(ch);
```

can be replaced by

```
    mvaddch(y, x, ch);
```

and

```
    wmove(win, y, x);
    waddch(win, ch);
```

can be replaced by

```
    mvwaddch(win, y, x, ch);
```

Note that the window description pointer (win) comes before the added (y, x) coordinates. If such pointers are needed, they are always the first parameters passed.

12.1. Variables

Many variables that describe the terminal environment are available to the programmer. They are:

Table 12-2 *Variables to Describe the Terminal Environment*

Type	*Name*	*Description*
WINDOW *	curscr	current version of the screen (terminal screen).
WINDOW *	stdscr	standard screen. Most updates are done here.
char *	Def_term	default terminal type if type cannot be determined
bool	My_term	use the terminal specification in Def_term as terminal, irrelevant of real terminal type
char *	ttytype	full name of the current terminal.
int	LINES	number of lines on the terminal
int	COLS	number of columns on the terminal
int	ERR	error flag returned by routines on a fail.
int	OK	error flag returned by routines when things go right.

There are also several `#define` constants and types which are of general usefulness:

`reg`	storage class `register` (for example, `reg int i;`)
`bool`	boolean type, actually a `char` (for example, `bool doneit;`)
`TRUE`	boolean 'true' flag (1).
`FALSE`	boolean 'false' flag (0).

12.2. Programming Curses

This is a description of how to actually use the screen package. In it, we assume all updating, reading, and so on, is applied to `stdscr`. All instructions will work on any window, by changing the function name and parameters as mentioned above.

Starting Up

To use the screen package, the routines must know about terminal characteristics, and the space for `curscr` and `stdscr` must be allocated. These functions are performed by `initscr()`. Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, `initscr()` returns ERR. `initscr()` must *always* be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either `curscr` or `stdscr` are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like `nl()` and `cbreak()` should be called after `initscr()`.

Now that the screen windows have been allocated, you can set them up for the run. If you want to, say, allow the window to scroll, use `scrollok()`. If you want the cursor to be left after the last change, use `leaveok()`. If this isn't done, `refresh()` moves the cursor to the window's current (y, x) coordinates after updating it. New windows of your own can be created, too, by using the functions `newwin()` and `subwin()`. `delwin()` gets rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables `LINES` and `COLS` to be what you want, and then call `initscr()`. This is best done before, but can be done either before or after, the first call to `initscr()`, as it always deletes any existing `stdscr` and/or `curscr` before creating new ones.

The Nitty-Gritty

Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what appears on a window are `addch()` and `move()`. `addch()` adds a character at the current (y, x) coordinates, returning ERR if it would cause the window to illegally scroll, that is, printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. `move()` changes the current (y, x) coordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into `mvaddch()` to do both things in one fell swoop.

The other output functions, such as addstr() and printw(), all call addch() to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call refresh(). To optimize finding changes, refresh() assumes that any part of the window not changed since the last refresh() of that window has not been changed on the terminal, that is, that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routines touchwin(), touchline(), and touchoverlap() are provided to make it look like the entire window has been changed, thus forcing refresh() check the whole subsection of the terminal for changes.

If you call wrefresh() with curscr, it will make the screen look like curscr thinks it looks like. This is useful for implementing a command to redraw the screen in case it get messed up.

Input

Input is essentially a mirror image of output. The complementary function to addch() is getch() which, if echo is set, calls addch() to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in *raw* or *cbreak* mode. If it is not, getch() sets it to be cbreak, reads in the character, and then resets the mode of the terminal to what it was before the call.

Miscellaneous

All sorts of functions exist for maintaining and changing information about the windows. For the most part, the descriptions in section 5.4. should suffice.

Finishing Up

To do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in getttmode() and setterm(), which are called by initscr(). To clean up after the routines, the routine endwin() is provided. It restores tty modes to what they were when initscr() was first called. Thus, anytime after the call to initscr, endwin() should be called before exiting.

12.3. Cursor Motion Optimization: Standing Alone

It is possible to use the cursor optimization functions of this screen package without the overhead and additional size of the screen updating functions. The screen updating functions are designed for uses where parts of the screen are changed, but the overall image remains the same. Certain other programs will find it difficult to use these functions in this manner without considerable unnecessary program overhead. For such applications, such as some ``crt hacks''[35] and optimizing cat(1)-type programs, all that is needed is the motion optimizations. This, therefore, is a description of what goes on at the lower levels of this screen package. The descriptions assume a certain amount of familiarity with programming problems and some finer points of C. None of it is terribly difficult, but you should be forewarned.

[35] Graphics programs designed to run on character-oriented terminals.

Terminal Information

To use a terminal's features to the best of a program's abilities, you must first know what they are. The termcap database describes these, but a certain amount of decoding is necessary, and there are, of course, both efficient and inefficient ways of reading them in. The algorithm that curses uses is taken from vi(1) and is efficient. It reads them into a set of variables whose names are two uppercase letters with some mnemonic value. For example, HO is a string which moves the cursor to the "home" position[36]. As there are two types of variables involving ttys, there are two routines. The first, gettmode(), sets some variables based upon the tty modes accessed by gtty(2) and stty(2). The second, setterm(), does a larger task by reading in the descriptions from the termcap database. This is the way these routines are used by initscr():

```
if (isatty(0)) {
        gettmode;
        if (sp=getenv("TERM"))
                setterm(sp);
}
else
        setterm(Def_term);
_puts(TI);
_puts(VS);
```

isatty() checks to see if file descriptor 0 is a terminal[37]. If it is, gettmode() sets the terminal description modes from a gtty(2). getenv() is then called to get the name of the terminal, and that value (if there is one) is passed to setterm(), which reads in the variables from termcap associated with that terminal. getenv() returns a pointer to a string containing the name of the terminal, which we save in the character pointer sp. If isatty() returns false, the default terminal Def_term is used. The TI and VS sequences initialize the terminal. _puts() is a macro which uses tputs() (see termcap(3X)) to put out a string. It is these things which endwin() undoes.

Movement Optimizations, or, Getting Over Yonder

Now that we have all this useful information, it would be nice to do something with it. The most difficult thing to do properly is motion optimization. When you consider how many different features various terminals have (tabs, backtabs, non-destructive space, home sequences, absolute tabs, . . .) you can see that deciding how to get from here to there can be a decidedly non-trivial task.

After using gettmode() and setterm() to get the terminal descriptions, the function mvcur() deals with this task. Its usage is simple: you simply tell it where you are now and where you want to go, as shown below.

[36] These names are identical to those variables used in the /etc/termcap database to describe each capability. See Appendix A for a complete list of those read, and termcap(5) for a full description.

[37] isatty() is defined in the default C library function routines. It does a gtty(2) on the file descriptor and checks the return value.

```
mvcur(0, 0, LINES/2, COLS/2)
```

would move the cursor from the home position (0, 0) to the middle of the screen. If you wish to force absolute addressing, you can use the function `tgoto()` from the `termcap`(3X) routines, or you can tell `mvcur()` that you are impossibly far away. For example, to absolutely address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

```
mvcur(0, COLS−1, LINES−1, 0)
```

12.4. Curses Functions

In the following definitions, ‘’ means that the ‘function’ is really a `#define` macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as `addch()`, it will show up as its ‘w’ counterpart. The arguments are given to show the order and type of each. Their names are not mandatory, just suggestive.

Output Functions

`addch()` and `waddch()` — Add Character to Window

```
addch(ch)
char    ch;

waddch(win, ch)
WINDOW  *win;
char    ch;
```

Add the character `ch` on the window at the current (y, x) co-ordinates. If the character is a NEWLINE (‘\n’) the line is cleared to the end, and the current (y, x) co-ordinates are changed to the beginning of the next line if newline mapping is on, or to the next line at the same x co-ordinate if it is off. A return (‘\r’) moves to the beginning of the line on the window. Tabs (‘\t’) are expanded into spaces in the normal tabstop positions of every eight characters. This returns `ERR` if it would cause the screen to scroll illegally.

`addstr()` and `waddstr()` — Add String to Window

```
addstr(st)
char    *str;

waddstr(win, str)
WINDOW  *win;
char    *str;
```

Add the string pointed to by `str` on the window at the current (y, x) coordinates. This returns `ERR` if it would cause the screen to scroll illegally. In this case, it puts on as much as it can.

box() — Draw Box Around Window

```
box(win, vert, hor)
WINDOW  *win;
char    vert, hor;
```

Draws a box around the window using vert as the character for drawing the vertical sides, and hor for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

clear() and wclear() — Reset Window

```
clear()

wclear(win)
WINDOW  *win;
```

Resets the entire window to blanks. If win is a screen, this sets the clear flag, which sends a clear-screen sequence on the next refresh() call. This also moves the current (y, x) co-ordinates to (0, 0).

clearok() — Set Clear Flag

```
clearok(scr, boolf)
WINDOW  *scr;
bool    boolf;
```

Sets the clear flag for the screen scr. If boolf is TRUE, this forces a clear-screen to be printed on the next refresh(), or stop it from doing so if boolf is FALSE. This only works on screens, and, unlike clear(), does not alter the contents of the screen. If scr is curscr, the next refresh() call causes a clear-screen, even if the window passed to refresh() is not a screen.

clrtobot() and wclrtobot() — Clear to Bottom

```
clrtobot()

wclrtobot(win)
WINDOW  *win;
```

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated mv function.

clrtoeol() and wclrtoeol() — Clear to End of Line

```
clrtoeol()

wclrtoeol(win)
WINDOW  *win;
```

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated mv function.

delch() and wdelch() — Delete Character

```
delch()

wdelch(win)
WINDOW  *win;
```

Delete the character at the current (y, x) co-ordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

deleteln() and wdeleteln() — Delete Current Line

```
deleteln()

wdeleteln(win)
WINDOW  *win;
```

Delete the current line. Every line below the current one moves up, and the bottom line becomes blank. The current (y, x) co-ordinates remains unchanged.

erase and werase() — Erase Window

```
erase()

werase(win)
WINDOW  *win;
```

Erases the window to blanks without setting the clear flag. This is analagous to `clear()`, except that it never causes a clear-screen sequence to be generated on a `refresh()`. This has no associated `mv` function.

flushok — Control Flushing of stdout

```
flushok(win, boolf)
WINDOW  *win;
bool boolf;
```

Normally, `refresh()` performs an `fflush()` on `stdout` when it is finished. `flushok()` allows you to control this. If `boolf` is TRUE (non-zero), `refresh()` performs the `fflush()`; if FALSE, `refresh()` does not.

idlok — Control Use of Insert/Delete Line

```
idlok(win, boolf)
WINDOW  *win;
bool boolf;
```

Reserved for future use. When implemented, this will signal `refresh()` as to whether it is safe to use "insert line" and "delete line" sequences to update a window.

insch() and winsch() — Insert Character

```
insch(c)
char    c;

winsch(win, c)
WINDOW  *win;
char    c;
```

Insert `c` at the current (y, x) co-ordinates Each character after it shifts to the right, and the last character disappears. This returns `ERR` if it would cause the screen to scroll illegally.

insertln() and winsertln() — Insert Line

```
insertln

winsertln(win)
WINDOW  *win;
```

Insert a line above the current one. Every line below the current line is shifted down, and the bottom line disappears. The current line becomes blank, and the current (y, x) co-ordinates remains unchanged. This returns ERR if it would cause the screen to scroll illegally.

move and wmove() — Move

```
move(y, x)
int y, x;

wmove(win, y, x)
WINDOW  *win;
int y, x;
```

Change the current (y, x) co-ordinates of the window to y, x. This returns ERR if it would cause the screen to scroll illegally.

overlay() — Overlay Windows

```
overlay(win1, win2)
WINDOW  *win1, *win2;
```

Overlay win1 on win2. The contents of win1, insofar as they fit, are placed on win2 at their starting (y, x) co-ordinates. This is done non-destructively, that is, blanks on win1 leave the contents of the space on win2 untouched.

overwrite() — Overwrite Windows

```
overwrite(win1, win2)
WINDOW  *win1, *win2;
```

Overwrite win1 on win2. The contents of win1, insofar as they fit, are placed on win2 at their starting (y, x) co-ordinates. This is done destructively, that is, blanks on win1 become blank on win2.

printw() and wprintw() — Print to Window

```
printw(fmt, arg1, arg2, ...)
char    *fmt;

wprintw(win, fmt, arg1, arg2, ...)
WINDOW  *win;
char    *fmt;
```

Performs a printf() on the window starting at the current (y, x) co-ordinates. It uses addstr() to add the string on the window. It is often advisable to use the field width options of printf() to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

refresh() and wrefresh() — Synchronize

```
refresh()

wrefresh(win)
WINDOW  *win;
```

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen to scroll illegally. In this case, it updates whatever it can without causing the scroll.

As a special case, if wrefresh() is called with the window curscr, the screen is cleared and repainted. This is useful for allowing the user to redraw the screen as needed.

standout() and wstandout() — Put Characters in Standout Mode

```
standout()

wstandout(win)
WINDOW  *win;

standend()

wstandend(win)
WINDOW  *win;
```

Start and stop putting characters onto win in standout() mode. standout() causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). standend() stops this. The sequences SO and SE (or US and UE if they are not defined) are used (see Appendix A).

Input Functions

crbreak and nocrbreak — Set or Unset from Cbreak mode

```
crbreak()

nocrbreak()
```

Set or unset the terminal to/from cbreak mode. The misnamed macros crmode() and nocrmode() are retained for backward compatibility.

echo() and noecho() — Turn Echo On or Off

```
echo()

noecho()
```

Sets the terminal to echo or not echo characters.

getch() and wgetch() — Get Character from Terminal

```
getch()

wgetch(win)
WINDOW  *win;
```

Gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If noecho() has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of noecho(), cbreak(), or rawmode set. If you do not set one, whatever routine you call to read characters sets cbreak for you, and then resets to the original mode when finished.

getstr() and wgetstr() — Get String from Terminal

```
getstr(st)
char    *str;

wgetstr(win, str)
WINDOW  *win;
char    *str;
```

Get a string through the window and put it in the location pointed to by str, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls getch() (or wgetch(win)) to get the characters needed to fill in the string until a NEWLINE or EOF is encountered. The NEWLINE is stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

raw() and noraw() — Turn Raw Mode On or Off

```
raw()

noraw()
```

Set or unset the terminal to/from raw mode. On version 7 UNIX† systems, this also turns off NEWLINE mapping (see nl()).

scanw() and wscanw() — Read String from Terminal

```
scanw(fmt, arg1, arg2, ...)
char    *fmt;

wscanw(win, fmt, arg1, arg2, ...)
WINDOW  *win;
char    *fmt;
```

Perform a scanf() through the window using fmt. It does this using consecutive getch()'s (or wgetch(win)'s). This returns ERR if it would cause the screen to scroll illegally.

Miscellaneous Functions

baudrate — Get the Baudrate

Returns the baud rate of the terminal. This is a system-dependent constant (defined in the header file <sys/tty.h>, which is included in <curses.h>).

† UNIX is a registered trademark of AT&T.

delwin() — Delete a Window

```
delwin(win)
WINDOW  *win;
```

Deletes the window from existence. All resources are freed for future use by calloc(3). If a window has a subwin() allocated window inside of it, deleting the outer window does not affect the subwindow, even though this does invalidate it. Therefore, subwindows should be deleted before their outer windows are.

endwin() — Finish up Window Routines

```
endwin()
```

Finish up window routines before exit. This restores the terminal to the state it was in before initscr() (or gettmode() and setterm()) was called. endwin() should always be called before exiting. endwin() does not itself exit — this is especially useful for resetting tty stats when trapping rubouts via signal(2).

erasechar — Get Erase Character

```
erasechar()
```

Returns the erase character for the terminal; that is, the character used by the terminal to erase single characters from the input.

getcap() — Get Termcap Capability

```
char *getcap(str)
char *str;
```

Return a pointer th the termcap capability described by str (see termcap(5) for details).

getyx() — Get Current Coordinates

```
getyx(win, y, x)
WINDOW  *win;
int y, x;
```

Puts the current (y, x) co-ordinates of win in the variables y and x. Since it is a macro, not a function, you do not pass the address of y and x.

inch() and winch() — Get Character at Current Coordinates

```
inch()

winch(win)
WINDOW  *win;
```

Returns the character at the current (y, x) co-ordinates on the given window. This does not make any changes to the window. This has no associated mv function.

initscr() — Initialize Screen Routines

```
initscr()
```

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it, none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal whose name is pointed to by Def_term (initialy dumb). If the boolean My_term is true, Def_term is always used. If the window size values for rows and columns as returned by the TIOCGWINSZ ioctl(2) request are non-zero, they are used. Otherwise, sizes are taken from the termcap description.

killchar — Get Kill Character

```
killchar()
```

Returns the terminal's line kill character; that is, the character used to erase an entire line from input.

leaveok() — Set Leave Cursor Flag

```
leaveok(win, boolf)
WINDOW  *win;
bool    boolf;
```

Sets the boolean flag for leaving the cursor after the last change. If boolf is TRUE, the cursor is left after the last update on the terminal, and the current (y, x) co-ordinates for win are changed accordingly. If it is FALSE, it is moved to the current (y, x) co-ordinates. This flag (initially FALSE) retains its value until changed by the user.

For example, say the current position is (0, 0) and we change the character at position (5, 10) in the window. After calling refresh(), the cursor is either moved to position (5, 10) (if the flag is TRUE) or the cursor is left at position (0, 0) (if the flag is FALSE).

longname() — Get Full Name of Terminal

```
longname(termbuf, name)
char    *termbuf, *name;
```

```
longname(termbuf, name)
char    *termbuf, *name;
```

Fills in name with the long (full) name of the terminal described by the termcap entry in termbuf. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable ttytype. termbuf is usually set via the termcap routine tgetent. fullname is the same as longname(), except that it gives the fullest name given in the entry, which can be quite verbose.

mvwin — Move Home Position of Window

```
mvwin(win, y, x)
WINDOW  *win;
int y, x;
```

Move the home position of the window `win` from its current starting coordinates to `y`, `x`. If that would put part or all of the window off the edge of the terminal screen, `mvwin()` returns `ERR` and does not change anything. For subwindows, `mvwin()` also returns `ERR` if you attempt to move it off its main window. If you move a main window, all subwindows are moved along with it.

newwin() — Create a New Window

```
WINDOW *
newwin(lines, cols, begin_y, begin_x)
int lines, cols, begin_y, begin_x;
```

Create a new window with `lines` lines and `cols` columns starting at position `begin_y`, `begin_x`. If either `lines` or `cols` is 0 (zero), that dimension is set to (`lines - begin_y`) or (`cols - begin_x`) respectively. Thus, to get a new window of dimensions `lines` × `cols`, use `newwin( 0, 0, 0, 0 )`.

nl() and nonl() — Turn Newline Mode On or Off

```
nl()

nonl()
```

Set or unset the terminal to/from `nl()` mode, that is, start/stop the system from mapping RETURN to NEWLINE. If the mapping is not done, `refresh()` can do more optimization, so it is recommended, but not required, that it be turned off.

scrollok — Set Scroll Flag for Window

```
scrollok(win, boolf)
WINDOW  *win;
bool    boolf;
```

Set the scroll flag for the given window. If `boolf` is `FALSE`, scrolling is not allowed. This is its default setting.

subwin() — Create a Subwindow

```
WINDOW *
subwin(win, lines, cols, begin_y, begin_x)
WINDOW  *win;
int lines, cols, begin_y, begin_x;
```

Create a new window with `lines` lines and `cols` columns starting at position (`begin_y`, `begin_x`) in the middle of the window `win`. This means that any change made to either window in the area covered by the subwindow is made on both windows. (`begin_y`, `begin_x`) are specified relative to the overall screen, not the relative (0, 0) of `win`. If either `lines` or `cols` is 0 (zero), that dimension is set to (`LINES` – `begin_y`) or (`COLS` – `begin_x`) respectively.

touchline — Indicate Line Has Been Changed

```
touchline(win, y, startx, endx)
WINDOW  *win;
int y, startx, endx;
```

This function performs a function similar to touchwin(), but on a single line. It marks the first change for the given line to be startx, if it is before the current first change mark, and the last change mark is set to be endx if it is currently less than endx.

touchoverlap — Indicate Overlapping Regions Have Been Changed

```
touchoverlap(win1, win2)
WINDOW  *win, *win2;
```

Touch the window win2 in the area which overlaps with win1. If they do not overlap, no changes are made.

touchwin() — Indicate Window Has Been Changed

```
touchwin(win)
WINDOW  *win;
```

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

unctrl() — Return Representation of Character

```
unctrl(ch)
char    ch;
```

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of ch. Control characters become their upper-case equivalents preceded by a ^ (circumflex character). Other letters stay just as they are.

Details

gettmode() — Get tty Statistics

```
gettmode()
```

Get the tty stats. This is normally called by initscr().

mvcur() — Move Cursor

```
mvcur(lasty, lastx, newy, newx)
int lasty, lastx, newy, newx;
```

Moves the terminal's cursor from lasty, lastx to newy, newx in an approximation of optimal fashion.

It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. move() and refresh() should be used to move the cursor position, so that the routines know what's going on.

scroll() — Scroll Window

```
scroll(win)
WINDOW  *win;
```

Scroll the window upward one line. This is normally not used by the user.

savetty() and resetty() — Save and Reset tty Flags

```
savetty()

resetty()
```

savetty() saves the current tty characteristic flags. resetty() restores them to what savetty() stored. These functions are performed automatically by initscr() and endwin().

setterm() — Set Terminal Characteristics

```
setterm(name)
char    *name;
```

Set the terminal characteristics to be those of the terminal named name, getting the terminal size from the TIOCGWINSZ ioctl(2) request if that size is non-zero, and otherwise from the environment. This is normally called by initscr().

tstp

```
tstp()
```

This function saves the current tty state and then puts the process to sleep. When the process gets restarted, it restores the tty state and then calls wrefresh(curscr) to redraw the screen. The initscr() function sets the signal SIGTSTP to trap to this routine.

_putchar()

```
_putchar()
```

Put out a character using the putchar() macro. This function is used to output every character that curses generates. Thus, it can be redefined by the user who wants to do non-standard things with the output. It is named with an initial '_' because it usually should be invisible to the programmer.

12.5. Capabilities from termcap

Note that the description of terminals is a difficult business, and we only attempt to summarize the capabilities here. For a full description see the termcap(5) manual pages.

Overview

Capabilities from termcap are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a LP at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed — specified by PC). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, for example, `12*` before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say '`P*`'.

Variables Set By `setterm()`

Table 12-3 *Variables Set by* `setterm()`

Type	*Name*	*Pad*	*Description*
`char *`	`AL`	P*	Add new blank Line
`bool`	`AM`		Automatic Margins
`char *`	`BC`		Back Cursor movement
`bool`	`BS`		BackSpace works
`char *`	`BT`	P	Back Tab
`bool`	`CA`		Cursor Addressable
`char *`	`CD`	P*	Clear to end of Display
`char *`	`CE`	P	Clear to End of line
`char *`	`CL`	P*	CLear screen
`char *`	`CM`	P	Cursor Motion
`char *`	`DC`	P*	Delete Character
`char *`	`DL`	P*	Delete Line sequence
`char *`	`DM`		Delete Mode (enter)
`char *`	`DO`		DOwn line sequence
`char *`	`ED`		End Delete mode
`bool`	`EO`		can Erase Overstrikes with ´ ´
`char *`	`EI`		End Insert mode
`char *`	`HO`		HOme cursor
`bool`	`HZ`		HaZeltine ˜ braindamage
`char *`	`IC`	P	Insert Character
`bool`	`IN`		Insert-Null blessing
`char *`	`IM`		enter Insert Mode (IC usually set, too)
`char *`	`IP`	P*	Pad after char Inserted using IM+IE
`char *`	`LL`		quick to Last Line, column 0
`char *`	`MA`		ctrl character MAp for cmd mode
`bool`	`MI`		can Move in Insert mode
`bool`	`NC`		No Cr: \r sends \r\n then eats \n
`char *`	`ND`		Non-Destructive space
`bool`	`OS`		OverStrike works
`char`	`PC`		Pad Character
`char *`	`SE`		Standout End (may leave space)
`char *`	`SF`	P	Scroll Forwards
`char *`	`SO`		Stand Out begin (may leave space)
`char *`	`SR`	P	Scroll in Reverse
`char *`	`TA`	P	TAb (not ^I or with padding)
`char *`	`TE`		Terminal address enable Ending sequence
`char *`	`TI`		Terminal address enable Initialization
`char *`	`UC`		Underline a single Character
`char *`	`UE`		Underline Ending sequence
`bool`	`UL`		UnderLining works even though !OS
`char *`	`UP`		UPline
`char *`	`US`		Underline Starting sequence
`char *`	`VB`		Visible Bell
`char *`	`VE`		Visual End sequence
`char *`	`VS`		Visual Start sequence
`bool`	`XN`		a Newline gets eaten after wrap

Names starting with X are reserved for severely nauseous glitches

For purposes of standout(), if SG is not 0, SO is set to NULL, and if UG is not 0, US is set to NULL. If, after this, SO is NULL, and US is not, SO is set to be US, and SE is set to be UE.

Variables Set By gettmode()

Table 12-4 *Variables Set By* gettmode()

type	*name*	*description*
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

12.6. The WINDOW structure

The WINDOW structure is defined as follows:

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 *
 *          @(#)win_st.c        6.1 (Berkeley) 4/24/86";
 */

# define  WINDOW    struct _win_st

struct _win_st {
          short              _cury, _curx;
          short              _maxy, _maxx;
          short              _begy, _begx;
          short              _flags;
          short              _ch_off;
          bool               _clear;
          bool               _leave;
          bool               _scroll;
          char               **_y;
          short              *_firstch;
          short              *_lastch;
          struct _win_st     *_nextp, *_orig;
};

# define  _ENDLINE  001
# define  _FULLWIN  002
# define  _SCROLLWIN004
# define  _FLUSH             010
# define  _FULLLINE 020
# define  _IDLINE            040
# define  _STANDOUT 0200
# define  _NOCHANGE -1
```

_cury()[38] and _curx() are the current (y, x) coordinates for the window. New characters added to the screen are added at this point. _maxy() and _maxx() are the maximum values allowed for (_cury, _curx). _begy() and _begx() are the starting (y, x) coordinates on the terminal for the window, that is, the window's home. _cury(), _curx(), _maxy(), and _maxx() are measured relative to (_begy, _begx), not the terminal's home.

_clear() tells if a clear-screen sequence is to be generated on the next refresh() call. This is only meaningful for screens. The initial clear-screen for the first refresh() call is generated by initially setting clear to be TRUE for curscr, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. _leave() is TRUE if the current (y, x) coordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. _scroll() is TRUE if scrolling is allowed.

_y() is a pointer to an array of lines which describe the terminal. Thus:

```
_y[i]
```

is a pointer to the ith line, and

```
_y[i][j]
```

is the jth character on the ith line. _flags() can have one or more values or'd into it.

For windows that are not subwindows, _orig is NULL. For subwindows, it points to the main window to which the window is subsidiary. _nextp is a pointer in a circularly linked list of all the windows which are subwindows of the same main window, plus the main window itself.

_firstch and _lastch are malloc()ed arrays which contain the index of the first and last changed characters on the line. _ch_off is the x offset for the window in the _firstch and _lastch arrays for this window. For main windows, this is always 0; for subwindows it is the difference between the starting point of the main window and that of the subwindow, so that change markers can be set relative to the main window. This makes these markers global in scope.

All subwindows share the appropriate portions of _y(), _firstch, _lastch, and _insdel with their main window.

_ENDLINE says that the end of the line for this window is also the end of a screen. _FULLWIN says that this window is a screen. _SCROLLWIN indicates that the last character of this screen is at the lower right-hand corner of the terminal; that is, if a character was put there, the terminal would scroll. _FULLLINE says that the width of a line is the same as the width of the terminal. If _FLUSH

[38] All variables not normally accessed directly by the user are named with an initial '_' to avoid conflicts with the user's variables.

is set, it says that fflush(stdout) should be called at the end of each refresh(). _STANDOUT says that all characters added to the screen are in standout mode. _INSDEL is reserved for future use, and is set by idlok(). _firstch is set to _NOCHANGE for lines on which there has been no change since the last refresh().

12.7. Example

Here is a simple example of how to use the package.

This example (twinkle) is intended to demonstrate the basic structure of a program using the screen updating sections of the package.

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It switches between patterns of asterisks, putting them on one by one in random order, and then taking them off in the same fashion.

```
# include       <curses.h>
# include       <signal.h>

/*
 * the idea for this program was a product
 * of the imagination of Kurt Schoens.  Not
 * responsible for minds lost or stolen.
 */

# define        NCOLS   80
# define        NLINES  24
# define        MAXPATTERNS     4

struct locs {
        char    y, x;
};

typedef struct locs     LOCS;

LOCS    Layout[NCOLS * NLINES]; /* current board layout */

int     Pattern,                /* current pattern number */
        Numstars;               /* number of stars in pattern */

main() {

        char            *getenv();
        int             die();

        srand(getpid());                /* initialize random sequence */

        initscr();
        signal(SIGINT, die);
        noecho();
        nonl();
        leaveok(stdscr, TRUE);
        scrollok(stdscr, FALSE);
        for (;;) {
                makeboard();            /* make the board setup */
                puton('*');             /* put on '*'s */
                puton(' ');             /* cover up with ' 's */
        }
}

/*
 * On program exit, move the cursor to the lower
 * left corner by direct addressing, since current
 * location is not guaranteed.  We lie and say we
 * used to be at the upper right corner to guarantee
 * absolute addressing.
 */
die() {

        signal(SIGINT, SIG_IGN);
        mvcur(0, COLS-1, LINES-1, 0);
        endwin();
        exit(0);
}


/*
 * Make the current board setup.  It picks a random
 * pattern and calls ison() to determine if the
 * character is on that pattern or not.
 */
makeboard() {

        reg int         y, x;
        reg LOCS        *lp;

        Pattern = rand() % MAXPATTERNS;
        lp = Layout;
        for (y = 0; y < NLINES; y++)
                for (x = 0; x < NCOLS; x++)
                        if (ison(y, x)) {
                                lp->y = y;
                                lp++->x = x;
                        }
        Numstars = lp - Layout;
}

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
reg int y, x; {

        switch (Pattern) {
          case 0:       /* alternating lines */
                return !(y & 01);
          case 1:       /* box */
                if (x >= LINES && y >= NCOLS)
                        return FALSE;
                if (y < 3 || y >= NLINES - 3)
                        return TRUE;
                return (x < 3 || x >= NCOLS - 3);
          case 2:       /* holy pattern! */
                return ((x + y) & 01);
          case 3:       /* bar across center */
                return (y >= 9 && y <= 15);
        }
        /* NOTREACHED */
}

puton(ch)
reg char        ch; {

        reg LOCS        *lp;
        reg int         r;
        reg LOCS        *end;
        LOCS            temp;

        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp;
                *lp = Layout[r];
                Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++) {
                mvaddch(lp->y, lp->x, ch);
                refresh();
        }
}
```

13

System V curses and terminfo

13

System V curses and terminfo

Screen management programs are a common component of many commercial computer applications. These programs handle input and output at a video display terminal. A screen program might move a cursor, print a menu, divide a terminal screen into windows, or draw a display on the screen to help users enter and retrieve information from a database.

This tutorial explains how to use the System V curses and terminfo libraries to write screen management programs on a SunOS system. This package includes a library of C routines, a database of terminals and terminal capabilities, and a set of SunOS system support tools. To start you writing screen management programs as soon as possible, the tutorial does not attempt to cover every part of the package. For instance, it covers only the most frequently used routines and then points you to curses(3V) and terminfo(5V) in the *SunOS Reference Manual* for more information.

Because the routines are compiled C functions, you should be familiar with the C programming language before using curses/terminfo. You should also be familiar with the C language Standard I/O library.

This chapter has five sections: The *Overview* describes curses, terminfo, and the other components of the System V terminal information utilities package.

Working with curses Routines describes the basic routines making up the curses(3V) library. It covers the routines for writing to a screen, reading from a screen, and building *windows*.[39] It also covers routines for more advanced screen management programs that draw line graphics, use a terminal's soft labels, and work with more than one terminal at the same time. Many examples are included to show the effect of using these routines.

Working with terminfo Routines describes the routines in the curses library that deal directly with the terminfo database to handle certain terminal capabilities, such as programming function keys.

Working with the terminfo Database describes the terminfo database, related support tools, and their relationship to the curses library.

curses Program Examples includes six programs that illustrate various curses routines.

[39] Here the term *windows* refers to a region within a single terminal screen.

13.1. Overview

What is curses?

curses(3V) is the library of routines that you use to write screen management programs on the SunOS system. The routines are C functions and macros; many of them resemble routines in the standard C library. For example, there's a routine printw() that behaves like printf(3V), and another named getch() that behaves like getc(3V). The automatic teller program at your bank might use printw() to print its menus and getch() to accept your requests for withdrawals (or, better yet, deposits). A visual screen editor like the SunOS screen editor vi(1) might also use these and other curses routines.

The curses library is located in the file /usr/5lib/libcurses.a. To compile a program using routines in this library, you must use the System V optional /usr/5bin/cc(1V) command, and include the -lcurses on the command line so that the link editor can locate and load them:

```
/usr/5bin/cc file.c -lcurses -o file
```

The name curses comes from the cursor optimization that this library of routines provides. Cursor optimization minimizes the amount a cursor has to move around a screen to update it. For example, if you had designed a screen editor program with curses routines and edited the sentence

```
curses/terminfo is a great package for creating screens.
```

to read

```
curses/terminfo is the best package for creating screens.
```

the program would output only the string 'thebest in place of '.agreat The other characters would be preserved. Because the amount of data transmitted—the output—is minimized, cursor optimization is also referred to as output optimization.

Cursor optimization takes care of updating the screen in a manner appropriate for the terminal on which a curses program is run. This means that the curses library can do what is required to update any of a large number of different terminal types. It searches the terminfo database (described below) to find the correct description for a terminal.

How does cursor optimization help you and those who use your programs? First, it saves you time in describing in a program how you want to update screens. Second, it saves a user's time when the screen is updated. Third, it reduces the load on your system. Fourth, it handles a large variety of terminals on which your program might be run.

Here's a simple curses program. It uses some of the basic curses routines to move a cursor to the middle of a screen and print the character string BullsEye. Each of these routines is described in the section *Working with curses Routines* later in this chapter. For now, just look at their names below and you will get an idea of what each of them does.

Figure 13-1 *A Simple* curses Program

```
#include <curses.h>

main()
{
   initscr();

   move( LINES/2 - 1, COLS/2 - 4 );
   addstr("Bulls");
   refresh();
   addstr("Eye");
   refresh();
   endwin();
}
```

What is terminfo?

terminfo refers to both of the following:

Terminfo Routines

This is a group of routines within the curses library for handling certain terminal capabilities. You can use these routines to program function keys (if your terminal has programmable keys), or write filters, for example. Shell programmers, as well as C programmers, can use the terminfo routines in their programs.

Terminfo Database

This is a database containing the descriptions of many terminals that can be used with curses programs. These descriptions specify the capabilities of a terminal and the way it performs various operations—for example, how many lines and columns it has and how its control characters are interpreted.

Each terminal description in the database is a separate, compiled file. You use the source code that terminfo(5V) describes to create these files and the command tic(8V) to compile them.

The compiled files are normally located in the directories /usr/share/lib/terminfo/?. These directories have single character names, each of which is the first character in the name of a terminal. For example, an entry for a virtual terminal emulator is normally located in the file /usr/share/lib/terminfo/v/virtual.

Here is a simple shell script that uses the `terminfo` database.

Figure 13-2 *A Shell Script Using* `terminfo` *Routines*

```
#   Clear the screen and show the 0,0 position.
#
tput clear
tput cup 0 0            # or tput home
echo "<- this is 0 0"
#
#   Show the 5,10 position.
#
tput cup 5 10
echo "<- this is 5 10"
```

How `curses` and `terminfo` Work Together

A screen management program with `curses` routines refers to the `terminfo` database at run time to obtain the information it needs about the terminal being used.

For example, suppose you are using a virtual terminal emulator to display the simple ''BullsEye'' program shown above. To execute properly, the program needs to know how many lines and columns the terminal screen has, in order to print the `BullsEye` in the middle of it. The description of the `ansi` terminal type in the `terminfo` database contains these values. All the `curses` program needs to know beforehand is the name of the terminal type. This is generally set automatically when you log in.

Other Components of the Terminal Information Utilities Package

Here is a complete list of the components discussed in this tutorial:

`captoinfo`(8V)
: a tool for converting terminal descriptions developed on earlier releases of the SunOS system to `terminfo` descriptions

`curses`(3V)
: the `curses` library

`infocmp`(8V)
: a tool for printing and comparing compiled terminal descriptions

`tabs`(1V)
: a tool for setting non-standard tab stops

`terminfo`(5V)
: the System V terminal information database

`tic`(8V)
: a tool for compiling terminal descriptions for the `terminfo` database

`tput`(1V)
: a tool for initializing the tab stops on a terminal and for outputting the value of a terminal capability

13.2. Working with curses Routines

This section describes the basic curses routines for creating interactive screen management programs. It begins by describing the routines and other program components that every curses program needs to work properly. Then it tells you how to compile and run a curses program. Finally, it describes the most frequently used curses routines that

- □ write output to and read input from a terminal screen
- □ control the data output and input — for example, to print output in bold type or prevent it from echoing (printing back on a screen)
- □ manipulate multiple screen images (windows)
- □ draw simple graphics
- □ manipulate soft labels on a terminal screen
- □ send output to and accept input from more than one terminal.

To illustrate the effect of using these routines, we include simple example programs as the routines are introduced. We also refer to a group of larger examples located in the section curses *Program Examples* in this chapter. These larger examples are more challenging; some make use of routines not discussed here.

What Every curses Program Needs

All curses programs need to include the header file <curses.h> and call the routines initscr(), refresh() or similar related routines, and endwin().

The Header File <curses.h>

The header file <curses.h> defines several global variables and data structures and defines several curses routines as macros.

To begin, let's consider the variables and data structures defined. <curses.h> defines all the parameters used by curses routines. It also defines the integer variables LINES and COLS; when a curses program is run on a particular terminal, these variables are assigned the vertical and horizontal dimensions of the terminal screen, respectively, by the routine initscr() described below. The header file defines the constants OK and ERR, too. Most curses routines have return values; the OK value is returned if a routine is properly completed, and the ERR value if some error occurs.

LINES and COLS are external (global) variables that represent the size of a terminal screen. The environment variables, LINES and COLUMNS, may be set in a user's shell environment; a curses program uses the environment variables to determine the size of a screen.

For more information about these variables, see *The Routines* initscr(), refresh(), *and* endwin() and *More about* initscr() *and Lines and Columns*, below.

Now let's consider the macro definitions. The <curses.h> header file defines many curses routines as macros that call (other macros or) curses routines. The line

```
#define refresh() wrefresh(stdscr)
```

shows when refresh is called, it is expanded to call the curses routine `wrefresh()`. The latter routine, in turn, calls the two `curses` routines `wnoutrefresh()` and `doupdate()`. Many other macros also combine two or three routines together to achieve a particular result.

Macro expansion in `curses` programs may cause problems with certain sophisticated C features, such as the use of automatic incrementing variables.

One final point about `<curses.h>`: it automatically includes `<stdio.h>` and the `<termio.h>`, terminal driver interface file. Including either file again in a program is redundant, but harmless.

The Routines `initscr()`, `refresh()`, and `endwin()`

The routines `initscr()`, `refresh()`, and `endwin()` initialize a terminal screen to an "in `curses` state," update the contents of the screen, and restore the terminal to an "out of `curses` state," respectively. Use the simple program that we introduced earlier to learn about each of these routines:

Figure 13-3 `initscr()`, `refresh()`, *and* `endwin()` *in a Program*

```
#include <curses.h>

main()
{
   initscr();      /* initialize terminal settings and <curses.h>
                      data structures and variables */

   move( LINES/2 - 1, COLS/2 - 4 );
   addstr("Bulls");
   refresh();      /* send output to (update) terminal screen */
   addstr("Eye");
   refresh();      /* send more output to terminal screen */
   endwin();       /* restore all terminal settings */
}
```

A `curses` program usually starts by calling `initscr()`; the program should call `initscr()` only once. Using the environment variable `TERM` as the section *How* `curses` *and* `terminfo` *Work Together* describes, this routine determines what terminal is being used. It then initializes all the declared data structures and other variables from `<curses.h>`. For example, `initscr()` would initialize `LINES` and `COLS` for the sample program on whatever terminal it was run. If a virtual terminal emulator were to be used, this routine would initialize `LINES` to 24 and `COLS` to 80. Finally, this routine writes error messages to `stderr` and exits if errors occur.

During the execution of the program, output and input is handled by routines like `move()` and `addstr()` in the sample program. For example,

```
move( LINES/2 - 1, COLS/2 - 4 );
```

says to move the cursor to the left of the middle of the screen. Then the line

```
addstr("Bulls");
```

says to write the character string `Bulls`. With a virtual terminal, these routines would position the cursor and write the character string at (11,36).

All curses routines that move the cursor move it from its home position in the upper left corner of a screen. The (LINES,COLS) coordinate at this position is (0,0) not (1,1). Notice that the vertical coordinate is given first and the horizontal second, which is the opposite of the more common 'x,y' order of screen (or graph) coordinates.

The –1 in the sample program takes the (0,0) position into account to place the cursor on the center line of the terminal screen.

Routines like move() and addstr() do not actually change a physical terminal screen when they are called. The screen is updated only when refresh() is called. Before this, an internal representation of the screen called a *window* is updated. This is a very important concept, which we discuss below under *More about* refresh() *and Windows.*

Finally, a curses program ends by calling endwin(). This routine restores all terminal settings and positions the cursor at the lower left corner of the screen.

Compiling a curses Program

You compile programs that include curses routines as C language programs using the /usr/5bin/cc command, which invokes the C compiler.

The routines are stored in the library /usr/5lib/libcurses.a. To direct the link editor to search this library, you must use the –l option with the cc command.

The general command line for compiling a curses program follows:

```
/usr/5bin/cc file.c -lcurses -o file
```

file.c is the name of the source program; and *file* is the resulting executable program.

More about initscr() and Lines and Columns

After determining a terminal's screen dimensions, initscr() sets the variables LINES and COLS. These variables are set from the terminfo variables lines and columns. These, in turn, are set from the values in the terminfo database, unless overridden by the window size obtained by the TIOCGWINSZ ioctl(2) request. If that size is zero, the values of the environment variables LINES and COLUMNS are used.

More about refresh() and Windows

As mentioned above, curses routines do not update a terminal until refresh() is called. Instead, they write to an internal representation of the screen called a *window*. When refresh() is called, the accumulated output is sent from the window to the current terminal screen.

A window acts a lot like the buffer used by vi(1). When you invoke vi to edit a file, the changes you make to the contents of the file are reflected in the buffer. The changes become part of the permanent file only when you use the w or ZZ command. Similarly, when you invoke a screen program made up of curses routines, they change the contents of a window. The changes become part of the current terminal screen only when refresh() is called.

<curses.h> supplies a default window named stdscr (standard screen), which is the size of the current terminal's screen, for all programs using curses routines. The header file defines stdscr to be of the type WINDOW*, a pointer to a C structure which you can think of as a two-dimensional array of characters representing a terminal screen. The program always keeps track of what is on the physical screen, as well as what is in stdscr. When refresh() is called, it compares the two screen images and sends a stream of characters to the terminal that make the current screen look like stdscr. A curses program considers

many different ways to do this, taking into account the various capabilities of the terminal, and similarities between what is on the screen and what is on the window. It optimizes output by printing as few characters as is possible. The following figure illustrates what happens when you execute the ''BullsEye'' curses program.

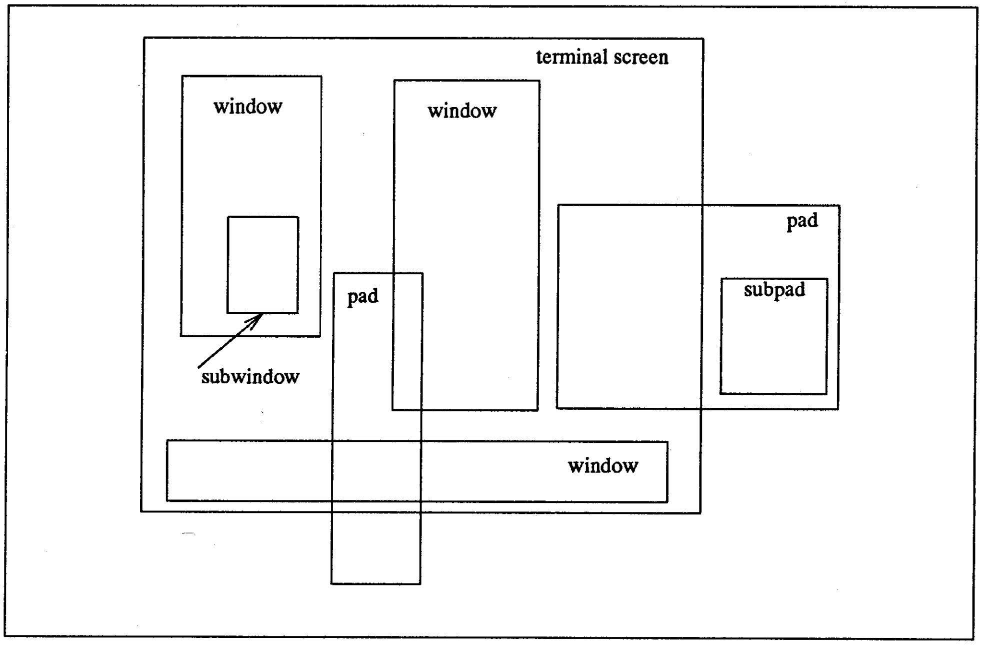
You can create other windows and use them instead of `stdscr`. Windows are useful for maintaining several different screen images. For example, many data entry and retrieval applications use two windows: one to control input and output and one to print error messages that don't mess up the other window.

It is possible to subdivide a screen into many windows, refreshing each one of them as desired. When windows overlap, the contents of the current screen show the most recently refreshed window. It is also possible to create a window within a window; the smaller window is called a subwindow. Assume that you are designing an application that uses forms, for example, an expense voucher, as a user interface. You could use subwindows to control access to certain fields on the form.

Some `curses` routines are designed to work with a special type of window called a *pad*. A pad is a window whose size is not restricted by the size of a screen or associated with a particular part of a screen. You can use a pad when you have a particularly large window or only need part of the window on the screen at any one time. For example, you might use a pad for an application with a spread sheet.

The illustration below represents what a pad, a subwindow, and some other windows might look like in comparison to a terminal screen.

Figure 13-4 *Multiple Windows and Pads Mapped to a Terminal Screen*



The section *Building Windows and Pads*, later in this chapter, describes the routines you use to create and use them.

Simple Output and Input

Output

The routines that curses provides for writing to stdscr are similar to those provided by the stdio(3V) library for writing to a file. They let you:

- □ write a character at a time — addch()
- □ write a string — addstr()
- □ format a string from a variety of input arguments — printw()
- □ move a cursor or move a cursor and print character(s) — move(), mvaddch(), mvaddstr(), mvprintw()
- □ clear a screen or a part of it — clear(), erase(), clrtoeol(),-clrtobot()

Following are descriptions and examples of these routines.

The curses library provides its own set of output and input functions. You should not use other I/O routines or system calls, like read(2) and write(2), in a curses program. They may cause undesirable results when you run the program.

addch() — Write a single character to stdscr

```
#include <curses.h>
int addch(ch)
chtype ch;
```

addch() is a macro that writes a single character to stdscr. The character is of the type chtype, which is defined in <curses.h>. chtype contains both data and attributes (see *Output Attributes* in this chapter for information about attributes); when working with variables of this type, make sure you declare them as chtype, and not as the underlying data type (for example, short) of chtype. This will ensure future compatibility.

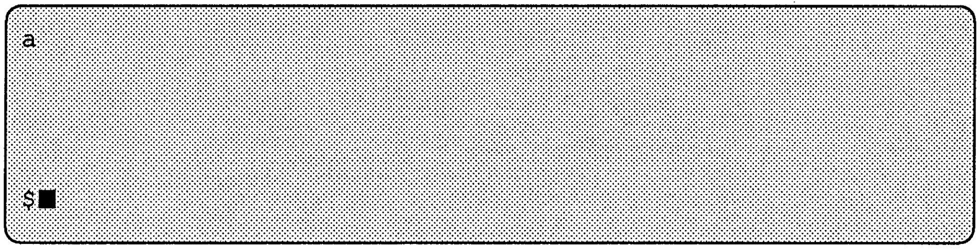
addch() does some character translations. For example, it maps the NEWLINE character to a clear-to-end-of-line, and moves the cursor to the next line. It maps the TAB character to an appropriate number of blanks. It maps other control characters to the appropriate '^*X*' notation.

addch() normally returns OK. The only time addch() returns ERR is after adding a character to the lower right-hand corner of a window that does not scroll.

Example:

```
#include <curses.h>

main()
{
   initscr();
   addch('a');
   refresh();
   endwin();
}
```

produces:



```
a





$■
```

Also see the show program under curses *Example Programs* later in this chapter.

addstr() — write a string of characters to stdscr

```
#include <curses.h>

int addstr(str)
char *str;
```

addstr() is a macro that follows the same translation rules as addch(); it calls addch() to write each character. addstr() returns OK on success and ERR on error.

For an example, refer to the "BullsEye" program, above.

printw() — formatted printing on stdscr

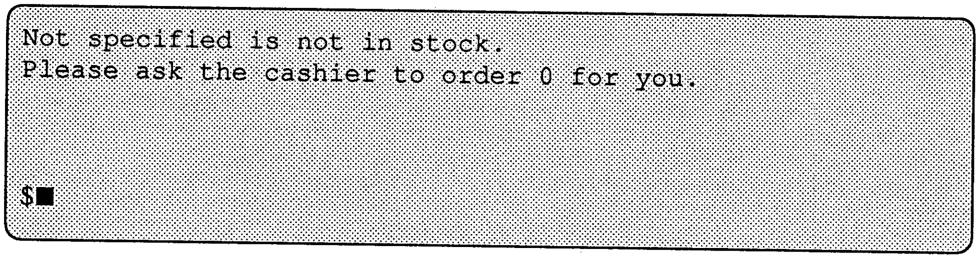
```
#include <curses.h>

int printw(fmt [,arg...])
char *fmt
```

Like printf, printw() takes a format string and a variable number of arguments. Like addstr(), printw() calls addch() to write the string. printw() returns OK on success and ERR on error.

Example:

```
#include <curses.h>

main()
{
   char* title = "Not specified";
   int no = 0;

   initscr();
   printw("%s is not in stock.\n", title);
   printw("Please ask the cashier to order %d for you.\n", no);
   refresh();
   endwin();
}
```

produces:



```
Not specified is not in stock.
Please ask the cashier to order 0 for you.



$■
```

move() — position the cursor for stdscr

```
#include <curses.h>

int move(y, x);
int y, x;
```

move() positions the cursor for stdscr at the given row y and the given column x.

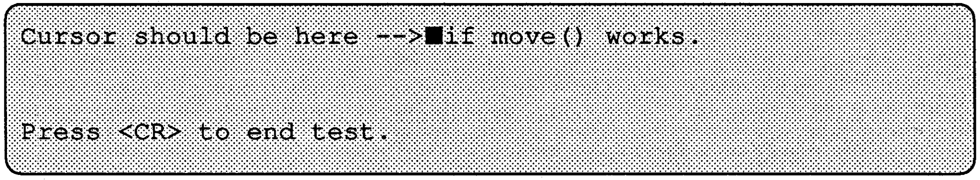
Notice that move() takes the y coordinate *before* the x coordinate. The upper left-hand coordinates for stdscr are (0,0), the lower right-hand (LINES – 1, COLS – 1). See the section initscr(), refresh(), *and* endwin() for more information.

move() returns OK on success and ERR on error. Trying to move to a screen position of less than (0,0) or more than (LINES - 1, COLS - 1) causes an error.

Example:

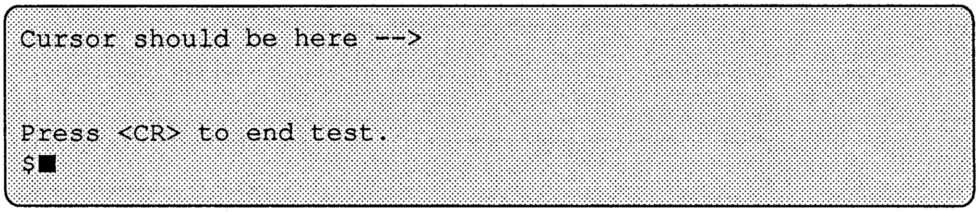
```
#include <curses.h>

main()
{
   initscr();
   addstr("Cursor should be here --> if move() works.");
   printw("\n\n\nPress <CR> to end test.");
   move(0,25);
   refresh();
   getch();      /* Gets <CR>; discussed below. */
   endwin();
}
```

produces:



```
Cursor should be here -->■if move() works.



Press <CR> to end test.
```

After you press RETURN, the screen looks like:



```
Cursor should be here -->



Press <CR> to end test.
$■
```

See the `scatter` program under `curses` *Program Examples* in this chapter for another example.

`mvaddch` — move and print a character

```
#include <curses.h>

int mvaddch( y, x, ch )
```

`mvaddch()` is a macro that moves the cursor to a given position and prints a character.

`mvaddstr` — move and print a string

```
#include <curses.h>

int mvaddstr( y, x, str )
```

`mvaddstr()` is a macro that moves the cursor to a given position and prints a string of characters.

mvprintw — move and print a formatted string

```
#include <curses.h>

int mvprintw( y, x, fmt [,arg]... )
```

mvprintw() is a macro that moves the cursor to a given position and prints a formatted string. of using move().

clear() and erase() — clear the screen

```
#include <curses.h>

int clear()
int erase()
```

clear() and erase() are macros that convert stdscr to all blanks. clear() assumes that the screen may have garbage that it doesn't know about; it first calls erase() and then clearok(), which clears the physical screen completely on the next call to refresh(). initscr() automatically calls clear().

clear() always returns OK; erase() returns no useful value.

clrtoeol() and clrtobot() — partial screen clears

```
#include <curses.h>

int clrtoeol()
int clrtobot()
```

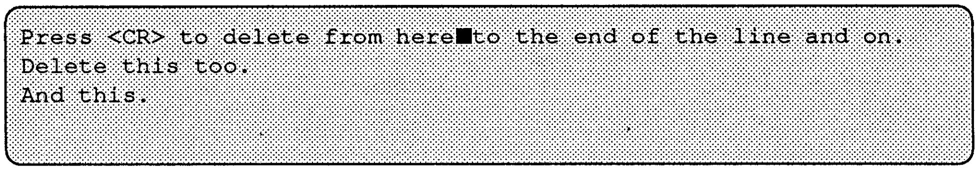
clrtoeol() and clrtobot() are macros that clear a portion of the screen. clrtoeol() changes the remainder of a line to all blanks. clrtobot() changes the remainder of a screen to all blanks. Both start with the current cursor position inclusive.

Neither returns any useful value.

Example:

```
#include <curses.h>

main()
{
    initscr();
    addstr("Press <CR> to delete from here to the end of the line and on.");
    addstr("\nDelete this too.\nAnd this.");
    move(0,30);
    refresh();
    getch();
    clrtobot();
    refresh();
    endwin();
}
```

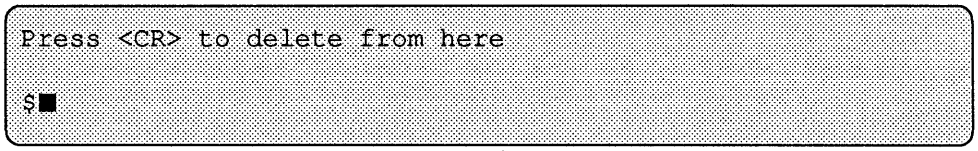
produces:



```
Press <CR> to delete from here■to the end of the line and on.
Delete this too.
And this.
```

Notice the two calls to `refresh()`: one to send the full screen of text to a terminal, the other to clear from the position indicated to the bottom of a screen.

Here's what the screen looks like when you press [RETURN]:



```
Press <CR> to delete from here

$■
```

See the `show` and `two` programs under `curses` *Example Programs* for examples of `clrtoeol()`.

Input

`curses` routines for reading from the current terminal are similar to those provided by the `stdio`(3V) library for reading from a file. They let you

- read a character at a time — `getch()`
- read a [NEWLINE]-terminated string — `getstr()`
- parse input, converting and assigning selected data to an argument list — `scanw()`

The primary routine is `getch()`, which processes a single input character and then returns that character. This routine is like the C library routine `getchar()`(3V) except that it makes several terminal- or system-dependent options available that are not possible with `getchar()`. For example, you can use `getch()` with the `curses` routine `keypad()`, which allows a `curses` program to interpret extra keys on a user's terminal, such as arrow keys, function keys, and other special keys that transmit escape sequences, and treat them as just another key.

`getch()` — read a single character from the current terminal

```
#include <curses.h>

int getch()
```

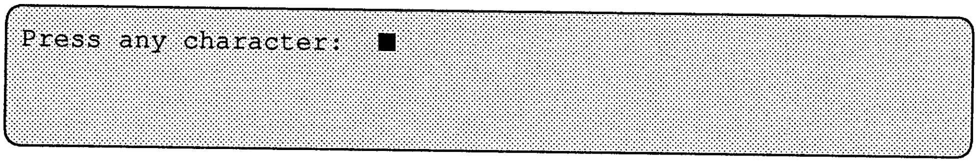
`getch()` is a macro that returns the value of the character or `ERR` on 'end of file', receipt of signals, or non-blocking read with no input.

See the discussions about `echo()`, `noecho()`, `cbreak()`, `nocbreak()`, `raw()`, `noraw()`, `halfdelay()`, `nodelay()`, and `keypad()` below.

Example:

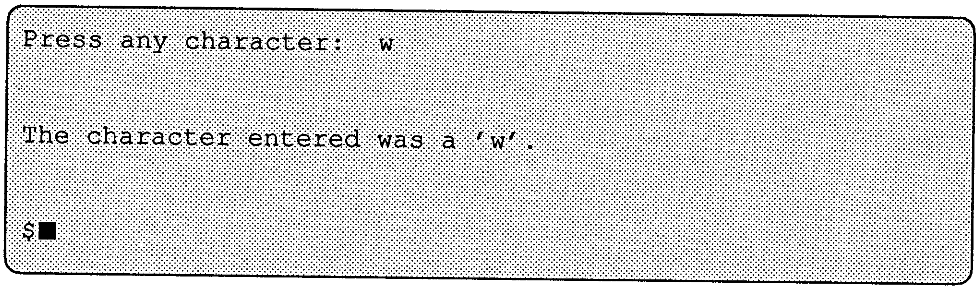
```
#include <curses.h>

main()
{
    int ch;

    initscr();
    cbreak();           /* Explained later in the section "Input Options" */
    addstr("Press any character:  ");
    refresh();
    ch = getch();
    printw("\n\n\nThe character entered was a '%c'.\n", ch);
    refresh();
    endwin();
}
```

The first refresh() sends the addstr() character string from stdscr to the terminal:



```
Press any character:  ■
```

Then assume that a w is typed at the keyboard. getch() accepts the character and assigns it to ch. Finally, the second refresh() is called:



```
Press any character:  w


The character entered was a 'w'.


$■
```

For another example of getch(), see the show program under curses *Example Programs*.

getstr() — read character string into a buffer

```
#include <curses.h>

int getstr(str)
char *str;
```

getstr() is a macro that calls getch() to read a string of characters into a buffer, until a RETURN, NEWLINE, or ENTER key is received from stdscr. getstr() does not check for buffer overflow.

getstr() returns ERR if getch() returns ERR; otherwise it returns OK.

See the discussions about echo(), noecho(), cbreak(), nocbreak(), raw(), noraw(), halfdelay(), nodelay(), and keypad() below.

Example:

```
#include <curses.h>

main()
{
char str[256];

   initscr();
   cbreak();      /* Explained later in the section "Input Options" */
   addstr("Enter a character string terminated by <CR>:\n\n");
   refresh()
   getstr(str);
   printw("\n\n\nThe string entered was \n‘%s’\n", str);
   refresh();
   endwin();
}
```

If you enter the string 'I enjoy learning about the SunOS system', the final screen (after entering RETURN) would appear as:

```
Enter a character string terminated by <CR>:

I enjoy learning about the SunOS system


The string entered was
‘I enjoy learning about the SunOS system.’

$■
```

scanw() — formatted input conversion

```
#include <curses.h>

int scanw(fmt [, arg...])
char *fmt;
```

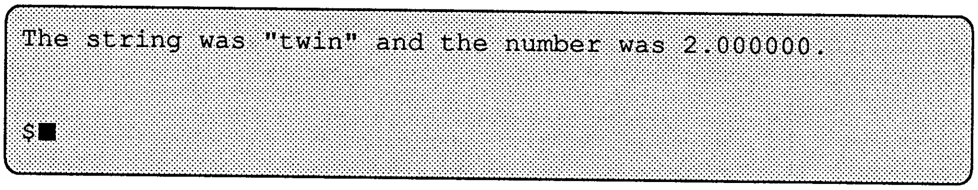
Like scanf(3V), scanw() uses a format string to convert input words and assign them to a variable number of arguments. scanw() returns the same values as scanf()

See scanf(3V) for more information.

Example:

```
#include <curses.h>

main()
{
   char string[100];
   float number;

   initscr();
   cbreak();           /* Explained later in the  */
   echo();             /* section "Input Options" */
   addstr("Enter a number and a string separated by a comma: ");
   refresh();
   scanw("%f,%s",&number,string);
   clear();
   printw("The string was \"%s\" and the number was %f.",string,number);
   refresh();
   endwin();
}
```

Notice the two calls to refresh(). The first call updates the screen with the character string passed to addstr(), the second with the string returned from scanw(). Also notice the call to clear(). Assume you entered the following when prompted: 2,twin. After running this program, your terminal screen would appear, as follows:



```
The string was "twin" and the number was 2.000000.


$■
```

Controlling Output and Input

Output Attributes

When we talked about addch(), we said that it writes a single character of the type chtype to stdscr. chtype has two parts: a part with information about the character itself, and another part with information about a set of attributes associated with the character. These attributes allow a character to be printed in reverse video, bold, underlined, and so on.

stdscr always has a set of current attributes that it associates with each character as it is written. However, using the routine attrset() and the related curses routines described below, you can change the current attributes. Below is a list of the attributes and what they mean.

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, a curses program attempts to find a substitute attribute. If none is possible, the attribute is ignored.

A_BLINK	blinking
A_BOLD	extra bright or bold
A_DIM	half bright
A_REVERSE	reverse video
A_STANDOUT	a terminal's best highlighting mode
A_UNDERLINE	underlining
A_ALTCHARSET	alternate character set

(See the section *Drawing Lines and Other Graphics*, below, for more information about these attributes.)

To use these attributes, you must pass them as arguments to attrset() and related routines; they can also be OR'ed with the bitwise OR (|) to addch().

Let's consider a use of one of these attributes. To display a word in bold, use the following code:

```
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
refresh();
```

Attributes can be turned on singly, such as attrset(A_BOLD) in the example, or in combination. To turn on blinking bold text, for example, you would use attrset(A_BLINK | A_BOLD). Individual attributes can be turned on and off with the curses routines attron() and attroff() without affecting other attributes. attrset(0) turns all attributes off.

Notice the attribute called A_STANDOUT. You might use it to make text attract the attention of a user. The particular hardware attribute used for standout is the most visually pleasing attribute a terminal has. Standout is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or reverse video, but instead just need to highlight some text. For such applications, the A_STANDOUT attribute is recommended. Two convenient functions, standout() and standend() can be used to turn on and off this attribute. standend(), in fact, turns off all attributes.

Bit Masks

In addition to the attributes listed above, there are two bit masks called A_CHARTEXT and A_ATTRIBUTES. You can use these bit masks with the curses function inch() and the C logical AND (&) operator to extract the character or attributes of a position on a terminal screen. See the discussion of inch() for more information.

Following are descriptions of attrset() and the other curses routines that you can use to manipulate attributes.

attron(), attrset(), and attroff() — set or modify attributes

```
#include <curses.h>

int attron( attrs )
chtype attrs;

int attrset( attrs )
chtype attrs;

int attroff( attrs )
chtype attrs;
```

attron() turns on the requested attribute attrs in addition to any that are currently on. *Attrs* is of the type chtype and is defined in <curses.h>.

attrset() turns on the requested attributes attrs instead of any that are currently turned on.

attroff() turns off the requested attributes, *attrs*, if they are on.

Attributes may be combined using the bitwise OR (|).

All return OK.

Example:
See the highlight program under curses *Example Programs*, below.

standout() and standend() — highlight with preferred attribute

```
#include <curses.h>

int standout()
int standend()
```

standout() turns on the preferred highlighting attribute, A_STANDOUT, for the current terminal. This routine is equivalent to attron(A_STANDOUT).

standend() turns off all attributes. This routine is equivalent to attrset(0).

Both always return OK.

Example:
See the highlight program under curses *Example Programs*, below.

Bells, Whistles, and Flashing Lights

Occasionally, you may want to get a user's attention. Two curses routines were designed to help you do this. They let you ring the terminal's bell and flash its screen.

flash() flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within ear shot of the user. The routine beep() can be called when an audible bell is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to beep() will flash the screen.)

beep() and flash() — ring bell or flash screen

```
#include <curses.h>

int flash()
int beep()
```

flash() tries to flash the terminal screen, if possible, otherwise it tries to ring the terminal bell.

beep() tries to ring the terminal bell, if possible, and, if not, tries to flash the terminal screen.

Neither returns any useful value.

Input Options

The SunOS system does a considerable amount of processing on input before an application ever sees a character; amongst other things, it:

- echoes (prints back) characters to a terminal as they are typed
- interprets an erase character, typically DELETE and a line kill character, typically
  CTRL-U (control-U)
- interprets a CTRL-D as end-of-file (EOF) character.
- interprets interrupt and quit characters
- strips the character's parity bit
- translates RETURN characters to NEWLINEs.

Because a curses program maintains total control over the screen, curses turns off echoing; it does the echoing itself. For an interactive screen, you may not want the system to process characters in the standard way. Some curses routines, noecho() and cbreak(), for example, have been designed so that you can alter the standard character processing. Using these routines in an application controls how input is interpreted.

Every curses program accepting input should set some input options so that when the program starts running, the terminal on which it runs will be in cbreak(), raw(), nocbreak(), or noraw() mode. Although the curses program starts up in echo() mode, as shown below, none of the other modes are guaranteed.

The combination of noecho() and cbreak() is most common in interactive screen management programs. Suppose, for instance, that you don't want the characters sent to your application program to be echoed wherever the cursor currently happens to be; instead, you want them echoed at the bottom of the screen. The curses routine noecho() is designed for this purpose. However, when noecho() turns off echoing, normal erase and kill processing is still on. Using the routine cbreak() causes these characters to be uninterpreted.

Figure 13-5 *Input Option Settings for* curses Programs

Input Options	Characters	
	Interpreted	Uninterpreted
Normal 'out of curses state'	interrupt, quit stripping <CR> to <NL> echoing erase, kill EOF	
Normal curses 'start up state'	echoing (simulated)	All else undefined.
cbreak() and echo()	interrupt, quit stripping echoing	erase, kill EOF
cbreak() and noecho()	interrupt, quit stripping	echoing erase, kill EOF
nocbreak() and noecho()	break, quit stripping erase, kill EOF	echoing
nocbreak() and echo()	See caution below.	
nl()	<CR> to <NL>	
nonl()		<CR> to <NL>
raw() (instead of cbreak())		break, quit stripping

Do not use the combination `nocbreak()` and `noecho()`. If you use it in a program and also use `getch()`, the program will go in and out of `cbreak()` mode to get each character. Depending on the state of the terminal driver when each character is typed, the program may produce undesirable output.

In addition to the routines noted above, you can use the `curses` routines `noraw()`, `halfdelay()`, and `nodelay()` to control input. These routines are described in `curses`(3V).

echo() and noecho() — turn echoing on and off

```
#include <curses.h>

int echo()
int noecho()
```

echo() turns on echoing of characters by curses as they are read in. This is the initial setting.

noecho() turns off the echoing.

Neither returns any useful value.

curses programs may not run properly if you turn on echoing with nocbreak(). After you turn echoing off, you can still echo characters with addch().

Examples:
See the editor and show programs under curses *Program Examples*, below.

cbreak() and nocbreak() — turn "break for each character" on or off

```
#include < curses.h >
int cbreak()
int nocbreak()
```

cbreak() turns on 'break for each character' processing. A program gets each character as soon as it is typed, but the erase, line kill, and CTRL-D characters are not interpreted.

nocbreak() returns to normal 'line at a time' processing. This is typically the initial setting.

Neither returns any useful value.

A curses program may not run properly if cbreak() is turned on and off within the same program or if the combination nocbreak() and echo() is used.

Example:
See the editor and show programs under curses *Program Examples.*

Building Windows and Pads

The section above entitled *More about* refresh() *and Windows* explained what windows and pads are and why you might want to use them. This section describes the curses routines you use to manipulate and create windows and pads.

Window Output and Input

The routines that you use to send output to and get input from windows and pads are similar to those you use with stdscr. The only difference is that you have to give the name of the window to receive the action. Generally, these functions have names formed by putting the letter w at the beginning of the name of a stdscr routine and adding the window name as the first parameter. For example, addch('c') would become waddch(mywin, ´c´) if you wanted to write the character c to the window mywin. Here's a list of the window (or w) versions of the output routines discussed in *Getting Simple Output and Input.*

```
waddch(win, ch)
mvwaddch(win, y, x, ch)
waddstr(win, str)
mvwaddstr(win, y, x, str)
wprintw(win, fmt [, arg ...])
mvwprintw(win, y, x, fmt [, arg ...])
wmove(win, y, x)
wclear(win) and werase(win)
wclrtoeol(win) and wclrtobot(win)
wrefresh()
```

You can see from their declarations that these routines differ from the versions that manipulate `stdscr` only in their names and the addition of a *win* argument. Notice that the routines whose names begin with `mvw` take the *win* argument before the *y, x* coordinates, which is contrary to what the names imply. See `curses`(3V) for more information about these routines, or the versions of the input routines `getch`, `getstr()`, and so on that you should use with windows.

All `w` routines can be used with pads except for `wrefresh()` and `wnoutrefresh()`. In place of these two routines, you have to use `prefresh()` and `pnoutrefresh()` with pads.

The Routines `wnoutrefresh()` and `doupdate()`

If you recall from the earlier discussion about `refresh()`, we said that it sends the output from `stdscr` to the terminal screen. We also said that it was a macro that expands to `wrefresh(stdscr)` (see *What Every* `curses` *Program Needs* and *More about* `refresh()` *and Windows*).

The `wrefresh()` routine is used to send the contents of a window (`stdscr` or one that you create) to a screen; it calls the routines `wnoutrefresh()` and `doupdate()`. Similarly, `prefresh()` sends the contents of a pad to a screen by calling `pnoutrefresh()` and `doupdate()`.

Using `wnoutrefresh()`—or `pnoutrefresh()` (this discussion will be limited to the former routine for simplicity)—and `doupdate()`, you can update terminal screens with more efficiency than using `wrefresh()` by itself. `wrefresh()` works by first calling `wnoutrefresh()`, which copies the named window to a data structure referred to as the virtual screen. The virtual screen contains what a program intends to display at a terminal. After calling `wnoutrefresh()`, `wrefresh()` then calls `doupdate()`, which compares the virtual screen to the physical screen and does the actual update. If you want to output several windows at once, calling `wrefresh()` will result in alternating calls to `wnoutrefresh()` and `doupdate()`, causing several bursts of output to a screen. However, by calling `wnoutrefresh()` for each window and then `doupdate()` only once, you can minimize the total number of characters transmitted and the processor time used. The sample program below uses only one `doupdate()`.

```
#include <curses.h>

main()
{
   WINDOW *w1, *w2;

   initscr();
   w1 = newwin(2,6,0,3);
   w2 = newwin(1,4,5,4);
   waddstr(w1, "Bulls");
   wnoutrefresh(w1);
   waddstr(w2, "Eye");
   wnoutrefresh(w2);
   doupdate();
   endwin();
}
```

Notice from the sample that you declare a new window at the beginning of a `curses` program. The lines

```
w1 = newwin(2,6,0,3);
w2 = newwin(1,4,5,4);
```

declare two windows named w1 and w2 with the routine `newwin()` according to certain specifications.

New Windows

Following are descriptions of the routines `newwin()` and `subwin()`, which you use to create new windows. For information about creating new pads with `newpad()` and `subpad()`, see `curses`(3V).

`newwin()` — open and return a pointer to new window

```
#include <curses.h>

WINDOW *newwin(nlines, ncols, begin_y, begin_x)
int nlines, ncols, begin_y, begin_x;
```

`newwin()` returns a pointer to a new window with a new data area. The variables `nlines` and `ncols` give the size of the new window. `begin_y` and `begin_x` give the screen coordinates from (0,0) of the upper left corner of the window as it is refreshed to the current screen.

Example:
See the `window` program under `curses` *Program Examples.*

subwin()

```
#include <curses.h>

WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
WINDOW *orig;
int nlines, ncols, begin_y, begin_x;
```

subwin() returns a new window that points to a section of another window, orig. nlines and ncols give the size of the new subwindow. begin_y and begin_x give the screen coordinates of the upper left corner of the window as it is refreshed to the current screen.

Subwindows and original windows can accidentally overwrite one another.

Subwindows of subwindows are not allowed.

Example:

```
#include <curses.h>

main()
{
 WINDOW *sub;

  initscr();
  box(stdscr,'w','w');      /* See the curses(3V) manual page for box() */
  mvwaddstr(stdscr,7,10,"------- this is 10,10");
  mvwaddch(stdscr,8,10,'|');
  mvwaddch(stdscr,9,10,'v');
  sub = subwin(stdscr,10,20,10,10);
  box(sub,'s','s');
  wnoutrefresh(stdscr);
  wrefresh(sub);
  endwin();
}
```

This program prints a border of ws around the stdscr (the sides of your terminal screen) and a border of s characters around the subwindow sub when it is run.

Using Advanced curses Features

Knowing how to use the basic curses routines to get output and input and to work with windows, you can design screen management programs that meet the needs of many users. The curses library, however, has routines that let you do more in a program than handle I/O and multiple windows. The following few pages briefly describe some of these routines and what they can help you do—namely, draw simple graphics, use a terminal's soft labels, and work with more than one terminal in a single curses program.

You should be comfortable using the routines previously discussed in this chapter and the other routines for I/O and window manipulation discussed on the curses(3V) manual page before you try to use the advanced curses features.

Routines for Drawing Lines and Other Graphics

Many terminals have an alternate character set for drawing simple graphics (or glyphs, or graphic symbols). You can use this character set in `curses` programs. `curses` use the same names for glyphs as the VT100 line drawing character set.

To use the alternate character set in a `curses` program, pass a set of variables whose names begin with `ACS_` to the `curses` routine `waddch()` or a related routine. For example, `ACS_ULCORNER` is the variable for the upper left corner glyph. If a terminal has a line drawing character for this glyph, `ACS_ULCORNER`'s value is the terminal's character for that glyph, `OR`ed (`|`) with the bit-mask `A_ALTCHARSET`. If no line-drawing character is available for that glyph, a standard *ASCII* character that approximates the glyph is stored in its place. For example, the default character for `ACS_HLINE`, a horizontal line, is a – (minus sign). When a close approximation is not available, a + (plus sign) is used. All the standard `ACS_` names and their defaults are listed in `curses`(3V).

Part of an example program that uses line drawing characters follows. The example uses the `curses` routine `box()` to draw a box around a menu on a screen. `box()` uses the line drawing characters by default or when | (the pipe) and – are chosen. (See `curses`(3V).) Up and down more indicators are drawn on the box border (using `ACS_UARROW` and `ACS_DARROW`) if the menu contained within the box continues above or below the screen:

```
box(menuwin, ACS_VLINE, ACS_HLINE);

/* output the up/down arrows */
wmove(menuwin, maxy, maxx - 5);

/* output up arrow or horizontal line */
if (moreabove)
   waddch(menuwin, ACS_UARROW);
else
   addch(menuwin, ACS_HLINE);

/*output down arrow or horizontal line */
if (morebelow)
   waddch(menuwin, ACS_DARROW);
else
   waddch(menuwin, ACS_HLINE);
```

Here's another example. Because a default down arrow (like the lowercase letter `v`) isn't very discernible on a screen with many lowercase characters on it, you can change it to an uppercase `V`.

```
if ( ! (ACS_DARROW & A_ALTCHARSET))
   ACS_DARROW = 'V';
```

Routines for Using Soft Labels

Another feature available on most terminals is a set of soft labels across the bottom of their screens. A terminal's soft labels are usually matched with a set of hard function keys on the keyboard. There are usually eight of these labels, each of which is usually eight characters wide and one or two lines high.

The curses library has routines that provide a uniform model of eight soft labels on the screen. If a terminal does not have soft labels, the bottom line of its screen is converted into a soft label area. It is not necessary for the keyboard to have hard function keys to match the soft labels for a curses program to make use of them.

Let's briefly discuss most of the curses routines needed to use soft labels: slk_init(), slk_set(), slk_refresh() and slk_noutrefresh(), slk_clear, and slk_restore.

When you use soft labels in a curses program, you have to call the routine slk_int() before initscr(). This sets an internal flag for initscr() to look at that says to use the soft labels. If initscr() discovers that there are fewer than eight soft labels on the screen, that they are smaller than eight characters in size, or that there is no way to program them, then it will remove a line from the bottom of stdscr to use for the soft labels. The size of stdscr and the LINES variable will be reduced by 1 to reflect this change. A properly written program, one that is written to use the LINES and COLS variables, will continue to run as if the line had never existed on the screen.

slk_init() takes a single argument. It determines how the labels are grouped on the screen should a line get removed from stdscr. The choices are between a 3-2-3 arrangement, and a 4-4 arrangement. The curses routines adjust the width and placement of the labels to maintain the pattern. The widest label generated is eight characters.

The routine slk_set() takes three arguments, the label number (1-8), the string to go on the label (up to eight characters), and the justification within the label (0 = left-justified, 1 = centered, and 2 = right-justified).

The routine slk_noutrefresh() is comparable to wnoutrefresh() in that it copies the label information onto the internal screen image, but it does not cause the screen to be updated. Since a wrefresh() commonly follows, slk_noutrefresh() is the function that is most commonly used to output the labels.

Just as wrefresh() is equivalent to a wnoutrefresh() followed by a doupdate(), so too the function slk_refresh() is equivalent to a slk_noutrefresh() followed by a doupdate().

To prevent the soft labels from getting in the way of a shell escape, slk_clear() may be called before doing the endwin(). This clears the soft labels off the screen and does a doupdate(). The function slk_restore() may be used to restore them to the screen. See the curses(3V) manual page for more information about the routines for using soft labels.

Working with More than One Terminal

A `curses` program can produce output on more than one terminal at the same time. This is useful for single process programs that access a common database, such as multi-player games.

Writing programs that output to multiple terminals is a difficult business, and the `curses` library does not solve all the problems you might encounter. For instance, the programs—not the library routines—must determine the filename and terminal-type of each terminal. The standard method, checking `TERM` in the environment, does not work, because each process can only examine its *own* environment.

Another problem you might face is that of multiple programs reading from one `tty` line. This situation produces a race condition and should be avoided. However, a program trying to take over another terminal cannot just shut off whatever program is currently running on its line. (Usually, security reasons would also make this inappropriate. But, for some applications, such as an inter-terminal communication program, or a program that takes over unused terminal lines, it would be appropriate.) A typical solution to this problem requires each user logged in on a line to run a program that notifies a master program that the user is interested in joining the master program and tells it the notification program's process ID, the name of the tty line, and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program and all programs exit.

A `curses` program handles multiple terminals by always having a current terminal. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary `curses` routines.

References to terminals in a `curses` program have the type `SCREEN*`. A new terminal is initialized by calling `newterm(`*type, outfd, infd*`)`. `newterm ()` returns a screen reference to the terminal being set up. *type* is a character string, naming the kind of terminal being used. *outfd* is a `stdio`(3V) file pointer (`FILE*`) used for output to the terminal and *infd* a file pointer for input from the terminal. This call replaces the normal call to `initscr()`, which calls `newterm(getenv(''TERM''), stdout, stdin)`.

To change the current terminal, call `set_term(`*sp*`)` where *sp* is the screen reference to be made current. `set_term()` returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with `newterm()`. Options such as `cbreak()` and `noecho()` must be set separately for each terminal. The functions `endwin()` and `refresh()` must be called separately for each terminal. The figure below shows a typical scenario to output a message to several terminals.

Figure 13-6 *Sending a Message to Several Terminals*

```
for (i=0; i<nterm; i++)
{
    set_term(terms[i]);
    mvaddstr(0, 0, "Important message");
    refresh();
}
```

See the two program under curses *Program Examples* for a more complete example.

13.3. Working with terminfo Routines

Some programs need to use lower-level routines than those offered by the curses routines. For such programs, the terminfo routines are offered. They do not manage your terminal screen, but rather, give you access to strings and capabilities which you can use yourself to manipulate the terminal.

terminfo routines should not be used directly, except in the circumstances noted at right; the equivalent curses routines protect your program from the idiosyncracies of physical terminals. When you use the terminfo routines, you must deal with them yourself. Also, these low-level routines may change, rendering programs that rely on them obsolete.

There are three circumstances when it is proper to use terminfo routines directly. The first is when you need only some screen management capabilities, for example, making text standout on a screen. The second is when writing a filter. A typical filter does one transformation on an input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of the terminfo routines is worthwhile. The third is when you are writing a special-purpose tool that sends a special string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line.

Otherwise, you are discouraged from using these routines: the higher level curses routines make your program more portable to other SunOS systems, and to a wider class of terminals.

What Every terminfo Program Needs

A terminfo program typically includes the header files and routines shown below:

Figure 13-7 *Typical Framework of a* terminfo Program

```
#include <curses.h>
#include <term.h>
...
    setupterm( (char*)0, 1, (int*)0 );
    ...
    putp(clear_screen);
    ...
    reset_shell_mode();
    exit(0);
```

The header files <curses.h> and <term.h> are required because they contain the definitions of the strings, numbers, and flags used by the terminfo routines. setupterm() takes care of initialization. Passing this routine the values (char*)0, 1, and (int*)0 invokes reasonable defaults. If setupterm() can't figure out what kind of terminal you are on, it prints an error message and exits. reset_shell_mode() performs functions similar to endwin() and should be called before a terminfo program exits.

A global variable like clear_screen is defined by the call to setupterm(). It can be output using the terminfo routines putp() or tputs(), which gives a user more control. This string should not be directly output to the terminal using the C library routine printf(3V), because it contains padding information. A program that directly outputs strings will fail on terminals that require padding or that use the xon/xoff flow control protocol.

At the terminfo level, the higher level routines like addch() and getch() are not available. It is up to you to output whatever is needed. For a list of capabilities and a description of what they do, see terminfo(5V); see curses(3V) for a list of all the terminfo routines.

Compiling and Running a terminfo Program

The general command line for compiling, and the guidelines for running a program with terminfo routines are the same as those for compiling any other curses program.

An Example terminfo Program

The example program, termhl, shows a simple use of terminfo routines. It is a version of the highlight program (see curses *Program Examples*) that does not use the higher level curses routines. termhl can be used as a filter. It includes the strings to enter bold and underline mode and to turn off all attributes.

```
/*
 * A terminfo level version of the highlight program.
 */

#include <curses.h>
#include <term.h>

int ulmode = 0;      /* Currently underlining */

main(argc, argv)
  int argc;
  char **argv;
{
   FILE *fd;
   int c, c2;
   int outch();

   if (argc > 2)
   {
      fprintf(stderr, "Usage: termhl [file]\n");
      exit(1);
   }
   if (argc == 2)
   {
      fd = fopen(argv[1], "r");
      if (fd == NULL)
      {
         perror(argv[1]);
         exit(2);
      }
   }
   else
   {
      fd = stdin;
   }
   setupterm((char*)0, 1, (int*)0);

   for (;;)
   {
      c = getc(fd);
      if (c == EOF)
      break;
      if (c == '\')
      {
         c2 = getc(fd);
         switch (c2)
         {
            case 'B':
            tputs(enter_bold_mode, 1, outch);
            continue;
            case 'U':
            tputs(enter_underline_mode, 1, outch);
            ulmode = 1;
            continue;
            case 'N':
            tputs(exit_attribute_mode, 1, outch);
            ulmode = 0;
            continue;
         }
         putch(c);
         putch(c2);
      }
      else
         putch(c);

   }
   fclose(fd);
   fflush(stdout);
   resetterm();
   exit(0);
}

/*
 * This function is like putchar, but it checks for underlining.
 */
putch(c)
   int c;
{
   outch(c);
   if (ulmode && underline_char)
   {
      outch('\b');
      tputs(underline_char, 1, outch);
   }
}

/*
 * Outchar is a function version of putchar that can be passed to
 * tputs as a routine to call.
 */
outch(c)
   int c;
{
   putchar(c);
}
```

Let's discuss the use of the function `tputs(`*cap, affcnt, outc*`)` in this program to gain some insight into the `terminfo` routines. `tputs()` applies padding information. Some terminals have the capability to delay output. Their terminal descriptions in the `terminfo` database probably contain strings like `$<20>`, which means to pad for 20 milliseconds (see the following section *Specifying Capabilities*). `tputs` generates enough pad characters to delay for the appropriate time.

`tput()` has three parameters. The first parameter is the string capability to be output.

The second is the number of lines affected by the capability. Some capabilities may require padding that depends on the number of lines affected. For example, `insert_line` may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention *affcnt* is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since *affcnt* is multiplied by the amount of time per item, and anything multiplied by 0 is 0.

The third parameter is a routine to be called with each character.

For many simple programs, *affcnt* is always 1 and *outc* always calls `putchar`. For these programs, the routine `putp(`*cap*`)` is a convenient abbreviation. `termhl` could be simplified by using `putp()`.

Now to understand why you should use the `curses` level routines instead of `terminfo` level routines whenever possible, note the special check for the `underline_char` capability in this sample program. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. `termhl` keeps track of the current mode, and if the current character is supposed to be underlined, outputs `underline_char`, if necessary. Low level details such as this are precisely why the `curses` level is recommended over the terminfo level. `curses` takes care of terminals with different methods of underlining and other terminal functions. Programs at the `terminfo` level must handle such details themselves.

`termhl` was written to illustrate a typical use of the `terminfo` routines. It is more complex than it need be in order to illustrate some properties of `terminfo` programs. The routine `vidattr` (see `curses`(3V)) could have been used instead of directly outputting enter_bold_mode, enter_underline_mode, and exit_attribute_mode. In fact, the program would be more robust if it did, since there are several ways to change video attribute modes.

13.4. Working with the terminfo Database

The terminfo database describes the many terminals with which curses programs, as well as some SunOS system tools, like vi(1), can be used. Each terminal description is a compiled file containing the names that the terminal is known by and a group of comma-separated fields describing the actions and capabilities of the terminal. This section describes the terminfo database, related support tools, and their relationship to the curses library.

Writing Terminal Descriptions

Descriptions of many popular terminals are already provided in the terminfo database. However, it is possible that you'll want to run a curses program on a terminal for which there is no existing description. In this case, you'll have to build the description.

The general procedure for building a terminal description is as follows:

1. Give the known names of the terminal.
2. Learn about, list, and define the known capabilities.
3. Compile the newly-created description entry.
4. Test the entry for correct operation.
5. Go back to step 2, add more capabilities, and repeat, as necessary.

Building a terminal description is sometimes easier when you build small parts of the description and test them as you go along. These tests can expose deficiencies in the ability to describe the terminal. Also, modifying an existing description of a similar terminal can make the building task easier.

Naming the Terminal

The name of a terminal is the first information given in a terminfo terminal description. This string of names, assuming there is more than one name, is separated by vertical bars (|). The first name given should be the most common abbreviation for the terminal. The last name given is typically a verbose entry that fully identifies the terminal by make and model. The long name or ''verbose'' is typically the manufacturer's formal name for the terminal. Names between the first and last entries are known synonyms for the terminal name. All but the verbose name should be typed in lowercase letters and contain no blanks. Naturally, the formal name is entered as closely as possible to the manufacturer's name.

Here is the name string from the description for a virtual terminal.

```
    virtual|VIRTUAL|cbunix|cb-unix|cb-unix virtual terminal,
```

Notice that the first name is the most commonly used abbreviation and the last is the long name. Also notice the comma at the end of the name string.

Here's the name string for a fictitious terminal, myterm:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
```

Terminal names should follow common naming conventions. These conventions start with a root name, like `virtual` or `myterm`, for example. Possible hardware modes or user preferences should be shown by adding a hyphen and a 'mode indicator' at the end of the name. For example, the 'wide mode' (which is shown by a –w) version of our fictitious terminal would be described as `myterm–w`. `terminfo`(5V) describes mode indicators in greater detail.

Learning About the Capabilities

After you complete the string of terminal names for your description, you have to learn about the terminal's capabilities so that you can properly describe them. To learn about the capabilities your terminal has, you should do the following:

See the owner's manual for your terminal. It should have information about the capabilities available and the character strings that make up the sequence transmitted from the keyboard for each capability.

Test the keys on your terminal to see what they transmit, if this information is not available in the manual. You can test the keys in one of the following wayss, type:

```
stty –echo; cat –vu
```

followed by the keys you want to test. To return to the shell and restore echo, type:

```
^D
stty echo
```

Note that `stty echo` is not displayed on the terminal screen.

Specifying Capabilities

Once you know the capabilities of your terminal, you have to provide them in your terminal description. Capability entries consist of a list of comma-separated fields containing the abbreviated `terminfo` name and, in some cases, the terminal's value for each capability. For example, `bel` is the abbreviated name for the beeping or ringing capability. On most terminals, a [CTRL-G] is the instruction that produces a beeping sound. Therefore, the beeping capability would be shown in the terminal description as `bel=^G,`.

The list of capabilities may continue across input lines as long as the continuation lines start with a white-space character, or consist of a comment. Comments can be included within the description by putting a # at the beginning of the line.

For a `curses` program to run on any given terminal, its description in the `terminfo` database must include, at least, the capabilities to move a cursor in all four directions and to clear the screen.

The `terminfo`(5V) manual page has a complete list of the capabilities you can use in a terminal description.

A terminal's character sequence (value) for a capability can be a keyed operation (like [CTRL-G]), a numeric value, or a parameter string containing the sequence of operations required to achieve the particular capability. In a terminal description, certain characters are used after the capability name to show what type of character sequence is required. Explanations of these characters are given below.

# This shows that a numeric value is to follow. This character follows a capability that needs a number as a value. For example, the number of columns is defined as `cols#80,`.

= This shows that the capability value is the character string that follows. This string instructs the terminal how to act and may actually be a sequence of commands. There are certain characters used in the instruction strings that have special meanings. These special characters follow:

^ This shows a control character is to be used. For example, the beeping sound is produced by a CTRL-G. This would be shown as `^G`.

`\E \e` These characters followed by another character show an escape instruction. An entry of `\EC` would transmit to the terminal as ESC-C.

`\n` These characters provide a NEWLINE character sequence.

`\l` These characters provide a LINEFEED character sequence.

`\r` These characters provide a RETURN character sequence.

`\t` These characters provide a TAB character sequence.

`\b` These characters provide a BACKSPACE character sequence.

`\f` These characters provide a FORMFEED character sequence.

`\s` These characters provide a SPACE character sequence.

`\`*nnn* This is a character whose three-digit octal is *nnn* (*nnn* can be from one to three digits).

`$<`*n*`>` These symbols are used to show a delay in milliseconds. The desired length of delay is enclosed inside the brackets. The amount of delay may be a whole number, a numeric value to one decimal place (tenths), or either form followed by an asterisk (*). The * shows that the delay is to be proportional to the number of lines affected by the operation. For example, a 20-millisecond delay per line would appear as `$<20*>`. See the `terminfo`(5V) manual page for more information about delays and padding.

Sometimes, it may be necessary to comment out a capability so that the terminal ignores this particular field. This is done by placing a period (.) in front of the abbreviated name for the capability. For example, if you would like to comment out the beeping capability, the description entry would appear as

```
.bel=^G,
```

With this background information about specifying capabilities, let's add the capability string to our description of `myterm`. We'll consider basic capabilities, screen-oriented capabilities, keyboard-entered capabilities, and parameter string capabilities.

Basic Capabilities

Some capabilities common to most terminals are bells, columns, lines on the screen, and overstriking of characters, if necessary. Suppose our fictitious terminal has these and a few other capabilities, as listed below. Note that the list gives the abbreviated `terminfo` name for each capability in the parentheses following the capability description:

- An automatic wrap around to the beginning of the next line whenever the cursor reaches the right-hand margin (`am`).
- The ability to produce a beeping sound. The instruction required to produce the beeping sound is `^G` (`bel`).
- An 80-column wide screen (`cols`).
- A 30-line long screen (`lines`).
- Use of xon/xoff protocol (`xon`).

By combining the name string with the capability descriptions that we now have, we get the following general `terminfo` database entry:

```
myterm|mytm|mine|fancy|terminal|My FANCY terminal,
        am, bel=^G, cols#80, lines#30, xon,
```

Screen-Oriented Capabilities

Screen-oriented capabilities manipulate the contents of a screen. Our example terminal `myterm` has the following screen-oriented capabilities. Again, the abbreviated command associated with the given capability is shown in parentheses.

- A RETURN is a CTRL-M (`cr`).
- A cursor up one line motion is a CTRL-K (`cuu1`).
- A cursor down one line motion is a CTRL-J (`cud1`).
- Moving the cursor to the left one space is a CTRL-H (`cub1`).
- Moving the cursor to the right one space is a CTRL-L (`cuf1`).
- Entering reverse video mode is an ESCAPE-D (`smso`).
- Exiting reverse video mode is an ESCAPE-Z (`rmso`).
- A clear to the end of a line sequence is an ESCAPE-K and should have a 3-millisecond delay (`el`).

A terminal scrolls when receiving a NEWLINE at the bottom of a page (`ind`).

The revised terminal description for `myterm` including these screen-oriented capabilities follows:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
        am, bel=^G, cols#80, lines#30, xon,
        cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
        smso=\ED, rmso=\EZ, el=\EK$<3>, ind=\n,
```

Keyboard-Entered Capabilities

Keyboard-entered capabilities are sequences generated when a key is typed on a terminal keyboard. Most terminals have, at least, a few special keys on their keyboard, such as arrow keys and the backspace key. Our example terminal has several of these keys whose sequences are, as follows:

- The backspace key generates a CTRL-H (kbs).
- The up arrow key generates an ESCAPE-[A (kcuu1).
- The down arrow key generates an ESCAPE-[B (kcud1).
- The right arrow key generates an ESCAPE-[C (kcuf1).
- The left arrow key generates an ESCAPE-[D (kcub1).

The home key generates an ESCAPE-[H (khome).

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
        am, bel=^G, cols#80, lines#30, xon,
        cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
        smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
        kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
        kcub1=\E[D, khome=\E[H,
```

Parameter String Capabilities

Parameter string capabilities are capabilities that can take parameters, such as those used to position a cursor on a screen, or to turn on a combination of video modes. To address a cursor, the cup capability is used and is passed two parameters: the row and column to address. String capabilities, such as cup and set attributes (sgr) capabilities, are passed arguments in a terminfo program by the tparm() routine.

The arguments to string capabilities are manipulated with special % sequences similar to those found in a call to printf(3V). In addition, many of the features found on a simple stack-based RPN calculator are available. cup, as noted above, takes two arguments: the row and column. sgr, takes nine arguments, one for each of the nine video attributes. See terminfo(5V) for the list and order of the attributes and further examples of sgr.

Our fancy terminal's cursor position sequence requires a row and column to be output as numbers separated by a semicolon, preceded by ESCAPE-[and followed with H. The coordinate numbers are 1-based rather than 0-based. Thus, to move to row 5, column 18, from (0,0), the sequence ;r"ESCAPE- [6 would be output.

Integer arguments are pushed onto the stack with a %p sequence followed by the argument number, such as %p2 to push the second argument. A shorthand sequence to increment the first two arguments is '%i'. To output the top number on the stack as a decimal, a %d sequence is used, exactly as in printf.

Our terminal's cup sequence is built up as follows:

cup=	Meaning
\E[	output ESCAPE-[
%i	increment the two arguments
%p1	push the 1st argument (the row) onto the stack
%d	output the row as a decimal
;	output a semi-colon
%p2	push the 2nd argument (the column) onto the stack
%d	output the column as a decimal
H	output the trailing letter

or

```
cup=\E[%i%p1%d;%p2%dH,
```

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
        am, bel=^G, cols#80, lines#30, xon,
        cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
        smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
        kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
        kcub1=\E[D, khome=\E[H,
        cup=\E[%i%p1%d;%p2%dH,
```

See terminfo(5V) for more information about parameter string capabilities.

Compiling the Description

The terminfo database entries are compiled using tic, the terminfo compiler command. This compiler translates terminfo source entries into the compiled format used by the terminfo and curses routines.

The source file for the source file is usually suffixed with .ti. For example, the description of myterm would be in a source file named myterm.ti. The compiled description of myterm would usually be placed in /usr/share/lib/terminfo/m/myterm, since the first letter in the description entry is m. Links would also be made to synonyms of myterm, for example, to /f/fancy. If the environment variable TERMINFO were set to a directory and exported before the entry was compiled, the compiled entry would be placed in the TERMINFO directory. All programs using the entry would then look in the new directory for the description file if TERMINFO were set, before looking in the default /usr/share/lib/terminfo. The general format for the tic command is:

```
tic [-v] [-c] sourcefile
```

With the -v, verbose option, the compiler traces its actions and prints messages regarding its progress. The -c option checks for errors. tic(8V) compiles only one file at a time. The following command line shows how to compile the terminfo source file for myterm.

```
tic -v myterm.ti
```

Refer to tic(8V) for more information.

Testing the Description

Let's consider ways to test a terminal description. First, you can test it by setting the environment variable TERMINFO to the path name of the directory containing the description. If programs run the same on the new terminal as they did on the older known terminals, then the new description is functional.

Or, you can use the tput(1V) command. This command outputs a string or an integer according to the type of capability being described. If the capability is a Boolean expression, then tput sets the exit code (0 for TRUE, 1 for FALSE) and produces no output. The general format for the tput command is as follows:

tput [-T*type*] *capname*

The type of terminal you are requesting information about is identified with the -T*type* option. Usually, this option is not necessary because the default terminal name is taken from the environment variable TERM. The *capname* field is used to show what capability to output from the terminfo database.

The following command line shows how to output the "clear screen" character sequence for the terminal being used:

```
tput clear
```

The following command line shows how to output the number of columns for the terminal being used:

```
tput cols
```

tput(8V) contains more information on the usage and possible messages associated with this command.

Comparing or Printing terminfo Descriptions

Sometime you may want to compare two terminal descriptions or quickly look at a description without going to the terminfo source directory. The infocmp(8V) command was designed to help you with both of these tasks. Compare two descriptions of the same terminal; for example,

```
mkdir /tmp/old /tmp/new
TERMINFO=/tmp/old tic oldvirtual.ti
TERMINFO=/tmp/new tic newvirtual.ti
infocmp -A /tmp/old -B /tmp/new -d virtual virtual
```

compares the old and new virtual entries.

To print out the terminfo source for the virtual, type:

```
infocmp -I virtual
```

Converting a termcap Description to a terminfo Description

The terminfo database is an alternative to the termcap database. Because of the many programs and processes that have been written with and for the termcap database, it is not feasible to do a complete conversion from termcap to terminfo. Since converting between the two requires experience with both, all entries into the databases should be handled with extreme caution. These files are important to the operation of your terminal.

The captoinfo(8V) command converts termcap(5) descriptions to terminfo(5V) descriptions. When a file is passed to captoinfo, it looks for termcap descriptions and writes the equivalent terminfo descriptions on the standard output. For example,

```
captoinfo /etc/termcap
```

converts the file /etc/termcap to terminfo source, preserving comments and other extraneous information within the file. The command line

```
captoinfo
```

looks up the current terminal in the termcap database, as specified by the TERM and TERMCAP environment variables and converts it to terminfo.

To convert a terminfo description into a termcap entry, use **infocmp -C**.

If you have been using cursor optimization programs with the -ltermcap or -ltermlib option in the /usr/5bin/cc command line, those programs should still be functional.

13.5. curses Program Examples

The following examples demonstrate uses of curses routines.

The editor Program

This program illustrates how to use curses routines to write a screen editor. For simplicity, editor keeps the buffer in stdscr; obviously, a real screen editor would have a separate data structure for the buffer. This program has many other simplifications: no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. First, it uses the move(), mvaddstr(), flash(), wnoutrefresh() and clrtoeol() routines. These routines are all discussed in this chapter under *Working with curses* Routines.

Second, it also uses some curses routines that we have not discussed. For example, the function to write out a file uses the mvinch() routine, which returns a character in a window at a given position. The data structure used to write out a file does not keep track of the number of characters in a line or the number of lines in the file, so trailing blanks are eliminated when the file is written. The program also uses the insch(), delch(), insertln(), and deleteln() routines. These functions insert and delete a character or line. See curses(3V) for more information about these routines.

Third, the editor command interpreter accepts special keys, as well as ASCII characters. On one hand, new users find an editor that handles special keys easier to learn about. For example, it's easier for new users to use the arrow keys to move a cursor than it is to memorize that the letter h means left, j means down, k means up, and l means right. On the other hand, experienced users usually like having the ASCII characters to avoid moving their hands from the home row position to use special keys.

Since not all terminals have arrow keys, your curses programs will work with more terminals if there is an ASCII character associated with each special key.

Fourth, the CTRL-L command illustrates a feature most programs using curses routines should have. Often some program beyond the control of the routines writes something to the screen (for instance, a broadcast message) or some line noise affects the screen so much that the routines cannot keep track of it. A user invoking editor can type CTRL-L, causing the screen to be cleared and redrawn with a call to wrefresh(curscr).

Finally, another important point is that the input command is terminated by CTRL-D, not the ESCAPE key. It is very tempting to use ESCAPE as a command, since it is one of the few special keys available on all keyboards. (RETURN and BREAK are the only others.) However, using escape as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with escape (i.e., escape sequences) to control the terminal, and have special keys that send escape sequences to the computer. If a computer receives an escape from a terminal, it cannot tell whether the user depressed the ESCAPE key or whether a special key was pressed.

editor and other curses programs handle the ambiguity by setting a timer. If another character is received during this time, and if that character might be the beginning of a special key, the program reads more input until either a full special key is read, the time out is reached, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press ESCAPE, then to type another key quickly, which causes the curses program to think a special key has been pressed. Also, a pause occurs until the escape can be passed to the user program, resulting in a slower response to the ESCAPE key.

Many existing programs use ESCAPE as a fundamental command, which cannot be changed without infuriating a large class of users. These programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a time-out solution. The moral is clear: when designing your curses programs, avoid the ESCAPE key.

editor — a Sample Program Listing

```
 /* editor: A screen-oriented editor.  The user
  * interface is similar to a subset of vi.
  * The buffer is kept in stdscr to simplify
  * the program.
  */

#include <stdio.h>
#include <curses.h>

#define CTRL(c) ((c) & 037)

main(argc, argv)
int argc;
char **argv;
{
        extern void perror(), exit();
        int i, n, l;
        int c;
        int line = 0;
        FILE *fd;

        if (argc != 2)
        {
                fprintf(stderr, "Usage: %s file\n", argv[0]);
                exit(1);
        }

        fd = fopen(argv[1], "r");
        if (fd == NULL)
        {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        cbreak();
        nonl();
        noecho();
        idlok(stdscr, TRUE);
        keypad(stdscr, TRUE);

        /* Read in the file */
        while ((c = getc(fd)) != EOF)
        {
                if (c == '\n')
                        line++;
                if (line > LINES - 2)
                        break;
                addch(c);
        }

        fclose(fd);

        move(0,0);
        refresh();
        edit();
        /* Write out the file */
        fd = fopen(argv[1], "w");
        for (l = 0; l < LINES - 1; l++)
        {
                n = len(l);
                for (i = 0; i < n; i++)
                        putc(mvinch(l, i) & A_CHARTEXT, fd);
                putc('\n', fd);
        }
        fclose(fd);

        endwin();
        exit(0);
}

len(lineno)
int lineno;
{
        int linelen = COLS - 1;

        while (linelen >= 0 && mvinch(lineno, linelen) == ' ')
                linelen--;
        return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
        int c;

        for (;;)
        {
                move(row, col);
                refresh();
                c = getch();

                /* Editor commands */
                switch (c)
                {

                /* hjkl and arrow keys: move cursor
                 * in direction indicated */
                case 'h':
                case KEY_LEFT:
                        if (col > 0)
                                col--;
                        else
                                flash();
                        break;

                case 'j':
                case KEY_DOWN:
                        if (row < LINES - 1)
                                row++;
                        else
                                flash();
                        break;

                case 'k':
                case KEY_UP:
                        if (row > 0)
                                row--;
                        else
                                flash();
                        break;

                case 'l':
                case KEY_RIGHT:
                        if (col < COLS - 1)
                                col++;
                        else
                                flash();
                        break;

                /* i: enter input mode */
                case KEY_IC:
                case 'i':
                        input();
                        break;

                /* x: delete current character */
                case KEY_DC:
                case 'x':
                        delch();
                        break;

                /* o: open up a new line and enter input mode */
                case KEY_IL:
                case 'o':
                        move(++row, col = 0);
                        insertln();
                        input();
                        break;

                /* d: delete current line */
                case KEY_DL:
                case 'd':
                        deleteln();
                        break;

                /* ^L: redraw screen */
                case KEY_CLEAR:
                case CTRL('L'):
                        wrefresh(curscr);
                        break;

                /* w: write and quit */
                case 'w':
                        return;

                /* q: quit without writing */
                case 'q':
                        endwin();
                        exit(2);
                default:
                        flash();
                        break;
                }
        }
}
/*
```

```
 * Insert mode: accept characters and insert them.
 *  End with ^D or EIC
 */
input()
{
        int c;

        standout();
        mvaddstr(LINES - 1, COLS - 20, "INPUT MODE");
        standend();
        move(row, col);
        refresh();
        for (;;)
        {
                c = getch();
                if (c == CTRL('D') || c == KEY_EIC)
                        break;
                insch(c);
                move(row, ++col);
                refresh();
        }
        move(LINES - 1, COLS - 20);
        clrtoeol();
        move(row, col);
        refresh();
}
```

The highlight Program

This program illustrates a use of the routine attrset(). highlight reads a text file and uses embedded escape sequences to control attributes. \U turns on underlining, \B turns on bold, and \N restores the default output attributes.

Note the first call to scrollok(), a routine that we have not previously discussed (see curses(3V)). This routine allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, scrollok() automatically scrolls the terminal up a line and calls refresh().

```
/*
 * highlight: a program to turn \U, \B, and
 * \N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */

#include <stdio.h>
#include <curses.h>

main(argc, argv)
int argc;
char **argv;
{
        FILE *fd;
        int c, c2;
        void exit(), perror();

        if (argc != 2)
        {
                fprintf(stderr, "Usage: highlight file\n");
                exit(1);
        }

        fd = fopen(argv[1], "r");

        if (fd == NULL)

        {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        scrollok(stdscr, TRUE);
        nonl();
        while ((c = getc(fd)) != EOF)
        {
                if (c == '\\')
                {
                        c2 = getc(fd);
                        switch (c2)
                        {
                        case 'B':
                                attrset(A_BOLD);
                                continue;
                        case 'U':
                                attrset(A_UNDERLINE);
                                continue;
                        case 'N':
                                attrset(0);
                                continue;
                        }
                        addch(c);
                        addch(c2);
                }
                else
                        addch(c);
        }
        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

The scatter Program

This program takes the first LINES – 1 lines of characters from the standard input and displays the characters on a terminal screen in a random order. For this program to work properly, the input file should not contain tabs or non-printing characters.

```
/*
 *     The scatter program.
 */

#include     <curses.h>
#include     <sys/types.h>

extern time_t time();

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS]; /* Screen Array */
int  T[MAXLINES][MAXCOLS]; /* Tag Array - Keeps track of   *
                            * the number of characters     *
                            * printed and their positions. */

main()
{
       register int row = 0,col = 0;
       register int c;
       int char_count = 0;
       time_t t;
       void exit(), srand();

       initscr();
       for(row = 0;row < MAXLINES;row++)
              for(col = 0;col < MAXCOLS;col++)
                     s[row][col]=' ';

       col = row = 0;
       /* Read screen in */
       while ((c=getchar()) != EOF && row < LINES ) {

              if(c != '\n')

              {
                     /* Place char in screen array */
                     s[row][col++] = c;
                     if(c != ' ')
                            char_count++;
              }
              else
              {
                     col = 0;
                     row++;
              }
       }

       time(&t);     /* Seed the random number generator */
       srand((unsigned)t);

       while (char_count)
       {
              row = rand() % LINES;
              col = (rand() >> 2) % COLS;
              if (T[row][col] != 1 && s[row][col] != ' ')
```

```
                {
                        move(row, col);
                        addch(s[row][col]);
                        T[row][col] = 1;
                        char_count--;
                        refresh();
                }
        }
        endwin();
        exit(0);
}
```

The show Program

show pages through a file, showing one screen of its contents each time you depress the space bar. The program calls cbreak() so that you can depress the space bar without having to hit return; it calls noecho() to prevent the space from echoing on the screen. The nonl() routine, which we have not previously discussed, is called to enable more cursor optimization. The idlok() routine, which we also have not discussed, is called to allow insert and delete line. (See curses(3V) for more information about these routines). Also notice that clrtoeol() and clrtobot() are called.

By creating an input file for show made up of screen-sized (about 24 lines) pages, each varying slightly from the previous page, nearly any exercise for a curses() program can be created. This type of input file is called a show script.

```
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
char *argv[];
{
        FILE *fd;
        char linebuf[BUFSIZ];
        int line;
        void done(), perror(), exit();

        if (argc != 2)
        {
                fprintf(stderr, "usage: %s file\n", argv[0]);
                exit(1);
        }

        if ((fd=fopen(argv[1], "r")) == NULL)
        {

                perror(argv[1]);
                exit(2);
        }

        signal(SIGINT, done);

        initscr();
        noecho();
        cbreak();
        nonl();
        idlok(stdscr, TRUE);

        while(1)
        {
                move(0,0);
                for (line = 0; line < LINES; line++)
                {
                        if (!fgets(linebuf, sizeof linebuf, fd))
                        {
                                clrtobot();
                                done();
                        }
                        move(line, 0);
                        printw("%s", linebuf);
                }
                refresh();
                if (getch() == 'q')
                        done();
        }
}

void done()
{
        move(LINES - 1, 0);
        clrtoeol();
        refresh();
        endwin();
        exit(0);
}
```

The two Program

This program pages through a file, writing one page to the terminal from which the program is invoked and the next page to the terminal named on the command line. It then waits for a space to be typed on either terminal and writes the next page to the terminal at which the space is typed.

two is just a simple example of a two-terminal curses program. It does not handle notification; instead, it requires the name and type of the second terminal on the command line. As written, the command "sleep 100000" must be typed at the second terminal to put it to sleep while the program runs, and the user of the first terminal must have both read and write permission on the second terminal.

```
#include <curses.h>
#include <signal.h>

SCREEN *me, *you;
SCREEN *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
int argc;
char **argv;
{
        void done(), exit();
        unsigned sleep();
        char *getenv();
        int c;

        if (argc != 4)
        {
             fprintf(stderr, "Usage: two othertty otherttytype inputfile\n");
             exit(1);
        fd = fopen(argv[3], "r");
        fdyou = fopen(argv[1], "w+");
        signal(SIGINT, done);        /* die gracefully */

        me = newterm(getenv("TERM"), stdout, stdin);  /* initialize my tty */
        you = newterm(argv[2], fdyou, fdyou);       /* Initialize the other terminal */

        set_term(me);     /* Set modes for my terminal */
        noecho();         /* turn off tty echo */
        cbreak();         /* enter cbreak mode */
        nonl();           /* Allow linefeed */
        nodelay(stdscr, TRUE);     /* No hang on input */

        set_term(you);    /* Set modes for other terminal */
        noecho();
        cbreak();
        nonl();
        nodelay(stdscr,TRUE);

        /* Dump first screen full on my terminal */
        dump_page(me);

        /* Dump second screen full on the other terminal */
        dump_page(you);

        for (;;) /* for each screen full */
        {
             set_term(me);
             c = getch();
             if (c == 'q')         /* wait for user to read it */
             done();
             if (c == ' ')
             dump_page(me);

             set_term(you);
             c = getch();
             if (c == 'q')         /* wait for user to read it */
             done();
             if (c == ' ')
             dump_page(you);
             sleep(1);
        }
}

dump_page(term)
  SCREEN *term;
{
        int line;

        set_term(term);
        move(0, 0);
        for (line = 0; line < LINES - 1; line++) {
             if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
             clrtobot();
             done();
             }
```

```
                mvaddstr(line, 0, linebuf);
        }
        standout();
        mvprintw(LINES - 1, 0, "--More--");
        standend();
        refresh();          /* sync screen */
}
/*
 * Clean up and exit.
 */
void done()
{
        /* Clean up first terminal */
        set_term(you);
        move(LINES - 1,0);           /* to lower left corner */

        clrtoeol();         /* clear bottom line */
        refresh();          /* flush out everything */
        endwin();           /* curses cleanup */

        /* Clean up second terminal */
        set_term(me);
        move(LINES - 1,0);           /* to lower left corner */
        clrtoeol();         /* clear bottom line */
        refresh();          /* flush out everything */
        endwin();           /* curses cleanup */
        exit(0);
}
```

The window Program

This example program demonstrates the use of multiple windows. The main display is kept in stdscr. When you want to put something other than what is in stdscr on the physical terminal screen temporarily, a new window is created covering part of the screen. A call to wrefresh() for that window causes it to be written over the stdscr image on the terminal screen. Calling refresh() on stdscr results in the original window being redrawn on the screen. Note the calls to the touchwin() routine (which we have not discussed — see curses(3V)) that occur before writing out a window over an existing window on the terminal screen. This routine prevents screen optimization in a curses program. If you have trouble refreshing a new window that overlaps an old window, it may be necessary to call touchwin() for the new window to get it completely written out.

```
#include <curses.h>

WINDOW *cmdwin;

main()

{

        int i, c;
        char buf[120];
        void exit();

        initscr();
        nonl();
        noecho();
        cbreak();

        cmdwin = newwin(3, COLS, 0, 0);/* top 3 lines */
        for (i = 0; i < LINES; i++)
                mvprintw(i, 0, "This is line %d of stdscr", i);

        for (;;)

        {
                refresh();
                c = getch();
                switch (c)

                {
                case 'c':       /* Enter command from keyboard */
                        werase(cmdwin);
                        wprintw(cmdwin, "Enter command:");
                        wmove(cmdwin, 2, 0);
                        for (i = 0; i < COLS; i++)
                                waddch(cmdwin, '-');
                        wmove(cmdwin, 1, 0);
                        touchwin(cmdwin);
                        wrefresh(cmdwin);
                        wgetstr(cmdwin, buf);
                        touchwin(stdscr);

                        /*
                         * The command is now in buf.
                         * It should be processed here.
                         */

                case 'q':
                        endwin();
                        exit(0);
                }

        }
}
```

A

SCCS Low-Level Commands

A

SCCS Low-Level Commands

This appendix contains a summary of the individual SCCS commands. The user-level interface to SCCS is described in chapter 8 of this manual. In the unlikely event that you need to use the 'raw' commands of SCCS, here they are. Be aware that the commands described here do not make any assumptions about where the SCCS-files are — you must spell them out in excruciating detail. The individual SCCS tools are not easy to use, but they do provide extremely close control over the SCCS database files. Of particular interest are the numbering of versions and branch versions, the `l.` file, which gives a description of what deltas were used on a `get`, and certain other SCCS commands.

The following topics are covered here:

- □ The scheme used to identify versions of text kept in an SCCS file.
- □ Basic information needed for day-to-day use of SCCS commands, including a discussion of the more useful arguments.
- □ Protection and auditing of SCCS files, including the differences between the use of SCCS by individual users on one hand, and groups of users on the other.

A.1. Low Level SCCS For Beginners

In this section, we present some basic concepts of SCCS. Examples are fragments of terminal sessions, with what you type shown in **`bold typewriter font`**, and what the terminal displays shown in `typewriter font`.

Note that all the SCCS commands described here live in the `/usr/sccs` directory, so you must either include the directory pathname explicitly when using SCCS commands, or include it in your shell's search path. This chapter assumes that you included have `/usr/sccs` in your path.

Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file; each set of changes usually depends on all previous sets. Each set of changes is called a *delta* and is assigned a name called the SCCS Identification string (SID).

The SID is composed of at most four components; for now let's focus on only the first two: the ''release'' and ''level'' numbers. Each set of changes to a file is named '*release . level*'; hence, the first delta is called '1.1', the second '1.2', the third '1.3', and so on. The release number can also be changed, allowing, for example, deltas '2.1', '3.19', etc. A change in the release number can be used, perhaps, to indicate a major update to the file, or to signal the start of a new round of related updates.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines the version of the SCCS file obtained by applying the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order, to the null (empty) version of the file. A.16.2.

A.2. SCCS File Numbering Conventions

You can think of the deltas applied to an SCCS file as the nodes of a tree; the root is the initial version of the file. The root delta (node) is normally named '1.1' and successor deltas (nodes) are named '1.2', '1.3', etc. We have already discussed these two components of the names of the deltas, the 'release' and 'level' numbers; and you have seen that normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, you have seen how to change the release number when making a delta, to indicate that a major change to the file is being made. The new release number applies to all successor deltas, unless it is specifically changed again. Thus, the evolution of a particular file may be represented as in Figure A-1.

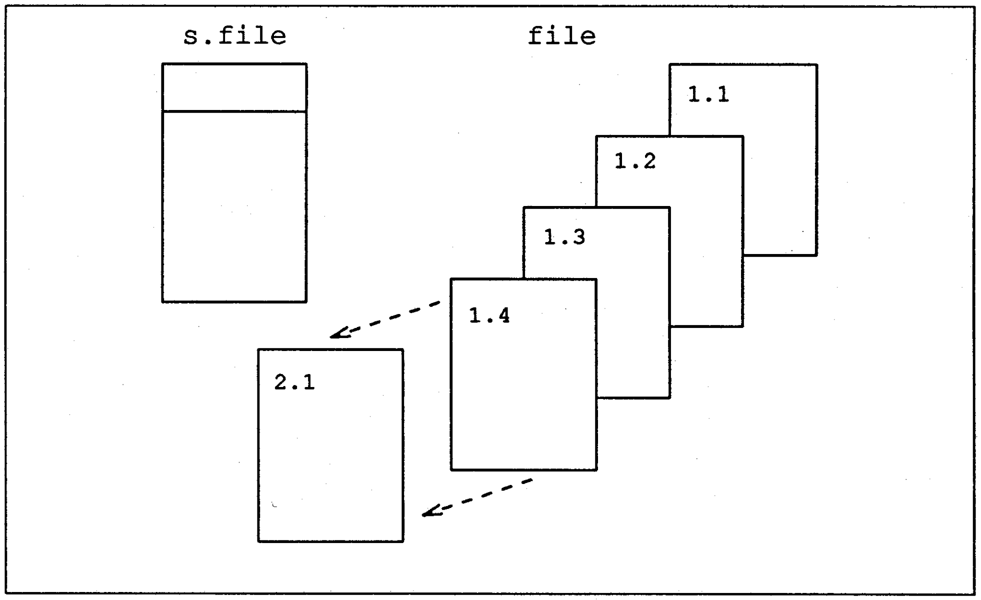


Figure A-1 *Evolution of an SCCS File*

We can call this structure the 'trunk' of the SCCS tree. It represents the normal sequential development of an SCCS file, in which changes that are part of any given delta are dependent upon all the preceding deltas.

Branches

However, there are situations when a branch is needed on the tree: when changes applied as part of a given delta are *not* dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3, and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas, precisely as shown in Figure 1. Assume that a production user reports a problem in version 1.3 which cannot wait until release 2 to be repaired. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user, but will *not* affect the changes being applied for release 2 (that is, deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a 'branch' of the tree, and its name consists of four components: the release and level numbers, as with trunk deltas, plus the 'branch' and 'sequence' numbers. Its SID thus appears as: '*release*.*level*.*branch*.*sequence*. The *branch* number is assigned to each branch that is a descendant of a particular trunk delta; the first such branch is 1, the next one 2, and so on. The *sequence* number is assigned, in order, to each delta on a particular branch. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Figure A-2.

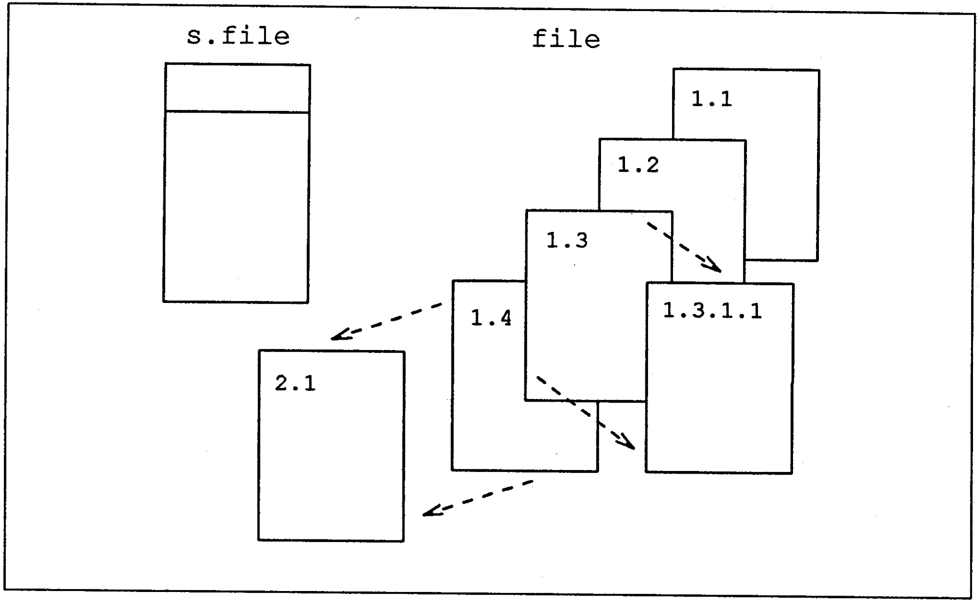


Figure A-2 *Tree Structure with Branch Deltas*

The concept of branching may be extended to any delta in the tree; the naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of a branch delta are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch, independently of its location relative to the trunk delta. Thus, a branch delta may always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is *not* possible to determine the entire path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.*n*. If a delta on this branch then has another branch emanating from it, all deltas on the new branch will be named 1.3.2.*n* (see Figure A-3. The only information that may be derived from the name of delta 1.3.2.2 is that it is the second chronological delta on the second chronological branch whose trunk ancestor is delta 1.3. In particular, it is *not* possible to determine from the name of delta 1.3.2.2 all of the deltas between it and its trunk ancestor (1.3).

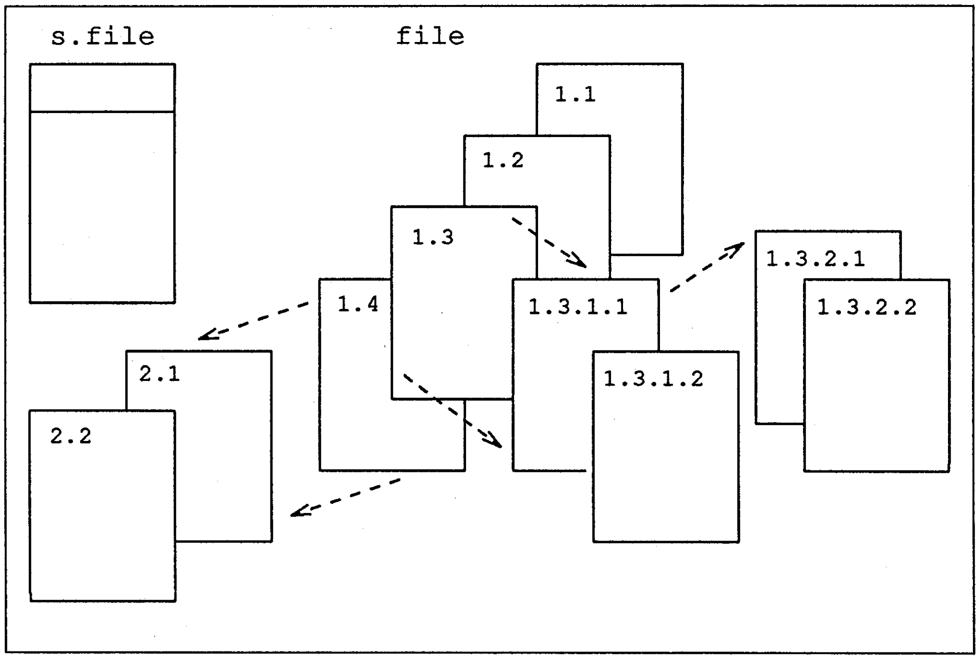


Figure A-3 *Extending the Branching Concept*

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible, because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

A.3. Summary of SCCS Commands

Here is a summary of all the SCCS commands and their major functions:

`admin` Creates SCCS files and applies changes to parameters of SCCS files. `admin` is described in section A.5.

`cdc` Changes the commentary associated with a delta. `cdc` is described in section A.6.

`comb` Combines two or more consecutive deltas of an SCCS file into a single delta. `comb` is described in section A.7.

`delta` Applies changes (deltas) to the text of SCCS files; that is, `delta` creates new versions. `delta` is described in section A.8.

`get` Retrieves versions of SCCS files. `get` is described in section A.9.

`help` Explains SCCS commands and diagnostic messages. `help` is described in section A.10.

`prs` Prints portions of an SCCS file in user-specified format. `prs` is described in section A.11.

`rmdel` Removes a delta from an SCCS file; useful for removing deltas that were created by mistake. `rmdel` is described in section A.12.

`sccsdiff`

Shows the differences between any two versions of an SCCS file. `sccsdiff` is described in section A.14.

`val` Validates an SCCS file. `val` is described in section A.16.

`what` Searches file(s) for all occurrences of a special pattern and prints what follows it. `what` is useful in finding identifying information inserted by `get`. `what` is described in section

A.4. SCCS Command Conventions

This section discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to *all* SCCS commands, except as indicated below.

SCCS commands, like most SunOS commands, accept options and filename arguments.

Options

Options begin with a minus sign (–), followed by a lower-case alphabetic character, and, in some cases, a value. Options modify actions of commands on which they are specified.

Filename Arguments

Filename arguments (which may be names of files and/or directories) specify the file(s) that the given SCCS command is to process; naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable files in the named directories are silently ignored.

In general, file arguments may *not* begin with a minus sign. However, if the name '–' (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line as the name of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the `find(1)` or `ls(1)` commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

Options specified for a given command apply to *all* filename arguments. Options are processed before any file arguments; therefore the placement of options is arbitrary, that is, options may be interspersed with file arguments. File arguments, however, are processed left to right.

Somewhat different argument conventions apply to the `help`, `what`, `sccsdiff`, and `val` commands.

Flags

Certain actions of various SCCS commands are modified by *flags* embedded in the text of SCCS files. Some of these flags are discussed below. For a complete description of all such flags, see `admin`.

Real/Effective User

The distinction between the *real user* (see `passwd(1)`) and the *effective user* ID is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same, that is, the user who is logged into the system.

Back-up Files Created During Processing

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the `x.` file, to ensure that the SCCS file will not be damaged if processing terminates abnormally. The name of the `x.` file is formed by replacing the '`s.`' of the SCCS-file name with '`x.`'. When processing is complete, the old SCCS file is removed and the `x.` file is renamed to take its place. The `x.` file is created in the directory containing the SCCS file, given the same permission mode (see `chmod(1)`), and is owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a *lock file*, called the `z.` file, whose (formed by replacing the '`s.`' with '`z.`'). The `z.` file contains the process ID number of the command that creates it, and its existence is an indication to other commands that that SCCS file is being updated. Thus, other commands that modify SCCS files will not process an SCCS file if the corresponding `z.` file exists. The `z.` file is created with mode 444 (read-only) in the directory containing the SCCS file, and is owned by the effective user. The `z.` file exists only for the duration of the execution of the command that creates it. In general, users can ignore `x.` files and `z.` files; they may be useful in the event of system crashes or similar situations.

Diagnostics

SCCS commands direct their diagnostic responses to the standard error file. SCCS diagnostics generally look like this:

```
ERROR [filename]: message text (code)
```

The *code* in parentheses may be used as an argument to a`help`toobtain

If the SCCS command detects a fatal error during the processing of a file it terminates processing of that file and proceeds with the next file in the series, if more than one file has been named.

A.5. `admin` — Create and Administer SCCS Files

`admin` creates new SCCS files and changes parameters of existing ones. Options and SCCS file names may appear in any order on the `admin` command line. SCCS file names must begin with the characters '`s.`'. A named file is created if it doesn't exist already, and its parameters are initialized according to the specified options. Any parameter not initialized by an option is assigned a default value. If a named file does exist, parameters corresponding to specified options are changed, and other parameters are left as is.

```
admin [ -n][-i[name]][-rrel][-t[name]][-fflag [flag-val]]...
   [-dflag [flag-val]]... [-alogin]...[-elogin]... [-m[mrlist]]
   [-y[comment]][-h][-z]filename ...
```

If a directory is named, `admin` behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with `s.`) and unreadable files are silently ignored. A name of `-` means the standard input — each line of the standard input is taken as the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

`admin` Options

Options are explained as though only one named file is to be processed, since options apply independently to each named file.

Creating a New SCCS File

`-n` A new SCCS file is being created.

Initial Text

`-i` [*name*]

Initial text: file *name* contains the text of a new SCCS file. The text is the first delta of the file — see `-r` option for delta numbering scheme. If *name* is omitted, the text is obtained from the standard input. Omitting the `-i` option altogether creates an empty SCCS file. You can only create one SCCS file with an `admin -i` command. Creating more than one SCCS file with a single `admin` command requires that they be created empty, in which case the `-i` option should be omitted. Note that the `-i` option implies the `-n` option.

Initial Release

`-r` *rel*

Initial release: the *rel*ease into which the initial delta is inserted. `-r` may be used only if the `-i` option is also used. The initial delta is inserted into release 1 if the `-r` option is not used. The level of the initial delta is always 1, and initial deltas are named 1.1 by default.

Descriptive Text

`-t` [*name*]

Descriptive text: The file *name* contains descriptive text for the SCCS file. The descriptive text file name *must* be supplied when creating a new SCCS file (either or both `-n` and `-i` options) and the `-t` option is used. In the case of existing SCCS files: 1) a `-t` option without a file name removes descriptive text (if any) currently in the SCCS file, and 2) a `-t` option with a file name replaces the descriptive text currently in the SCCS file with any text in the named file.

Set a Flag

`-f`*flag*

Set *flag*: specifies a *flag*, and, possibly, a value for the *flag*, to be placed in the SCCS file. Several `-f` options may be supplied on a single `admin` command line. *Flag*s and their values appear in the FLAGS section after this list of options.

Delete a Flag

`-d`*flag*

Delete *flag* from an SCCS file. The `-d` option may be specified only when processing existing SCCS files. Several `-d` options may be supplied on a single `admin` command. See the FLAGS section below.

Unlock Releases

`-l` *list*

Unlock the specified *list* of releases. See the `-f` option for a description of the `l` flag and the syntax of a *list*.

Add Login Name

-a *login*

Add *login* name, or numerical group ID, to the list of users who may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several -a options may appear on a single admin command line. As many *login*s, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, anyone may add deltas.

Erase Login Name

-e *login*

Erase *login* name, or numerical group ID, from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all *login* names common to that group ID. Several -e options may be used on a single admin command line.

Insert Comment Text

-y [*comment*]

The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of delta. If the -y option is omitted, a default comment line is inserted in the form:

```
date and time created yy/mm/dd hh:mm:ss by login
```

The -y option is valid only if the -i and/or -n options are specified (that is, a new SCCS file is being created).

Modification List

-m [*mrlist*]

The list of Modification Requests (MR) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to delta. The v flag must be set and the MR numbers are validated if the v flag has a value (the name of an MR number validation program). Diagnostics are displayed if the v flag is not set or MR validation fails.

Check Structures of SCCS File

-h Check the structure of the SCCS file (see *sccsfile*(5)), and compare a newly computed check-sum (the sum of all the characters in the SCCS file except those in the first line) with the check-sum that is stored in the first line of the SCCS file.

The -h option inhibits writing on the file, so that it nullifies the effect of any other options supplied, and is, therefore, only meaningful when processing existing files.

Recompute Checksum

-z recompute the SCCS file check-sum and store it in the first line of the SCCS file (see **-h**, above).

Using the -z option on a truly corrupted file may prevent future detection of the corruption.

Flags In SCCS Files

The list below is a description of the *flags* which may appear as arguments to the `-f` (set flags) and `-d` (delete flags) options.

Branch Deltas can be Created

`b` When set, the `-b` option can be used on a `get` command to create branch deltas.

Highest Retrievable Release

`c` *ceil*

The highest release (ceiling) which may be retrieved by a `get` command for editing. The ceiling is a number less than or equal to 9999. The default value for an unspecified `c` flag is 9999.

Lowest Retrievable Release

`f` *floor*

The lowest release (floor) which may be retrieved by a `get` command for editing. The floor is a number greater than 0 but less than 9999. The default value for an unspecified `f` flag is 1.

Default Delta Number

`d` *SID*

The default delta number (ID) to be used by a `get` command.

No ID Keywords Fatal Error

`i` Treats the 'No id keywords (ge6)' message issued by `get` or `delta` as a fatal error. In the absence of the `i` flag, the message is only a warning. The message is displayed if no SCCS identification keywords (see `get`) are found in the text retrieved or stored in the SCCS file.

Encoded Binary File

`e 1`

If the `e` flag appears with a `1` argument, the file is an encoded (see `uuencode`(1C) representation of a binary data file.

Allow Concurrent Edits

`j` Concurrent `get` commands for editing may apply to the same SID of an SCCS file. This allows multiple concurrent updates to the same version of the SCCS file.

Locked Releases

`l` *list*

A *list* of locked releases to which deltas can no longer be made. A `get -e` fails when applied against one of these locked releases. The *list* has the following syntax:

< *list* > ::= < *range* > | < *list* > , < *range* >
< *range* > ::= RELEASE NUMBER | `a`

The character `a` in the *list* is equivalent to specifying *all releases* for the named SCCS file.

Create Null Deltas

`n` The `delta` command creates a 'null' delta in each release (if any) being skipped when a delta is made in a *new* release. For example, releases 3 and 4 are skipped when making delta 5.1 after delta 2.7. These null deltas serve as 'anchor points' so that branch deltas may be created from them later. If the `n` flag is absent from the SCCS file, skipped releases will be non-existent in the SCCS file, preventing branch deltas from being created from them in the future.

q *text*
text is defined by the user. The *text* is substituted for all occurrences of the %Q% keyword in SCCS file text retrieved by get.

Module Name

m *module*
Module name of the SCCS file substituted for all occurrences of the %M% keyword in SCCS file text retrieved by get. If the m flag is not specified, the value assigned is the name of the SCCS file with the leading s. removed.

Module Type

t *type*
Type of module in the SCCS file substituted for all occurrences of %Y% keyword in SCCS file text retrieved by get.

Validity Checking Program

v [*program*]
Validity checking *program*: delta prompts for Modification Request (MR) numbers as the reason for creating a delta. The optional *program* specifies the name of an MR number validity checking program (see delta). If this flag is set when creating an SCCS file, the -m option must also be used even if its value is null.

Files Used

The last component of all SCCS file names must be of the form s.*filename*. New SCCS files are given mode 444 (see chmod). Write permission in the pertinent directory is, of course, required to create a file. All writing done by admin is to a temporary x. file, called x.*filename*, (see get(1)), created with mode 444 if the admin command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of *admin*, the SCCS file is removed (if it exists), and the x. file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

It is recommended that directories containing SCCS files be mode 755 and that SCCS files themselves be mode 444. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner allowing use of a text editor. *Care must be taken*! The edited file should *always* be processed by an admin -h to check for corruption followed by an admin -z to generate a proper check-sum. Another admin -h is recommended to ensure the SCCS file is valid.

admin also uses a transient lock file (called z.*filename*, to prevent simultaneous updates to the SCCS file by different users. See get for further information.

Examples of Using `admin`

Suppose you have a file called `lang` that contains a list of programming languages:

```
hermes% cat  lang
C
PL/I
FORTRAN
COBOL
Algol
hermes%
```

We wish to give SCCS custody of 'lang' by using `admin` (which *administer*s SCCS files) to create an SCCS file and initialize delta 1.1. To do so, we use `admin` as shown, and `admin` responds with a message:

```
hermes% admin  -ilang  s.lang
No id keywords (cm7)
hermes%
```

All SCCS files *must* have names that begin with 's.', hence, 's.lang'. The `-i` option, together with its value 'lang', indicates that `admin` is to create a new SCCS file and *initialize* it with the contents of the file 'lang'. This initial version is a set of changes applied to the null SCCS file; it is delta 1.1.

The message is a warning message (which may also be issued by other SCCS commands) that you can ignore for the present.

Remove the file 'lang' now — it can easily be reconstructed with the `get` command, described in section

Inserting Commentary for the Initial Delta

You can use the `-y` and `-m` options with `admin`, just as with `delta`, to insert initial descriptive commentary and/or MR numbers when an SCCS file is created. If you don't use `-y` to comment, `admin` automatically inserts a comment line of the form:

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

If you want to supply MR numbers (`-m` option), the `v` flag must also be set (using the `-f` option described below). The `v` flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a *delta commentary* in the SCCS file (see *sccsfile*(5)). Thus:

```
hermes% admin  -ifirst  -mmrnum1  -fv  s.abc
```

Note that the `-y` and `-m` options are only effective if a new SCCS file is being created.

Initializing and Modifying SCCS File Parameters

The portion of the SCCS file reserved for *descriptive text* may be initialized or changed through the use of the `-t` option. The descriptive text is intended as a summary of the contents and purpose of the SCCS file; actually its contents and length are up to you.

When an SCCS file is being created and the -t option is supplied, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command

```
hermes% admin -ifirst -tdesc s.abc
```

specifies that the descriptive text is to be taken from file 'desc'.

When processing an *existing* SCCS file, the -t option specifies that the descriptive text (if any) currently in the file is to be *replaced* with the text in the named file. Thus:

```
hermes% admin -tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of 'desc'. Omitting the filename after the -t option *removes* the descriptive text from the SCCS file:

```
hermes% admin -t s.abc
```

The *flags* — see the section entitled *Descriptive Text* — of an SCCS file may be initialized and changed with the -f (flag) option, or may be deleted with the -d (delete) option. The flags of an SCCS file direct certain actions of the various commands. See admin for a description of all the flags. For example, the i flag specifies that the warning message stating there are no ID keywords contained in the SCCS file should be treated as an error, and the d (default SID) flag specifies the default version of the SCCS file to be retrieved by the get command. The -f option sets a flag and, possibly, sets its value. For example:

```
hermes% admin -ifirst -fi -fmmodname s.abc
```

sets the i flag and the m (module name) flag. The value 'modname' specified for the m flag is the value that the get command uses to replace the %M% ID keyword. In the absence of the m flag, the name of the g-file is used as the replacement for the %M% ID keyword. Note that several -f options may be supplied on a single admin command, and that -f options may be supplied whether the command is creating a new SCCS file or processing an existing one.

The -d option deletes a flag from an SCCS file, and may only be specified when processing an existing file. As an example, the command:

```
hermes% admin -dm s.abc
```

removes the m flag from the SCCS file. Several -d options may be supplied on a single admin command, and may be interspersed with -f options.

SCCS files contain a list (*user list*) of login names and/or group IDs of users who are allowed to create deltas. This list is normally empty, implying that *anyone* may create deltas. To add login names and/or group IDs to the list, use the admin command with the -a option. For example:

```
hermes% admin -awendy -aalison -a1234 s.abc
```

adds the login names 'wendy' and 'alison' and the group ID '1234' to the list. The -a option may be used whether admin is creating a new SCCS file or processing an existing one, and may appear several times. The -e option is used in an analogous manner if one wishes to remove ('erase') login names or group IDs from the list. A.9.

A.6. cdc — Change Delta Commentary

cdc changes the *delta commentary*, for the SID specified by the -r option, of each named SCCS file.

```
cdc -rSID [-m[ mrlist ] ] [ -y [ comment ] ] filename ...
```

Delta commentary is defined to be the Modification Request (MR) and comment information normally specified via the delta command (-m and -y options).

If a directory is named, cdc behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with s.) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see the NOTES below) each line of the standard input is taken to be the name of an SCCS file to be processed.

Arguments to cdc, which may appear in any order, consist of options and file names.

cdc Options

All the described options apply independently to each named file:

SID *Identification String*

-r*SID*

Specifies the *SCCS IDentification* string of a delta for which the delta commentary is to be changed.

MR List

-m[*mrlist*]

If the SCCS file has the v flag set (see admin), a list of MR numbers to be added and/or deleted in the delta commentary of the SID specified by the -r option *may* be supplied. A null MR list has no effect.

MR entries are added to the list of MRs in the same manner as that of delta. To delete an MR, precede the MR number with the character ! (see EXAMPLES. If the MR to be deleted is currently in the list of MRs, it is removed and changed into a ''comment'' line. A list of all deleted MRs is placed in the comment section of the delta commentary and preceded by a comment line stating that they were deleted.

If -m is not used and the standard input is a terminal, the prompt MRs? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The MRs? prompt always precedes the comments? prompt (see -y option).

MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the MR list.

Note that if the v flag has a value (see admin), it is taken to be the name of a program (or shell procedure) which validates the correctness of the MR numbers. If a non-zero exit status is returned from the MR number validation program, cdc terminates and the delta commentary remains unchanged.

Comment Text

-y[*comment*]

Arbitrary text used to replace the *comment*(s) already existing for the delta specified by the -r option. The previous comments are kept and preceded by a comment line stating that they were changed. A null *comment* has no effect.

If -y is not specified and the standard input is a terminal, the prompt comments? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the *comment* text.

Examples of Using cdc

```
hermes% cdc -r1.6 -m"bl78-12345 !bl77-54321 bl79-00001" -ytrouble s.file
```

adds bl78-12345 and bl79-00001 to the MR list, removes bl77-54321 from the MR list, and adds the comment trouble to delta 1.6 of s.file.

```
hermes% cdc -r1.6 s.file
MRs? !bl77-54321 bl78-12345 bl79-00001
comments? trouble
```

does the same thing.

NOTE *If SCCS file names are supplied to the* cdc *command via the standard input (*- on the command line), then the -m and -y options must also be used.

Files Used

x.file (see delta)
z.file (see delta)

A.7. comb — Combine SCCS Deltas

comb generates a Bourne Shell procedure which, when run, will reconstruct the given SCCS files.

```
comb [ -o ] [ -s ] [ -p SID ] [ -c list ] filename ...
```

If a directory is named, comb behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with s.) and unreadable files are silently ignored. If a name of - is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored. The generated shell procedure is written on the standard output.

comb Options

Options are explained as though only one named file is to be processed, but the effects of any option apply independently to each named file.

ID String

-p *SID*
The *S*CCS *ID*entification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

Preserve List

-c *list*
A *list* of deltas to be preserved. All other deltas are discarded. See get for the syntax of a *list*.

Access at Release

-o For each get -e generated, the reconstructed file is accessed at the release of the delta to be created. In the absence of the -o option, the reconstructed file is accessed at the most recent ancestor. Use of the -o option may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

Generate Report

-s Generate a shell procedure which, when run, will produce a report giving, for each file: the file name, size (in blocks) after combining, original size (also in blocks), and percentage change computed by:

100 * (original   combined) / original

It is recommended that before any SCCS files are actually combined, you should use this option to determine exactly how much space is saved by the combining process.

If no options are specified, comb preserves only leaf deltas and the minimal number of ancestors needed to preserve the tree.

Files Used

s.COMB
The name of the reconstructed SCCS file.
comb?????
Temporary.

Limitations of the comb Command

comb may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file to actually be larger than the original.

A.8. delta — Make a Delta

delta permanently introduces into the named SCCS file changes that were made to the file retrieved by get (called the ,g-file or generated file).

```
delta [ -r SID ] [ -s ] [ -n ][ -g list ] [ -m [ mrlist ] ][ -y [ comment ] ] [ -p ] filename ...
```

delta makes a delta to each named SCCS file. If a directory is named, delta behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with s.) and unreadable files are silently ignored. If a name of - is given, the standard input is read (see WARNINGS; each line of the standard input is taken to be the name of an SCCS file to be processed.

delta may issue prompts on the standard output depending upon certain options specified and flags (see admin) that may be present in the SCCS file (see -m and -y options below).

delta Options

Options apply independently to each named file.

Delta Number

-r *SID*
Uniquely identifies which delta is to be made to the SCCS file. The use of this option is necessary only if two or more outstanding *get*'s for editing (**get -e**) on the same SCCS file were done by the same person (login name). The SID value specified with the -r option can be either the SID specified on the get command line or the SID to be made as reported by the get command (see get). A diagnostic results if the specified SID is ambiguous, or, if necessary and omitted on the command line.

No Report

-s Do not display the created delta's ID, number of lines inserted, deleted and unchanged in the SCCS file.

Retain g-file

-n Retain the edited g-file which is normally removed at completion of delta processing.

Ignore List

-g *list*
Specifies a *list* of deltas to be *ignored* when the file is accessed at the change level (ID) created by this delta. See get for the definition of *list*.

MR Number

-m [*mrlist*]
If the SCCS file has the v flag set (see admin), a Modification Request (MR) number *must* be supplied as the reason for creating the new delta.

If -m is not used and the standard input is a terminal, the prompt MRs? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The MRs? prompt always precedes the comments? prompt (see -y option).

MR's in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the MR list.

Note that if the v flag has a value (see admin), it is taken to be the name of a program (or shell procedure) which will validate the correctness of the MR numbers. If a non-zero exit status is returned from MR number validation program, delta terminates (it is assumed that the MR numbers were not all valid).

Comment Text

-y [*comment*]
Arbitrary text to describe the reason for making the delta. A null string is considered a valid *comment*.

If -y is not specified and the standard input is a terminal, the prompt comments? is issued on the standard output before the standard input is read;

if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the comment text.

Display Differences

`-p` Display (on the standard output) the SCCS file differences before and after the delta is applied in a `diff` format.

Files Used

g-file — Existed before the execution of `delta`; removed after completion of `delta`.

`p.`file — Existed before the execution of `delta`; may exist after completion of `delta`.

`q.`file — Created during the execution of `delta`; removed after completion of `delta`.

`x.`file — Created during the execution of `delta`; renamed to SCCS file after completion of `delta`.

`z.`file — Created during the execution of `delta`; removed during the execution of `delta`.

`d.`file — Created during the execution of `delta`; removed after completion of `delta`.

`/bin/diff`
Program to compute differences between the ''gotten'' file and the g-file.

NOTE *Lines beginning with an ASCII SOH character (binary 001) cannot be placed in the SCCS file unless the* SOH *is escaped. This character has special meaning to SCCS (see* `sccsfile(5)`*) and will cause an error.*

NOTE *A* `get` *of many SCCS files, followed by a* `delta` *of those files, should be avoided when the* `get` *generates a large amount of data. Instead, multiple* `get`/`delta` *sequences should be used.*

NOTE *If the standard input (*`-`*) is specified on the* `delta` *command line, the* `-m` *(if necessary) and* `-y` *options must also be present. Omission of these options is an error.*

Examples of Using `delta`

To record the changes that were applied to 'lang' within the SCCS file, use the `delta` command. `delta` asks for comments describing the change, and you respond with a description of why the changes were made:

```
hermes% delta  s.lang
comments? added SNOBOL and Ratfor
        More messages from delta — see below
hermes%
```

`delta` then reads the `p.`file and determines what changes were made to the file `lang`. `delta` does this by doing its own `get` to retrieve the original version, and then applying `diff(1)` to the original version and the edited version. When the changes to 'lang' have been stored in 's.lang', the dialogue with `delta` looks like:

```
hermes% delta  s.lang
comments? added SNOBOL and Ratfor
1.2
2 inserted
0 deleted
5 unchanged
hermes%
```

The number '1.2' is the name of the delta just created, and the next three lines are a summary of the changes made to 's.lang'.

More Notes on delta

delta does a series of checks before creating the delta:

1. Searches the p.file for an entry containing the user's login name, because the user who retrieved the g-file must be the one who creates the delta. delta displays an error message if the entry is not found. Note that if the login name of the user appears in more than one entry (that is, the same user did a get -e more than once on the same SCCS file), the -r option must be used with delta to specify an SID that uniquely identifies the p.file entry[40].
2. Performs the same permission checks as get -e.

If these checks succeed, delta compares the g-file (via diff(1)) with its own, temporary copy of the g-file as it was before editing, to determine what has been changed. This temporary copy of the g-file is called the d.file (its name is formed by replacing the s. of the SCCS file name with d.); delta retrieves it by doing its own get at the SID specified in the p.file entry. If you would like to see the results of delta's diff, use the -p option to display it on standard output.

In practice, the most common use of delta is:

```
hermes% delta  s.abc
```

If your standard output is a terminal, delta replies: 'comments?'. You may now type a response — usually a description of why the delta is being made — of up to 512 characters, terminating with a newline character. Newline characters *not* intended to terminate the response should be preceded by '\'.

If the SCCS file has a v flag, delta asks for 'MRs?' before prompting for 'comments?' (again, this prompt is printed only if the standard output is a terminal). Enter MR[41] numbers, separated by blanks and/or tabs, and terminate your response with a newline character.

If you want to enter commentary (comments and/or MR numbers) directly on the command line, use the -y and/or -m options, respectively. For example:

[40] The SID specified may be either the SID retrieved by get, or the SID delta is to create.

[41] In a tightly controlled environment, one would expect deltas to be created only as a result of some trouble report, change request, trouble ticket, etc. (collectively called here Modification Requests, or MRs) and would think it desirable or necessary to record such MR number(s) within each delta.

```
hermes% delta -y"descriptive comment" -m"mrnum1 mrnum2" s.abc
```

inserts the 'descriptive comment' and the MR numbers 'mrnum1' and 'mrnum2' without prompting or reading from standard input. `-m` can only be used if the SCCS file has a `v` flag. These options are useful when `delta` is executed from within a Shell procedure.

The commentary (comments and/or MR numbers), whether solicited by `delta` or supplied via options, is recorded as part of the entry for the delta being created, and applies to *all* SCCS files processed by the same invocation of `delta`. Thus if `delta` is used with more than one file argument, and the first file named has a `v` flag, all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Only files conforming to these rules are processed.

After the prompts for commentary, and before any other output, `delta` displays:

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

if it finds no ID keywords in the edited `g`-file while making a delta. If there *were* any ID keywords in the SCCS file, this might mean one of two things. The keywords may have been replaced by their values (if a `get` without the `-e` option was used to retrieve the `g`-file). Or, the keywords may have been accidentally deleted or changed while editing the `g`-file. Of course, the file may never have had any ID keywords. In any case, it is left up to you to decide whether any action is necessary, but the delta is made regardless (unless there is an `i` flag in the SCCS file, which makes this a fatal error and kills the delta).

When processing is complete, `delta` displays a message containing the SID of the created delta (obtained from the `p`. file entry), and the counts of lines inserted, deleted, and left unchanged. Thus, a typical message might be:

```
1.4
14 inserted
7 deleted
345 unchanged
```

The reported counts may not agree with your sense of changes made; there are a number of ways to describe a set of such changes, especially if lines are moved around in the `g-file`, and `delta` may describe the set differently than you. However, the *total* number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited `g`-file.

After processing of an SCCS file is complete, the corresponding `p`. file entry is removed from the `p.file`[42]. If there is only *one* entry in the `p`. file, the `p`. file itself is removed.

[42] All updates to the `p`. file are made to a temporary copy, the `q`. file, whose use is similar to the use of the `x`. file described above.

In addition, delta removes the edited g-file, unless the -n option is specified. Thus:

```
hermes% delta -n s.abc
```

keeps the g-file upon completion of processing.

The -s (silent) option suppresses all output that is normally directed to the standard output, except the initial prompts for commentary. If you use -s with -y (and, possibly, -m), delta neither reads standard input nor writes to standard output.

A.9. get — Get Version of SCCS File

get generates an ASCII text file from each named SCCS file according to the specified option. Arguments may be specified in any order, options apply to all named SCCS files. If a directory is named, get behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of - is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

```
get [ -rSID ] [ -ccutoff ] [ -ilist ][ -xlist ] [ -aseq-no. ] [ -k ] [ -e ]
        [ -l[p]] [ -p ] [ -m ] [ -n ] [ -s ] [ -b ] [ -g ] [ -t ] filename ...
```

The generated text is normally written into a file called the g-file (whose name is derived from the SCCS file name by simply removing the leading **s.**; see also *FILES*, below).

get Options

Options are explained below as though only one SCCS file is to be processed, but the effects of any option argument applies independently to each named file.

ID String

-r *SID*

The string (ID) of the version (delta) of an SCCS file to be retrieved. Table 1 below shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the ID of the version to be eventually created by delta if the -e option is also used), as a function of the SID specified.

Cutoff

-c *cutoff*

Cutoff date-time, in the form: YY[MM[DD[HH[MM[SS]]]]]

No changes (deltas) to the SCCS file which were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, -c7502 is equivalent to -c750228235959. Any number of non-numeric characters may separate the various 2 digit pieces of the *cutoff* date-time. This feature allows one to specify a *cutoff* date in the form: -c77/2/2 9:22:25. Note that this implies that one may use the %E% and %U% identification keywords.

Get for Editing

-e This get is for editing or making a change (delta) to the SCCS file via a subsequent use of delta. A get -e applied to a particular version (ID) of the SCCS file prevents further get -e commands on the same SID until delta is run or the j (joint edit) flag is set in the SCCS file (see admin). Concurrent use of get -e for different IDs is always allowed.

If the g-file generated by a get -e is accidentally ruined in the process of editing it, it may be regenerated by re-running a get with the -k option in place of the -e option.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file (see admin) are enforced when the -e option is used.

New Branch

-b Used with the -e option to indicate that the new delta should have an SID in a new branch as shown in Table 1. This option is ignored if the b flag is not present in the file (see admin) or if the retrieved delta is not a leaf *delta*. A leaf delta is one that has no successors on the SCCS file tree.

NOTE *A branch delta may always be created from a non-leaf delta.*

Include List

-i *list*

A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

< *list* > ::= < *range* > | < *list* > , < *range* >
< *range* > ::= ID | ID-ID

ID, the SCCS Identification of a delta, may be in any form shown in the 'ID Specified' column of Table 1. Partial IDs are interpreted as shown in the 'ID Retrieved' column of Table 1.

Exclude List

-x *list*

A *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the -i option for the *list* format.

Don't Expand ID Keywords

-k Do not replace identification keywords (see below) in the retrieved text by their value. The -k option is implied by the -e option.

Write Delta Summary

-l[p]

Write a delta summary into an l.file. If -lp is used, the delta summary is written on the standard output and the l.file is not created. See *FILES* for the format of the l.file.

Write Text to Standard Output

-p Write the text retrieved from the SCCS file to the standard output. No g-file is created. All output which normally goes to the standard output goes to the standard error file instead, unless the -s option is used, in which case it disappears.

Suppress All Output

-s Suppress all output normally written on the standard output. However, fatal error messages (which always go to the standard error file) remain unaffected.

Show delta IDs

-m Precede each text line retrieved from the SCCS file with the ID of the delta that inserted the text line in the SCCS file. The format is: ID, followed by a horizontal tab, followed by the text line.

Show Module Names

-n Precede each generated text line with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the -m and -n options are used, the format is: %M% value, followed by a horizontal tab, followed by the -m option generated format.

Don't Retrieve Text

-g Do not actually retrieve text from the SCCS file. It is primarily used to generate an l. file, or to verify the existence of a particular ID.

Access Top Delta

-t Access the most recently created ('top') delta in a given release (for example, **-r1**), or release and level (for example, **-r1.2**).

Delta Sequence Number

-a *seq-no.*
The delta sequence number of the SCCS file delta (version) to be retrieved (see *sccsfile*(5)). This option is used by the comb command; it is not a generally useful option, and users should not use it. If both the -r and -a options are specified, the -a option is used. Care should be taken when using the -a option in conjunction with the -e option, as the SID of the delta to be created may not be what one expects. The -r option can be used with the -a and -e options to control the naming of the SID of the delta to be created.

For each file processed, get responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the -e option is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a new-line) before it is processed. If the -i option is used included deltas are listed following the notation 'Included'; if the -x option is used, excluded deltas are listed following the notation 'Excluded'.

Table A-1 *Determination of SCCS Identification String*

SID* Specified	`-b` Option Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	—	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB+1).1
R	—	Trunk succ.# in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	no	No trunk succ.	R.L	R.(L+1)
R.L	yes	No trunk succ.	R.L	R.L.(mB+1).1
R.L	—	Trunk succ. in release ≥ R	R.L	R.L.(mB+1).1
R.L.B	no	No branch succ.	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch succ.	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch succ.	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch succ.	R.L.B.S	R.L.(mB+1).1
R.L.B.S	—	Branch succ.	R.L.B.S	R.L.(mB+1).1

* 'R', 'L', 'B', and 'S' are the 'release', 'level', 'branch', and 'sequence' components of the SID, respectively; 'm' means 'maximum'. Thus, for example, 'R.mL' means 'the maximum level number within release R'; 'R.L.(mB+1).1' means 'the first sequence number on the *new* branch (that is, maximum branch number plus one) of level L within release R'. Note that if the SID specified is of the form 'R.L', 'R.L.B', or 'R.L.B.S', each of the specified components *must* exist.

** 'hR' is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

***

Forces creation of the *first* delta in a *new* release.

# Successor.

† The `-b` option is effective only if the `b` flag (see `admin`) is present in the file. An entry of - means 'irrelevant'.

‡ This case applies if the `d` (default SID) flag is *not* present in the file. If the `d` flag *is* present in the file, the SID obtained from the `d` flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

Identification Keywords

When you generate a g-file to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc., within the g-file, so that this information appears in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. *Identification* (ID) *keywords* appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords.

The format of an ID keyword is an upper-case letter enclosed by percent signs (%). For example, %I% is an ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, %H% is an ID keyword for the current date (in the form 'mm/dd/yy'), and %M% is the name of the g-file.

Thus, using get on an SCCS file that contains the C declaration:

```
char  identification [ ] = "%M%  %I%  %H%";
```

gives (for example) the following:

```
char  identification [ ] = "modulename  2.3  03/17/83";
```

If there are no ID keywords in the text, get might display:

```
No id keywords (cm7)
hermes%
```

This message is normally treated as a warning by get.
However, if an i flag is present in the SCCS file, it is treated as an error — see section A.8 for further information.

Table A-2 *Identification Keywords*

Keyword	*Value*
%M%	Module name: either the value of the m flag in the file (see admin), or if absent, the name of the SCCS file with the leading s. removed.
%I%	SCCS identification (ID) (%R%.%L%.%B%.%S%) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%T%	Current time (HH:MM:SS).
%E%	Date newest applied delta was created (YY/MM/DD).
%G%	Date newest applied delta was created (MM/DD/YY).
%U%	Time newest applied delta was created (HH:MM:SS).
%Y%	Module type: value of the t flag in the SCCS file (see admin).
%F%	SCCS file name.

Table A-2 *Identification Keywords— Continued*

Keyword	*Value*
%P%	Fully qualified SCCS file name.
%Q%	The value of the q flag in the file (see admin).
%C%	Current line number. This keyword is intended for identifying messages output by the program such as 'this shouldn't have happened' type errors. It is *not* intended to be used on every line to provide sequence numbers.
%Z%	The 4-character string @(#) recognizable by what.
%W%	A shorthand notation for constructing what strings for program files. %W% = %Z%%M%<*tab*>%I%
%A%	Another shorthand notation for constructing what strings for nonstandard program files. %A% = %Z%%Y% %M% %I%%Z%

Retrieving Different Versions

You can retrieve versions other than the default version of an SCCS file by using various options. Normally, the default version is the most recent delta of the highest-numbered release on the *trunk* of the SCCS file tree. However, if the SCCS file being processed has a d (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the -r option of get.

The -r option specifies an SID to be retrieved, in which case the d (default SID) flag (if any) is ignored. For example, to retrieve version 1.3 of file 's.abc', type:

```
hermes% get  -r1.3  s.abc
1.3
64 lines
hermes%
```

A branch delta may be retrieved in the same way:

```
hermes% get  -r1.5.2.3  s.abc
1.5.2.3
234 lines
hermes%
```

When a two- or four-component SID is specified as a value for the -r option (as above) and the particular version does not exist in the SCCS file, an error message results.

If you omit the level number of the SID, get retrieves the *trunk* delta with the highest level number within the given release, if the given release exists:

```
hermes% get  -r3  s.abc
3.7
213 lines
hermes%
```

get retrieved delta 3.7, the highest level trunk delta in release 3. If the given release does not exist, get goes to the next-highest existing release, and retrieves the *trunk* delta with the highest level number. For example, if release 9 does not exist in file 's.abc', and release 7 is actually the highest-numbered release below 9, then get would generate:

```
hermes% get  -r9  s.abc
7.6
420 lines
hermes%
```

indicating that trunk delta 7.6 is the latest version of file 's.abc' below release 9.

Similarly, if you omit the sequence number of an SID, as in:

```
hermes% get  -r4.3.2  s.abc
4.3.2.8
89 lines
hermes%
```

get retrieves the branch delta with the highest sequence number on the given branch, if it exists. If the given branch does not exist, an error message results.

The -t option retrieves the latest ('top') version in a particular *release* (that is, when no -r option is supplied, or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is trunk delta 3.5, doing a get -t on release 3 produces:

```
hermes% get  -r3  -t  s.abc
3.5
59 lines
hermes%
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command produces:

```
hermes% get  -r3  -t  s.abc
3.2.1.5
46 lines
hermes%
```

Retrieving to Make Changes

Specifying the -e option to the get command indicates the intent to make a delta sometime later, and, as such, its use is restricted. If the -e option is present, get checks the following things:

1. The *user list*, the list of *login* names and/or *group IDs* of users allowed to make deltas, to determine if the login name or group ID of the user executing get is on that list. Note that a *null* (empty) user list behaves as if it contained *all* possible login names.

2. That the *release* (R) of the version being retrieved satisfies the relation:

   ```
   floor ≤ R ≤ ceiling
   ```

   to determine if the release being accessed is a protected release. The *floor* and *ceiling* are specified as *flags* in the SCCS file.

3. That the *release* (R) is not *locked* against editing. The *lock* is specified as a flag in the SCCS file.

4. Whether or not *multiple concurrent edits* are allowed for the SCCS file as specified by the j flag in the SCCS file. Multiple concurrent edits are described in the section entitled *Concurrent Edits of the Same SID*.

get terminates processing of the corresponding SCCS file if any of the first three conditions fails.

If the above checks succeed, get with the -e option creates a g-file in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user.

get terminates with an error if a *writable* g-file already exists — this is to prevent inadvertent destruction of a g-file that already exists and is being edited for the purpose of making a delta.

ID keywords appearing in the g-file are *not* substituted by get when the -e option is specified, because the generated g-file is to be subsequently used to create another delta, and replacement of ID keywords would permanently change them within the SCCS file. In view of this, get does not check for the presence of ID keywords within the g-file, so that the message: 'No id keywords (cm7)' is never displayed when get is invoked with the -e option.

In addition, a get with the -e option creates (or updates) a p.file, for passing information to the delta command. Let's look at an example of get -e:

```
hermes% get -e s.abc
1.3
new delta 1.4
67 lines
hermes%
```

The message indicates that get has retrieved version 1.3, which has 67 lines; the version delta will create is version 1.4.

If the -r and/or -t options are used together with the -e option, the version retrieved for editing is as specified by the -r and/or -t options.

The options -i and -x may be used to specify a list of deltas to be *included* and *excluded*, respectively, by get. See get for the syntax of such a list.

'Including a delta' forces the changes that constitute the particular delta to be included in the retrieved version — this is useful for applying the same changes to more than one version of the SCCS file. 'Excluding a delta' forces it *not* to be applied. This is useful for undoing the effects of a previous delta in the version of the SCCS file to be created.

Whenever deltas are included or excluded, `get` checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. Two deltas can interfere, for example, when each one changes the same line of the retrieved `g`-file. Any interference is indicated by a warning that displays the range of lines within the retrieved `g`-file in which the problem may exist. The user is expected to examine the `g`-file to determine whether a problem actually exists, and to take whatever corrective measures are deemed necessary.

NOTE *The* `-i` *and* `-x` *options should be used with extreme care.*

The `-k` option to `get` can be used to regenerate a `g`-file that may have been accidentally removed or ruined after executing `get` with the `-e` option, or to simply generate a `g`-file in which the replacement of ID keywords has been suppressed. Thus, a `g`-file generated by the `-k` option is identical to one produced by `get` executed with the `-e` option. However, no processing related to the `p.` file takes place.

Concurrent Edits of Different SIDs

The ability to retrieve different versions of an SCCS file allows a number of deltas to be 'in progress' at any given time. In general, several people may simultaneously edit the same SCCS file provided they are editing *different versions* of that file. This is the situation we discuss in this section. However, there is a provision for multiple concurrent edits, so that more than one person can edit the *same version* — see the section entitled *Concurrent Edits of the Same SID*.

The `p.` file — created via a `get -e` command — is named by replacing the 's.' in the SCCS file name with 'p.'. The `p.` file is created in the directory containing the SCCS file, is given mode 644 (readable by everyone, writable only by the owner), and is owned by the effective user. The `p.` file contains the following information for each delta that is still 'in progress':[43]

□ The SID of the retrieved version.

□ The SID that will be given to the new delta when it is created.

□ The login name of the real user executing `get`.

The first execution of `get -e` *creates* the `p.` file for the corresponding SCCS file. Subsequent executions only *update* the `p.` file by inserting a line containing the above information. Before inserting this line, however, `get` performs two checks. First, it searches the entries in the `p.` file for an SID which matches that of the requested version, to make sure that the requested version has not already been retrieved. Secondly, `get` determines whether or not multiple concurrent edits are allowed. If the requested version has been retrieved and multiple

[43] Other information may be present, but is not of concern here. See `get` for further discussion.

concurrent edits are not allowed, an error message results. Otherwise, the user is informed that other deltas are in progress, and processing continues.

It is important to note that the various executions of `get` should be carried out from different directories. Otherwise, only the first use of `get` will succeed; since subsequent `gets` would attempt to overwrite a *writable* g-file, they produce an SCCS error condition. In practice, this problem does not arise: normally such multiple executions are performed by different users[44] from different working directories.

Table A-1 shows, for the most useful cases, what version of an SCCS file is retrieved by `get`, as well as the SID of the version to be eventually created by `delta`, as a function of the SID specified to `get`.

Concurrent Edits of the Same SID

Normally, `gets` for editing (`-e` option specified) cannot operate concurrently on the same SID. Usually `delta` must be used before another `get -e` on the same SID. However, multiple concurrent edits (two or more *successive* `get -e` commands based on the same retrieved SID) *are* allowed if the `j` flag is set in the SCCS file. Thus:

```
hermes% get  -e  s.abc
1.1
new delta 1.2
5 lines
hermes%
```

may be immediately followed by:

```
hermes% get  -e  s.abc
1.1
new delta 1.1.1.1
5 lines
hermes%
```

without an intervening use of `delta`. In this case, a `delta` command corresponding to the first `get` produces delta 1.2 (assuming 1.1 is the latest (most recent) trunk delta), and the `delta` command corresponding to the second `get` produces delta 1.1.1.1.

Options that Affect Output

When the `-p` option is specified, `get` writes the retrieved text to the standard output, rather than to a g-file. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create g-files with arbitrary names:

```
hermes% get  -p  s.abc  >  arbitrary-filename
```

[44] See the section entitled *Protection* for a discussion of how different users can use SCCS commands on the same files.

The -s option suppresses all output that is *normally* directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, and so on, do not appear on the standard output. -s does not affect messages directed to the diagnostic output. -s is often used in conjunction with the -p option to 'pipe' the output of get, as in:

```
hermes% get -p -s s.abc | nroff
```

A get -g verifies the existence of a particular SID in an SCCS file but does not actually retrieve the text. This may be useful in a number of ways. For example,

```
hermes% get -g -r4.3 s.abc
```

displays the specified SID if it exists in the SCCS file, and generates an error message if it doesn't. -g can also be used to regenerate a p. file that has been destroyed:

```
hermes% get -e -g s.abc
```

get used with the -l option creates an l. file, which is named by replacing the 's.' of the SCCS file name with 'l.'. This file is created in the current directory, with mode 444 (read-only), and is owned by the real user. It contains a table (format described in get) showing which deltas were used in constructing a particular version of the SCCS file. For example:

```
hermes% get -r2.3 -l s.abc
```

generates an l. file showing which deltas were applied to retrieve version 2.3 of the SCCS file. Specifying a *value* of 'p' with the -l option, as in:

```
hermes% get -lp -r2.3 s.abc
```

sends the generated output to the standard output rather than to the l.file. Note that the -g option may be used with the -l option to suppress the actual text retrieval.

The -m option identifies the origin of each change applied to an SCCS file. -m tags each line of the generated g-file with the SID of the delta it came from. The SID precedes the line, and is separated from the text by a tab character.

When the -n option is specified, each line of the generated g-file is preceded by the value of the %M% ID keyword and a tab character. The -n option is most often used in a pipeline with grep(1).
For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory:

```
hermes% get -p -n -s directory | grep pattern
```

If both the -m and -n options are specified, each line of the generated g-file is preceded by the value of the %M% ID keyword and a tab (the effect of the -n option), followed by the line in the format produced by the -m option.

Since using the -m option, the -n option, or both, modifies the contents of the g-file, such a g-file must *not* be used for creating a delta. Therefore, neither the -m nor the -n option may be used with the -e option.

Files Used

Several auxiliary files may be created by *get*, These files are known generically as the g-file, l.file, p.file, and z.file. The letter before the "dot" is called the tag. The current version, or ''g-file has no tag. An auxiliary file name is based on the format of the SCCS-file name: the last component of the SCCS-file name is of the form s.*version-name*, the auxiliary files are named by replacing the leading s. with the tag. The g-file is an exception to this scheme: its name is derived by removing the s. prefix. For example, for s.xyz.c, the auxiliary file names would be xyz.c (g-file), l.xyz.c, p.xyz.c, and z.xyz.c.

g-file

The g-file, which contains the generated text, is created in the current directory (unless the -p option is used). A g-file is created in all cases, whether or not any lines of text were generated by the get. It is owned by the real user. If the -k option is used or implied its mode is 644; otherwise its mode is 444. Only the real user need have write permission in the current directory.

l-file

The l.file contains a table showing which deltas were applied in generating the retrieved text. The l.file is created in the current directory if the -l option is used; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory.

Format of Lines in the l-file

Lines in the l.file have the following format:

a. A blank character if the delta was applied; * otherwise.
b. A blank character if the delta was applied or wasn't applied and ignored; * if the delta wasn't applied and wasn't ignored.
c. A code indicating a 'special' reason why the delta was or was not applied:
'I': Included.
'X': Excluded.
'C': Cut off (by a -c option).
d. Blank.
e. SCCS identification (ID).
f. Tab character.
g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation.
h. Blank.
i. Login name of person who created delta.

The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

p.file

The p.file passes information resulting from a get -e along to *delta*. Its contents are also used to prevent a subsequent execution of a get -e for the same SID until delta is executed or the joint edit flag, j, (see admin) is set in the SCCS file. The p.file is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the p.file is: the gotten ID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the `get` was executed, followed by a blank and the `-i` option if it was present,. followed by a blank and the `-x` option if it was present, followed by a new-line. There can be an arbitrary number of lines in the `p.` file at any time; no two lines can have the same new delta ID.

z-file

The z-file serves as a *lock-out* mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command (that is, `get`) that created it. The `z.` file is created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the `p.` file apply for the z-file. The z-file is created mode 444.

Limitations of the `get` Command

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user doesn't, only one file may be named when the `-e` option is used.

A.10. `help` — Ask for SCCS Help

`help` finds information to explain a message from a command or explain the use of a command. Zero or more arguments may be supplied. If no arguments are given, `help` prompts for one.

```
help [args]
```

The arguments may be either message numbers (which normally appear in parentheses following messages) or command names, of one of the following types:

type 1 Begins with non-numerics, ends in numerics. The non-numeric prefix is usually an abbreviation for the program or set of routines which produced the message (for example, `ge6`, for message 6 from the `get` command).

type 2 Does not contain numerics (as a command, such as **get**)

type 3 Is all numeric (for example, `212`)

The response of the program is the explanatory information related to the argument, if there is any.

When all else fails, try `help stuck`.

Example of `help`

The following asks for help on the `ge5` error message and information about the `rmdel` command:

```
hermes% help ge5 rmdel
ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
        rmdel -rSID name ...
hermes%
```

Files Used

/usr/lib/help
directory containing files of message text.

A.11. prs — Print SCCS File

prs prints, on the standard output, parts or all of an SCCS file (see sccsfile(5)) in a user supplied format. If a directory is named, prs behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with s.), and unreadable files are silently ignored. If a name of - is given, the standard input is read, in which case each line is taken to be the name of an SCCS file or directory to be processed; non-SCCS files and unreadable files are silently ignored.

```
prs [ -d[ dataspec ] ] [ -r[ SID ] ][ -e ] [ -l ] [ -a ] filename ...
```

prs Options

Options apply independently to each named file.

Output data specification

-d [*dataspec*]
Specifies the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see A.11.2) interspersed with optional user supplied text.

ID string

-r [*SID*]
Specifies the *S*CCS *ID*entification (ID) string of a delta for which information is desired. If no SID is specified, the SID of the most recently created delta is assumed.

Information on earlier deltas

-e Requests information for all deltas created *earlier* than and including the delta designated via the -r option.

Information on later deltas

-l Requests information for all deltas created *later* than and including the delta designated via the -r option.

Information for all deltas

-a Requests printing of information for both removed, that is, delta type = *R*, (see rmdel) and existing, that is, delta type = *D*, deltas. If the -a option is not specified, information for existing deltas only is provided.

In the absence of the `-d` options, `prs` displays a default set of information consisting of: delta-type, release number and level number, date and time last changed, user-name of the person who changed the file, lines inserted, changed, and unchanged, the MR numbers, and the comments.

Data Keywords

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see `sccsfile`(5)) have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information printed by `prs` consists of: 1) the user supplied text; and 2) appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is either *Simple* (S), in which keyword substitution is direct, or *Multi-line* (M), in which keyword substitution is followed by a carriage return.

User supplied text is any text other than recognized data keywords. A tab is specified by \t and carriage return/new-line is specified by **\n**.

Table A-3 *SCCS Files Data Keywords*

Keyword	*Data Item*	*File Section*	*Value*	*Format*
`:Dt:`	Delta information	Delta Table	See below*	S
`:DL:`	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
`:Li:`	Lines inserted by Delta	"	nnnnn	S
`:Ld:`	Lines deleted by Delta	"	nnnnn	S
`:Lu:`	Lines unchanged by Delta	"	nnnnn	S
`:DT:`	Delta type	"	*D* or *R*	S
`:I:`	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
`:R:`	Release number	"	nnnn	S
`:L:`	Level number	"	nnnn	S
`:B:`	Branch number	"	nnnn	S
`:S:`	Sequence number	"	nnnn	S
`:D:`	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
`:Dy:`	Year Delta created	"	nn	S
`:Dm:`	Month Delta created	"	nn	S
`:Dd:`	Day Delta created	"	nn	S
`:T:`	Time Delta created	"	:Th:::Tm:::Ts:	S
`:Th:`	Hour Delta created	"	nn	S
`:Tm:`	Minutes Delta created	"	nn	S
`:Ts:`	Seconds Delta created	"	nn	S
`:P:`	Programmer who created Delta	"	logname	S
`:DS:`	Delta sequence number	"	nnnn	S
`:DP:`	Predecessor Delta seq-no.	"	nnnn	S
`:DI:`	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
`:Dn:`	Deltas included (seq #)	"	:DS: :DS: ...	S
`:Dx:`	Deltas excluded (seq #)	"	:DS: :DS: ...	S
`:Dg:`	Deltas ignored (seq #)	"	:DS: :DS: ...	S
`:MR:`	MR numbers for delta	"	text	M
`:C:`	Comments for delta	"	text	M

Table A-3 *SCCS Files Data Keywords— Continued*

Keyword	*Data Item*	*File Section*	*Value*	*Format*
:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	*yes* or *no*	S
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	*yes* or *no*	S
:BF:	Branch flag	"	*yes* or *no*	S
:J:	Joint edit flag	"	*yes* or *no*	S
:LK:	Locked releases	"	:R:...	S
:Q:	User defined keyword	"	text	S
:M:	Module name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	*yes* or *no*	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of *what*(1) string	N/A	:Z::M:\t:I:	S
:A:	A form of *what*(1) string	N/A	:Z::Y: :M: :I::Z:	S
:Z:	*what*(1) string delimiter	N/A	@(#)	S
:F:	SCCS file name	N/A	text	S
:PN:	SCCS file path name	N/A	text	S

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

Examples of Using `prs`

```
hermes% prs -d"Users and/or user IDs for :F: are:\n:UN:" s.file
```

may produce on the standard output:

```
Users and/or user IDs for s.file are:
xyz
131
abc
```

```
hermes% prs -d"Newest delta for pgm :M:: :I: Created :D: By :P:" -r s.file
```

may produce on the standard output:

```
Newest delta for pgm main.c: 3.7 Created 77/12/1 By cas
```

As a *special case:*

```
hermes% prs s.file
```

may produce on the standard output:

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
bl78-12345
bl79-54321
COMMENTS:
this is the comment line for s.file initial delta
```

for each delta table entry of the ''D'' type. The only option argument allowed to be used with the *special case* is the `-a` option.

Files Used

/tmp/pr?????

A.12. `rmdel` — Remove Delta from SCCS File

`rmdel` removes the delta specified by the *SID* from each named SCCS *file*. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the SID specified must *not* be that of a version being edited for the purpose of making a delta (that is, if a `p.` file (see `get`) exists for the named SCCS file, the SID specified must *not* appear in any entry of the `p.` file).

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

If a directory is named, `rmdel` behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with `s.`) and unreadable files are silently ignored. If a name of `-` is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

The exact permissions necessary to remove a delta are documented earlier in this manual under `sccs` User's—SCCS Simply stated, they are either 1) if you make a delta you can remove it; or 2) if you own the file and directory you can remove a delta.

The delta to be removed must be a 'leaf' delta; that is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Figure A-3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed, deltas 1.3.2.1 and 2.1 can be removed, and so on.

To remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either have created the delta being removed, or be the owner of the SCCS file and its directory.

You must specify the *complete* SID of the delta to be removed, preceded by `-r`. The SID must have two components for a trunk delta, and four components for a branch delta. Thus:

```
hermes% rmdel -r2.3 s.abc
```

removes (trunk) delta '2.3' of the SCCS file.

Before removing the delta, `rmdel` checks the following things:

1. the *release* number (R) of the given SID satisfies the relation:

   ```
   floor ≤ R ≤ ceiling
   ```

2. the SID specified is *not* that of a version for which a `get` for editing has been executed and whose associated `delta` has not yet been made.
3. the login name or group ID of the user either appears in the file's *user list* or the *user list* is empty.
4. the release specified cannot be *locked* against editing (that is, if the `l` flag is set (see `admin`), the release specified *must* not be contained in the list).

If these conditions are satisfied, the delta is removed. Otherwise, processing is terminated.

After the specified delta has been removed, its type indicator in the *delta table* of the SCCS file is changed from 'D' (delta) to 'R' (removed).

Files Used

x-file	(see `delta`)
z-file	(see `delta`)

A.13. `sact` — Display SCCS Editing Activity

`sact` informs the user of any SCCS files which have had one or more `get -e` commands applied to them, that is, there are files out for editing, and deltas are pending. If a directory is named on the command line, `sact` behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

```
sact filename . . .
```

The output for each named file consists of five fields separated by spaces.

Field Number	*Meaning*
1	specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta.
2	specifies the SID for the new delta to be created.
3	contains the logname of the user who will make the delta (that is, executed a `get` for editing).
4	contains the date that `get -e` was executed.
5	contains the time that `get -e` was executed.

A.14. sccsdiff — Display Differences in SCCS Versions

sccsdiff compares two versions of an SCCS file and generates the differences between the two versions. Any number of SCCS files may be specified, but options apply to all files.

```
sccsdiff -rSID1 -rSID2 [ -p ] [ -sn ] filename . . .
```

sccsdiff Options

-r*SID?*
: *SID1* and *SID2* specify the deltas of an SCCS file that are to be compared. Versions are passed to diff in the order given.

-p pipe output for each file through pr.

-s*n*
: *n* is the file segment size that diff(1) will use. This is useful when the system load is high.

Files Used

/tmp/get?????
: Temporary files

Diagnostics from sccsdiff

file : No differences

If the two versions are the same.

A.15. unget — Undo a Previous SCCS get

Unget undoes the effect of a get -e done prior to creating the intended new delta. If a directory is named, unget behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

```
unget [ -rSID ] [ -s ] [ -n ] filename . . .
```

unget Options

Options apply independently to each named file.

Delta to be removed

-r *SID*
: Uniquely identifies which delta is no longer intended. (This would have been specified by get as the ''new delta''). The -r option is necessary only if two or more outstanding *get*s for editing on the same SCCS file were done by the same person (login name). A diagnostic results if the specified *SID* is ambiguous, or if it is necessary but omitted from the command line.

Suppress delta ID

-s Suppress displaying the intended delta's *SID*.

Retain gotten file

-n Retain the gotten file — it is normally removed from the current directory.

A.16. `val` — Validate SCCS File

`val` determines if the specified *file* is an SCCS file meeting the characteristics specified by the optional argument list. Arguments to `val` may appear in any order.

```
val -

                        or
val [ -s ] [ -rSID ] [ -mname ] [ -ytype ] filename . . .
```

`val` has a special argument, `-`, which means read the standard input until an end-of-file condition is detected. Each line read is independently processed as if it were a command line argument list.

`val` generates diagnostic messages on the standard output for each command line and file processed and also returns a single 8-bit code upon exit as described below.

`val` Options

Options apply independently to each named file on the command line.

Suppress error messages

`-s` Silence diagnostic messages normally generated for errors detected while processing the specified files.

Delta number

`-r` *SID*
The argument value ID (*S*CCS *ID*entification String) is an SCCS delta number. A check is made to determine if the SID is ambiguous (for instance, `-r`1 is ambiguous because it physically does not exist but implies 1.1, 1.2, etc. which may exist) or invalid (for instance, `-r`1.0 or `-r`1.1.0 are invalid because neither case can exist as a valid delta number). If the SID is valid and not ambiguous, a check is made to determine if it actually exists.

Compare module names

`-m` *name*
name is compared with the SCCS `%M%` keyword in *file*.

Compare module types

`-y` *type*
type is compared with the SCCS `%Y%` keyword in *file*.

The 8-bit code returned by `val` is a disjunction of the possible errors, that is, can be interpreted as a bit string where (moving from left to right) set bits are interpreted as follows:

Table A-4 *Codes Returned from val Command*

Bit	*Meaning*
0	missing file argument
1	unknown or duplicate option
2	corrupted SCCS file
3	can't open file or file not SCCS
4	SID is invalid or ambiguous
5	SID does not exist
6	%Y%, -y mismatch
7	%M%, -m mismatch

Note that `val` can process two or more files on a given command line and in turn can process multiple command lines (when reading the standard input). In these cases an aggregate code is returned — logical OR of the codes generated for each command line and file processed.

Limitations of the `val` Command

`val` can process up to 50 files on a single command line. Any number above 50 produces a memory dump.

`what` — Identify SCCS Files

`what` finds SCCS identifying information within *any* specified file. `what` does not use any options, nor does it treat directory names and a name of '–' (a lone minus sign) in any special way, as do other SCCS commands.

`what` searches the given file(s) for all occurrences of the string `@(#)`, which is the replacement for the `%Z%` ID keyword (see `get`). `what` then displays whatever follows that string until the first double quote `),` `(` greater than (>), backslash (\), newline, or (non-printing) NUL character.

As an example, let's begin with the SCCS file `s.prog.c` (a C program), which contains the following line:

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

We then do the following `get`:

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

and finally compile the resulting g-file to produce `prog.o` and `a.out`. Using `what` as follows then displays:

```
hermes% what  prog.c  prog.o  a.out
prog.c:
        prog.c:3.4
prog.o:
        prog.c:3.4
a.out:
        prog.c:3.4
hermes%
```

The string `what` searches for need not be inserted via an ID keyword of `get` — it may be inserted in any convenient manner.

A.17. SCCS Files

This section discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

Protection

SCCS relies on the capabilities of the operating system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (that is, changes made by non-SCCS commands). The only protection features provided directly by SCCS are the *release lock* flag, the *release floor* and *ceiling* flags, and the *user list*.

New SCCS files created by `admin` are given mode 444 (read-only). It is best *not* to change this mode, as it prevents any direct modification of the files by non-SCCS commands.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files. The contents of directories should correspond to convenient logical groupings, for example, subsystems of a large project.

SCCS files must have only *one* link (name). Commands that modify SCCS files do so by creating a temporary copy of the file (called the *x-file*), and, upon completion of processing, remove the old file and rename the *x-file*. If the old file has more than one link, removing it and renaming the *x-file* would break the link. Rather than process such files, SCCS commands produce an error message. All SCCS files *must* have names that begin with 's.'.

When only one user uses SCCS, the real and effective user IDs are the same, and that user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

However, in those situations in which several users with different user IDs are assigned responsibility for one SCCS file (for example, in large software development projects), one user (equivalently, one user ID) must be chosen as the 'owner' of the SCCS files and as the one who will 'administer' them (for example, by using `admin`).
This user is termed the *SCCS administrator* for that project. Because other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the `get`, `delta`, and, if desired, `rmdel` and `cdc` commands.

The interface program must be owned by the SCCS administrator, and must have the *set-user- ID on execution* bit on (see `chmod(1)`), so that the effective user ID is the administrator's user ID. This program's function is to invoke the desired SCCS command and to cause it to *inherit* the privileges of the interface program for the duration of that command's execution. In this manner, the owner of an SCCS file can modify it at will. Other users whose *login* names or *group* IDs are

in the *user list* for that file (but who are *not* its owners) are given the necessary permissions only for the duration of the execution of the interface program, and are thus able to modify the SCCS files only through the use of `delta` and, possibly, `rmdel` and `cdc`.
The project-dependent interface program, as its name implies, must be custom-built for each project.

Layout of an SCCS File

SCCS files are composed of lines of ASCII text arranged in six parts, as follows:

Checksum A line containing the 'logical' sum of all the characters of the file (*not* including this checksum itself).

Delta Table Information about each delta, such as its type, SID, date and time of creation, and commentary included.

User Names List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.

Flags Indicators that control certain actions of various SCCS commands.

Descriptive Text Text provided by the user; usually a summary of the contents and purpose of the file.

Body Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.

Detailed information about the contents of the various sections of the file may be found in `sccsfile`(5). In the following, the *checksum* is the only portion of the file discussed.

Because SCCS files are ASCII files, they may be processed by various commands: editors such as `vi`(1), text processing programs such as `grep`(1), `awk`(1), and `cat`(1), and so on. This is quite useful when an SCCS file must be modified manually (for example, when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly), or when one wants to simply 'look' at the file.

CAUTION **Extreme care should be exercised when modifying SCCS files with non-SCCS commands.**

Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, all or part of an SCCS file is destroyed. SCCS commands (like most commands) display an error message when a file does not exist. In addition, SCCS commands use the *checksum* stored in the SCCS file to determine whether a file has been *corrupted* since it was last accessed (has lost data, or has been changed). The *only* SCCS command which will process a corrupted SCCS file is `admin` with the `-h` or `-z` options. This is discussed below.

SCCS files should be audited (checked) for possible corruptions on a regular basis. The simplest and fastest way to audit such files is to use `admin` with the `-h` option on them:

```
hermes% admin  -h  s.file1  s.file2  ...
        or
hermes% admin  -h  directory1  directory2  ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message

```
corrupted file (co6)
```

is produced for that file. This process continues until all files have been examined. When examining directories (as in the second example above), the process just described does not detect *missing* files. A simple way to detect whether any files are missing from a directory is to periodically list the contents of the directory (using `ls(1)`), and compare the current listing with the previous one. Any file which appears on the previous list but not the current one has been removed by some means.

When a file has been corrupted, the appropriate method of restoration depends upon the extent of the corruption. If damage is extensive, the best solution is to restore the file from a backup copy. When damage is minor, repairing the file with your favorite text editor may be possible. If you do repair the file with the system's text processing capabilities, you must use `admin` with the `-z` option to recompute the checksum to bring it into agreement with the actual contents of the file:

```
hermes% admin  -z  s.file
```

After this command is executed on a file, any corruption which may have existed in that file will no longer be detectable.

B

make Enhancements Summary

B

make Enhancements Summary

B.1. New Features

Default Makefile

make's implicit rules and macro definitions are no longer hard-coded within the program itself. They are now contained in the default makefile /usr/include/make/default.mk. make reads this file automatically, unless there is a file in the local directory named default.mk. When you use a local default.mk file, you must add an include /usr/include/make/default.mk directive to get the standard implicit rules and predefined macros.

The State File .make.state

make also reads from a state file, .make.state in the directory. When the special-function target .KEEP_STATE is used in the makefile, make writes out a cumulative report for each target containing a list of hidden dependencies (as reported by compilation processors such as cpp), and the most recent rule used to build each target. The state file is very similar in format to an ordinary makefile.

Hidden Dependency Checking

When activated by the presence of the .KEEP_STATE target, make uses information reported from cpp, f77, make, pc and other compilation commands, and performs a dependency check against any header files (or in some cases, libraries) that are incorporated into the target file. These "hidden" dependency files do not appear in the dependency list, and often do not reside in the local directory.

Command Dependency Checking

When .KEEP_STATE is in effect, if any command line used to build a target changes between make runs (either as a result of editing the makefile or because of a different macro expansion), the target is treated as if it were out of date; make rebuilds it (even if it is newer that the files it depends on).

Automatic sccs Extraction

Tilde Rules Superceded

This version of make automatically runs sccs get, as appropriate, when there is no rule to build a target file. A tilde appended to a suffix in the suffixes list indicates that sccs extraction is appropriate for files having that suffix. There are no longer special versions of implicit rules that include commands to extract current versions of sccs files.

To inhibit or alter the procedure for automatic extraction of the current sccs version, redefine the .SCCS_GET special-function target. An empty rule for this target inhibits automatic extraction entirely.

sccs History Files

make no longer searches the current directory for sccs history (s.) files. These files must now reside in an SCCS subdirectory.

Pattern Matching Rules: More Convenient than Suffix Rules

Pattern matching rules have been added to simplify the process of adding new implicit rules of your own design. A target entry of the form:

tp%*ts* : *dp*%*ds*
rule

defines a pattern matching rule for building a target from a a related dependency file. *tp* is the target's prefix; *ts*, its suffix. *dp* is the dependency's prefix; *ds*, its suffix. The % symbol is a wild card that matches a contiguous string of zero or more characters appearing in both the target and the dependency filename. For example, the following target entry defines a pattern matching rule for building a troff output file, ending in .tr from a file that uses the -ms macro package ending in .ms:

```
%.tr: %.ms
        troff -t -ms $< > $@
```

With this entry in the makefile, the command:

```
make doc.tr
```

produces:

```
hermes% make doc.tr
troff -t -ms doc.ms > doc.tr
```

Using that same entry, if there is a file named doc2.ms the command:

```
make doc2.tr
```

produces:

```
hermes% make doc2.tr
troff -t -ms doc2.ms > doc2.tr
```

An explicit target entry overrides any pattern matching rule that might apply to a target. Pattern matching rules, in turn, normally override implicit rules. An exception to this is when the pattern matching rule has no commands in the rule portion of its target entry. In this case, make continues the search for a rule to build the target, and using as its dependency the file that matched the (dependency) pattern.

Pattern Replacement Macro References

As with suffix rules and pattern matching rules, pattern replacement macro references has been added to provide a more general method for altering the values of words in a specific macro reference than that already provided by suffix replacement in macro references. A pattern replacement macro reference takes the form:

$ (*macro* :*p* %*s* =*np* %*ns*)

where *p* is an existing prefix (if any), *s* is an existing suffix (if any), *np* and *ns* are the new prefix and suffix, respectively, and % is a wild card character matching a string of zero or more characters within a word. The prefix and suffix replacements are applied to all words in the macro value that match the existing pattern. This feature is useful for prefixing the name of a subdirectory to each item in a list of files. For instance, the following makefile:

```
SOURCES= x.c y.c z.c
SUBFILES.o= $(SOURCES:%.c=subdir/%.o)
all:
        echo $(SUBFILES.o)
```

produces:

```
hermes% make
echo subdir/x.o subdir/y.o subdir/z.o
subdir/x.o subdir/y.o subdir/z.o
```

Please note that pattern replacement macro references should not appear on the dependency line of a pattern matching rule's target entry. This produces unexpected results. With the makefile:

```
OBJECT= .o
x:
%: %.$(OBJECT:%o=%Z)
        cp $< $@
```

it looks as if make should attempt to build a target named, x from a file named x.Z. However, the pattern matching rule is not recognized; make cannot determine which of the % characters in the dependency line apply to the pattern matching rule, and which apply to the macro reference. Consequently, the target entry for x.Z is never reached. To avoid problems like this, you can use an intermediate macro on another line:

```
OBJECT= .o
ZMAC= $(OBJECT:%o=%Z)
%: %.$(ZMAC)

x:
%: %$(ZMAC)
        cp $< $@
```

New Options

There are a number of new options:

–d Display dependency-check results for each target processed. Displays all dependencies that are newer, or indicates that the target was built as the result of a command dependency.

–dd The same function as –d had in earlier versions of make. Displays a great deal of output about all details of the make run, including internal states, etc.

–D Display the text of the makefile.

–DD Display the text of the makefile, and of the default makefile being used.

–p Print macro definitions and target entries.

–P Report on dependency checks without rebuilding targets.

Support for Modula-2

This version of make contains predefined macros and implicit rules for compiling Modula-2 sources.

Naming Scheme for Predefined Macros

The naming scheme for predefined macros has been rationalized, and the implicit rules have been rewritten to reflect the new scheme. The macros and implicit rules are upward compatible with existing makefiles.

For example, there is now a macro called SUFFIXES, that contains the default entries for the suffixes list; the target entry for the default suffixes list looks like:

```
.SUFFIXES: $(SUFFIXES)
```

If you want to insert new suffixes at the head of the list, you can do so quite simply as follows:

```
.SUFFIXES:
.SUFFIXES: .ms .tr $(SUFFIXES)
```

Other examples include the macros for standard compilations commands:

LINK.c Standard cc command line for producing executable files.
COMPILE.c Standard cc command line for producing object files.

New Special-Purpose Targets

The .KEEP_STATE target should not be removed once it has been used in a make run.

.KEEP_STATE When included in a makefile, this target enables hidden dependency and command dependency checking. In addition, make updates the state file .make.state after each run.

.INIT and .DONE

These targets can be used to supply commands to perform at the beginning and end, respectively, of each make run.

.SCCS_GET This target contains the rule for extracting current versions from sccs history files.

New Implicit Rule for lint

Implicit rules have been added to support incremental verification with lint.

Macro Processing Changes

A macro's value can now be of virtually any length.

Macros: Definition, Substitution, and Suffix Replacement

New Append Operator: +=

This operator appends a SPACE, followed by a word or words, onto the existing value of the macro.

Conditional Macro Definitions: :=

This operator indicates a conditional (targetwise) macro definition. A makefile entry of the form:

target := *macro* = *value*

indicates that *macro* takes the indicated *value* while processing *target* and its dependencies.

Suffix Replacement Precedence

Substring replacement now takes place following expansion of the macro being referred to. Previous versions of make applied the substitution first, with results that were counterintuitive.

Nested Macro References

make now expands inner references before parsing the outer reference. So, a nested reference as in this example:

```
CFLAGS-g = -I../include
OPTION = -g
$(CFLAGS$(OPTION))
```

now yields the value -I../include, rather than a null value, as it would have in previous versions.

Cross-Compilation Macros

The predefined macros HOST_ARCH HOST_MACH TARGET_ARCH and TARGET_MACH are available for use in cross-compilations. By default, the *arch* macros are set to the value returned by the arch command; the *mach* macros are set to the value returned by mach.

Improved ar Library Support

Lists of Members

make automatically updates an *ar* library member from a file having the same name as the member. Also, make now supports lists of members as dependency names of the form:

> *lib.a*: *lib.a* (*member member* . . .)

Handling of ar's Name Length Limitation

make now copes with the 15-character member-name length limitation in ar. It now recognizes a member name that matches the first 15 characters of a filename as the member corresponding to the file.

Target Groups

It is now possible to specify that a rule produces a set of target files. A + sign between target names in the target entry indicates that the named targets comprise a group. The target group's rule is performed once, at most, in a make invocation.

Clearing Definitions of Special Targets and Implicit Rules

To clear the dependency list and rule for a special target, implicit rule, or any target with a name beginning with '.', add a target entry to the makefile with no dependency list and no rule. For example, to clear a previous .DEFAULT rule, add the line:

```
    .DEFAULT:
```

to your makefile.

B.2. Incompatibilities with Previous Versions of make

New Meaning for -d Option

The -d option now reports the reason why a target is considered out of date.

Dynamic Macros

Although the dynamic macros < and * were documented being assigned only for implicit rules and the .DEFAULT target, in some cases they actually were assigned for explicit target entries. The assignment action is now documented properly.

The actual value assigned to each of these macros is derived by the same procedure used within implicit rules (this hasn't changed). This can lead to unexpected results when they are used in explicit target entries.

Even if you supply explicit dependencies, make doesn't use them to derive values for these macros. Instead, it searches for an appropriate implicit rule and dependency file. For instance, if you have the explicit target entry:

```
test: test.f
        echo $<
```

and the files: test.c and test.f, you might expect that $< would be assigned the value test.f. This is *not* the case. It is assigned test.c, because .c is ahead of .f in the suffixes list:

```
hermes% make test
echo test.c
test.c
```

For explicit entries, we recommend a strictly deterministic method for deriving a dependency name using macro references and suffix replacements. For example, you could use: $@.f instead of $< to derive the dependency name. To derive the basename of a .o target file, you could use the suffix replacement macro reference: $(@:.o=) instead of $*.

When hidden dependency checking is in effect, the $? dynamic macro's value includes the names of hidden dependencies, such as header files. This can lead to failed compilations when using a target entry such as:

```
x: x.c
        $(CC) -o $@ $?
```

and the file x.c #include's header files. The workaround is to replace $? with $<.

Index

Notes