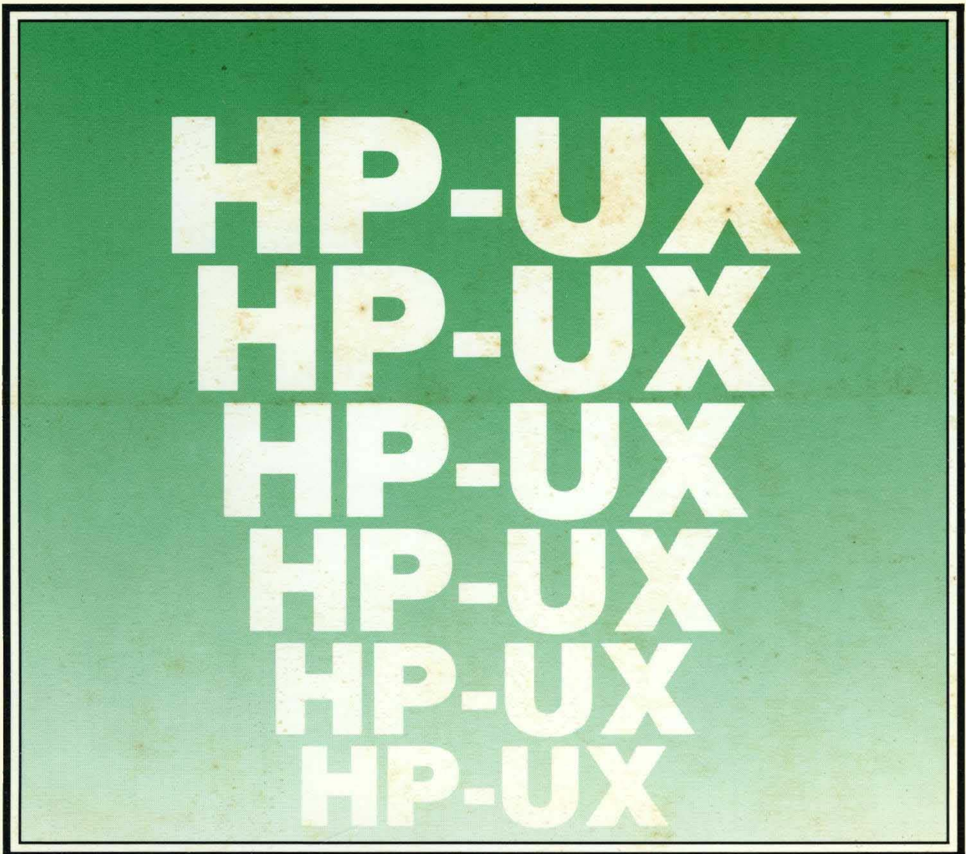
HEWLETT PACKARD

HP-UX Concepts and Tutorials

Vol. 2: Programming Environment



HP-UX Concepts and Tutorials
Vol. 2: Programming Environment

Manual Reorder No. 97089-90030

© Copyright 1985 Hewlett-Packard Company

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

Use of this manual and flexible disc(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs can be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the Rights in Technical Data and Software clause in DAR 7-104.9(a).

© Copyright 1980, Bell Telephone Laboratories, Inc.

Hewlett-Packard Company

3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

July 1984...First Edition – Part numbered 97089-90004 was 4 volumes and was shipped with HP-UX 4.0 on Series 500 Computers and with HP-UX 2.1, 2.2, 2.3, and 2.4 on Series 200 Computers. Each volume did not have an individual part number. This was obsoleted in April, 1985 and replaced with Manual Kit #97070-87903 which includes:

	Title	Manual P/N	Binder P/N
Vol. 1:	Text Processing and Formatting	97089-90020	9282-1023
Vol. 2:	Programming Environment	97089-90030	9282-1023
Vol. 3:	Software Development Tools	97089-90040	9282-1023
Vol. 4:	Shells and Miscellaneous Tools	97089-90050	9282-1023
Vol. 5:	Data Communications	97089-90060	9282-1023
Vol. 6:	Graphics	97089-90070	9282-1023

April 1985...Edition 1 – Volume 2: Programming Environment

Contents

The articles contained in *HP-UX Concepts and Tutorials* are provided to help you use the commands and utilities provided with HP-UX. The articles have several sources. Some were written at Hewlett-Packard specifically for HP computers. Others were written at Bell Laboratories or University of California at Berkeley and have been tailored for HP computers.

HP-UX Concepts and Tutorials has six volumes:

- Volume 1: Text Processing and Formatting
- Volume 2: Programming Environment
- Volume 3: Software Development Tools
- Volume 4: Shells and Miscellaneous Tools
- Volume 5: Data Communications
- Volume 6: Graphics

This is "Vol. 2: Programming Environment" and the articles it includes are:

1. HP-UX Programming
2. Using C on the HP 9000 Series 500 Computer
3. Using the C Library Routines
4. Lint: C Program Checker
5. MC68000 Assembler on HP-UX
6. Ratfor: A Preprocessor for a Rational FORTRAN
7. Native Language Support
8. Using curses and terminfo

Warranty Statement

Hewlett-Packard products are warranted against defects in materials and workmanship. For Hewlett-Packard computer system products sold in the U.S.A. and Canada, this warranty applies for ninety (90) days from the date of shipment.* Hewlett-Packard will, at its option, repair or replace equipment which proves to be defective during the warranty period. This warranty includes labor, parts, and surface travel costs, if any. Equipment returned to Hewlett-Packard for repair must be shipped freight prepaid. Repairs neccessitated by misuse of the equipment, or by hardware, software, or interfacing not provided by Hewlett-Packard are not covered by this warranty.

HP warrants that its software and firmware designated by HP for use with a CPU will execute its programming instructions when properly installed on that CPU. HP does not warrant that the operation of the CPU, software, or firmware will be uninterrupted or error free.

NO OTHER WARRANTY IS EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HEWLETT-PACKARED SHALL NOT BE LIABLE FOR CONSEQUENTIAL DAMAGES.

HP 9000 Series 200

For the HP 9000 Series 200 family, the following special requirements apply. The Model 216 computer comes with a 90-day, Return-to-HP warranty during which time HP will repair your Model 216, however, the computer must be shipped to an HP Repair Center.

All other Series 200 computers come with a 90-Day On-Site warranty during which time HP will travel to your site and repair any defects. The following minimum configuration of equipment is necessary to run the appropriate HP diagnostic programs: 1) .5 Mbyte RAM; 2) HP-compatible 3.5″ or 5.25″ disc drive for loading system functional tests, or a system install device for HP-UX installations; 3) system console consisting of a keyboard and video display to allow interaction with the CPU and to report the results of the diagnostics.

To order or to obtain additional information on HP support services and service contracts, call the HP Support Services Telemarketing Center at (800) 835-4747 or your local HP Sales and Support office.

*For other countries, contact your local Sales and Support Office to determine warranty terms.

Table of Contents

HP-UX Programming

Introduction 1
Basics 2
  Program Arguments 2
  The "Standard Input" and "Standard Output" 2
The Standard I/O Library 4
  File Access 4
  Error Handling – *Stderr* and *Exit* 6
  Miscellaneous I/O Functions 7
Low-level I/O 8
  File Descriptors 8
  Read and Write 9
  Open, Creat, Close, Unlink 10
  Random Access – *Lseek* 12
  Error Processing 12
Processes 13
  The "System" Function 13
  Low-level Process Creation – *Execl* and *Execv* 13
  Control of Processes – *Fork* and *Wait* 14
  Pipes 15
  Signals – Interrupts and All That 18
Appendix – The Standard I/O Library 22
  General Usage 22
  Calls 23

HP-UX Programming

Introduction

This tutorial describes how to write programs that interface with the HP-UX operating system in a non-trivial 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.

The document collects material which is scattered throughout several sections of the *HP-UX 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*. Some of the material in this tutorial is based on topics covered more carefully there. You should also be familiar with HP-UX itself.

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 individual 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 them all.

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.

The "Standard Input" and "Standard Output"

The simplest input mechanism is to read 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 can be substituted for the terminal by using the < convention: if *prog* uses *getchar*, then the command line

```
prog <file
```

causes *prog* to read *file* 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 HP-UX pipe mechanism:

```
otherprog | prog
```

provides the standard input for *prog* from the standard output 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*,

```
prog >outfile
```

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

```
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 HP-UX 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 new-line 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. It causes the C compiler to read 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:

```
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 6 discusses status returns in more detail.

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.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. 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

```
wc x.c y.c
```

prints 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 file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be **opened** by the standard library function *fopen*. *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 is

```
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 only allowable modes are read (`rA) write (w`) or append (`a`)

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. 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* will return the null pointer value `NULL` (which is 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*. *Getc* and *putc* return EOF on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files 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 pre-defined 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 stand-alone 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* except 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 can have open simultaneously, it's a good idea to release resources when they are no longer needed. There is also 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. *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 program actually signals errors in another way, using the function *exit* to terminate program execution. The argument of *exit* is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero 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* causes immediate termination 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 previously illustrated.

Normally output with *putc*, etc., is buffered (except to *stderr*); to force it out immediately, use *fflush(fp)*.

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*.

fgets(buf, size, fp) copies the next line from *fp*, up to and including a new-line, 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 onto the input stream *fp*; a subsequent call to *getc*, *fscanf*, etc., will encounter *c*. Only one character of push-back per file is permitted.

Low-level I/O

This section describes the bottom level of I/O on the HP-UX system. The lowest level of I/O in HP-UX 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 HP-UX 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?), and 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 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 the standard input, the standard output, and the 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 worrying about opening the files.

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

```
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

```
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 new-line, 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 512, which corresponds to a physical block size 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 512         /* best size for HP-UX */

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 to be written by *write*; 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 Series 200/500 computers, but not necessarily for other computers and systems.)

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 512
getchar()                   /* buffered version */
{
    static char        buf[BUFSIZE];
    static char        *bufp = buf;
    static int         n = 0;

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

Open, Creat, Close, 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* [sic].

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 re-write old ones.

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

returns a file descriptor if it was able to 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 HP-UX 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 HP-UX 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 512
#define PMODE 0644        /* RW for owner, R for group, others */

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

    if (argc != 3)
        error("Usage: cp from to", NULL);
    if ((f1 = open(argv[1], 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 15-25) 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 re-use 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 — 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 cell *errno*. The meanings of the various error numbers are listed in the entry for *errno(2)* in the *HP-UX Reference*. 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 print out the reason for failure. The routine *perror* will print 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 printed by your program.

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 new-line at the end) and executes it. For instance, to time-stamp 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 the appendix.

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.

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

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

The first argument to *execl* is the **file name** 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 place-holder. 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 make the second pass simply 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, say

```
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 non-zero; 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 non-zero 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 non-zero 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 pointing at the right files, 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 reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
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 will 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 will wait until data arrives; if a process writes into a pipe which is too full, it will wait 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 the pipe with a *pipe* system call; it then *forks* 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 *closes* 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. (Yes, this is a bit tricky, but it's a standard idiom.) 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 from the parent instead of reading. You may find it a useful exercise to step through that case.

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.

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	Sent when the **DEL** character is typed;
Quit	Generated by the **FS** character;
Hangup	Caused by hanging up the phone; 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; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine that alters the default action is called **signal**. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address, and 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);
```

causes interrupts to be 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 to allow the program to 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 printout should not cause it to terminate 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);     /* save 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; it is an array of some sort. The *setjmp* routine then saves the state of things. When an interrupt occurs, a call is forced to the *onintr* routine, which can print a message, set flags, or whatever. *longjmp* takes as argument an object stored into by *setjmp*, and restores control to the location after 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 on occurrence of a signal sets a flag and then returns instead of calling *exit* or *longjmp*, execution will continue 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* program 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 signal-catching is combined with execution of 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 *system*:

```
#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 Series 200/500 computers; 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
```

Appendix — The Standard I/O Library

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

- 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.
- 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.
- 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 other than the one upon which the program was written.

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 file
stdout	The name of the standard output file
stderr	The name of the standard error file
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. 512) 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 cannot 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, in effect, constants and cannot be assigned to.

Calls

`FILE *fopen(<filename>, <type>) char *<filename>, *<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. The value returned is a file pointer. If it is NULL, the attempt to open failed.

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

closes the stream named by *ioptr*, if necessary, then reopens it as if by *fopen*. If the attempt to open fails, NULL is returned. Otherwise *ioptr*, now refers to the new file. Often the reopened stream is *stdin* or *stdout*.

`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 is a legal character.

`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(c, ioptr) FILE *ioptr;`

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, but EOF is returned on error.

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

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

`fclose(ioptr) FILE *ioptr;`

closes the file corresponding to *ioptr* after any buffers are emptied. Any buffering allocated by the I/O system is freed. *fclose* is automatic on normal termination of the program.

`fflush(ioptr) FILE *ioptr;`

writes out any buffered information on the (output) stream named by *ioptr*. Output files are normally buffered if and only if they are not directed to the terminal; however, *stderr* always starts off unbuffered and remains so unless *setbuf* is used, or unless it is reopened.

`exit(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(ioptr) FILE *ioptr;
```

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

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

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

```
getchar( );
```

is identical to `getc(stdin)`.

```
putchar(c);
```

is identical to `putc(c, stdout)`.

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

reads up to $n-1$ characters from the stream *ioptr* into the character pointer *s*. The read terminates with a new-line character. The new-line character is placed in the buffer followed by a null character. *Fgets* returns the first argument, or NULL if error or end-of-file occurred.

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

writes the null-terminated string (character array) *s* on the stream *ioptr*. No new-line is appended. No value is returned.

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

pushes the argument character *c* back on the input stream named by *ioptr*. Only one character can be pushed back.

```
printf(format, a1, . . . ) char *format;
fprintf(ioptr, format, a1, . . . ) FILE *ioptr; char *format;
sprintf(s, format, a1, . . . )char *s, *format;
```

printf writes on the standard output. *fprintf* writes on the named output stream. *sprintf* puts characters in the character array (string) named by *s*. The specifications are as described in section *printf* (3) of the *HP-UX Reference*.

```
scanf(format, a1, . . . ) char *format;
fscanf(ioptr,\ format,\ a1, . . . ) FILE *ioptr; char *format;
sscanf(s, format, a1, . . .  ) char *s, *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 a 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(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
```

reads *nitems* of data beginning at *ptr* from file *ioptr*. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the *fopen* call.

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

like *fread*, but in the other direction.

```
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(string) char *string;
```

string is executed by the shell as if typed at the terminal.

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

returns the next 32-bit 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.

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

writes the integer *w* on the named output stream.

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

setbuf can be used after a stream has been opened but before I/O has started. If *buf* is NULL, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size: `char buf[BUFSIZ];`

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

returns the integer file descriptor associated with the file.

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

adjusts the location of the next byte in the stream named by *ioptr*. *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 HP-UX systems, the offset must be a value returned from *ftell* and the ptrname must be 0).

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

returns the byte offset (measured from the beginning of the file) associated with the named stream. Any buffering is properly accounted for. (On HP-UX 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(uid, buf) char *buf;
```

searches the password file 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.

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

allocates *num* bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. NULL is returned if no space is available.

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

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

```
cfree(ptr) char *ptr;
```

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>*.

`isalpha(c)`	returns non-zero if the argument is alphabetic.
`isupper(c)`	returns non-zero if the argument is upper-case alphabetic.
`islower(c)`	returns non-zero if the argument is lower-case alphabetic.
`isdigit(c)`	returns non-zero if the argument is a digit.
`isspace(c)`	returns non-zero if the argument is a spacing character: tab,
`ispunct(c)`	returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.
`isalnum(c)`	returns non-zero if the argument is a letter or a digit.
`isprint(c)`	returns non-zero if the argument is printable-a letter, digit, or punctuation character.
`iscntrl(c)`	returns non-zero if the argument is a control character.
`isascii(c)`	returns non-zero if the argument is an ASCII character, i.e., less than octal 0200.
`toupper(c)`	returns the uppercase character corresponding to the lowercase letter *c*.
`tolower(c)`	returns the lowercase character corresponding to the uppercase letter.

Table of Contents

Using C on HP9000 Series 500 Computers

Introduction ... 1
Data Types and Manipulations ... 1
  Data Type Sizes ... 1
  Char Data Type ... 1
  Register Data Type ... 1
  Integer Overflow ... 1
  Division by Zero ... 2
  Identifiers ... 2
  Shift Operators ... 2
  Bit Fields ... 2
Code/Data Limitations ... 2
Portability Considerations ... 3

Using C on HP 9000 Series 500 Computers

Introduction

The purpose of this article is to describe the machine dependent features of the C programming language as it is implemented on the HP 9000 Series 500 computers. No attempt is made here to fully describe C. When applicable, page numbers are given that reference pages in the Kernighan and Ritchie text, *The C Programming Language*, which are related to the discussion.

Data Types and Manipulations

Data Type Sizes

The following table gives the sizes and alignment requirements of the six data types implemented in C (page 34):

Type	Size	Alignment Requirements
char	8 bits	byte boundary
short	16 bits	half word
int	32 bits	full word
long	32 bits	full word
float	32 bits	full word
double	64 bits	full word

Char Data Type

The **char** data type is treated as signed by default. This implies that, if a **char** is assigned to an **int**, sign extension will take place (page 40).

Register Data Type

Because the Series 500 computers are stack machines, declaring a variable to be **register** is ignored, and is treated as a no-op (page 81).

Integer Overflow

Integer overflow does *not* generate an error by default (page 185).

Division by Zero

Whenever division by zero occurs, you get the (somewhat misleading) error message "Floating exception" at run-time.

Identifiers

Internal identifiers have 16 significant characters. External identifiers have 15 significant characters (page 179).

Shift Operators

An arithmetic shift is performed if the left operand is signed. If the left operand is unsigned, a logical shift is performed (page 45). (Remember that integer constants are treated as signed unless cast to unsigned.)

Bit Fields

Bit fields are assigned left to right, and are treated as unsigned (page 138).

Code/Data Limitations

The following limitations exist on the Series 500 computers:

a maximum of 2^19 bytes of local variables in any procedure;

a maximum of 2^19 bytes of parameters in any function call;

any branch instruction generated by a procedure must be within 2^18 bytes of its target;

structure functions cannot return a structure bigger than 2^24 bytes.

If you violate any of the above limits, you get the message "impossible reach" from the assembly step of *cc*. Other limitations are:

a maximum of 255 procedures in any single compilation (i.e. any single ".c" file and everything it **#includes**). If you exceed this, you get "proctable overflow" from the assembler;

a maximum of 32 767 lines of assembly code generated by *cc*. If you exceed this, you get "too many lines" from the assembler. To work around this, break your program up into smaller pieces;

a maximum of 2^19 bytes of global scalar data (includes all global scalar variables, all static scalar variables, all global and static structures, and 4 bytes for each global or static array). If you exceed this, you get "byte offset too large" from the linker, *ld*.

When compiling with *cc*, you can recognize assembler errors by the fact that they make reference to a file called */tmp/ctm3x*, where *x* is a single letter. Also, you can use the **–v** option to watch the compilation process, and note where the error occurs.

Portability Considerations

The following list should be kept in mind when transporting C code to the Series 500 computers from other machines:

the Series 500 computers do *not* swap bytes;

dereferencing a null pointer for a read or write operation generates a run-time error. On some other machines, dereferencing a null pointer for a read operation returns zero;

beware of attempts to use absolute addressing. The use of hard-coded addresses is not likely to work on any machine to which you want to port code;

even though the stack grows toward higher memory addresses, parameters are stacked toward decreasing addresses. Thus, if you want to use a pointer to step through a variable length parameter list, you must *decrement* the pointer.

Table of Contents

Using the C Library Routines

Part 1: Standard Input/Output Routines ... 3
Input/Output Using Stdin and Stdout ... 4
Single-character Input/Output ... 4
String Input/Output ... 5
Formatted Input/Output ... 6
Scanf ... 6
Conversion Specifications ... 7
Integer Conversion Characters ... 7
Character Conversion Characters ... 7
Floating-point Conversion Characters ... 8
Literal Characters ... 9
Examples ... 9
Printf ... 12
Literal Characters ... 12
Conversion Specifications ... 12
Conversion Characters ... 13
Examples ... 15
Input/Output from/to Strings ... 17
Reading Data from a String ... 17
Writing Data Into a String ... 19
Input/Output Using Ordinary Files ... 21
Opening Ordinary Files ... 21
Single-character Input/Output ... 23
Character Push-back ... 25
String Input/Output ... 26
Formatted Input/Output ... 28
Binary Input/Output ... 29
Stream Status and Control Routines ... 33
Stream Status Inquiry Routines ... 33
Re-positioning Stream I/O Operations (*rewind*, *ftell*, *fseek*) ... 35
Stream Control Routines ... 39
fclose ... 39
setbuf ... 39
setvbuf ... 40
fflush ... 41
freopen ... 42
Converting Between File Pointers and File Descriptors ... 43
Interprocess Communication ... 45

Part 2: Math Routines 47
Absolute Value Functions. 48
Power, Square Root, and Logarithmic Functions 49
Trigonometric Functions. 50
Miscellaneous Functions. 53
Calculating Upper and Lower Bounds. 53
Calculating Remainders 53
Calculating A Hypotenuse. 55
Generating Random Numbers 55
Floating-point Exponentiation Routines. 56
Part 3: Character Conversion and Classification 57
Converting Between Uppercase and Lowercase 57
Character Classification 57
String Manipulation. 58
Concatenating Strings 58
Copying Strings 58
Comparing Strings 60
Finding the Length of a String. 61
Finding Characters in Strings. 61
Miscellaneous String Routines 63
Finding Characters Common to Two Strings 63
Breaking a String into Tokens. 63
Part 4: Date and Time Manipulation 65

Using the C Library Routines

The purpose of this tutorial is to illustrate the use of the library routines described in Section 3 of the *HP-UX Reference* manual that are most commonly used. Examples are included to demonstrate programming techniques.

This article assumes that you have a working knowledge of the C programming language. No attempt is made here to explain or teach C programming techniques, other than those that are relevant to a particular library routine.

Material is presented in three sections, each dealing with the following topics in the order listed:

- Standard Input/Output Routines,
- Math Routines, including trigonometric and other functions, and
- String Manipulation Routines.

Standard Input/Output Routines

Part 1

There are more library routines in this category than in any other. Described under this heading are routines that perform all kinds of input and output, from single characters to entire strings. Also described are routines that adjust I/O buffering, routines that enable input from or output to files, and routines that enable random access to data. These routines require that the include file **stdio.h** be **#include**d in C programs containing calls to them.

The standard I/O routines are inseparably linked with files. A file must be *opened* before its contents can be used. Three "files" are automatically opened for you by the system. Including **stdio.h** in your program assigns buffering to them. These three "files" are the *standard input*, *standard output*, and *standard error* files. Their names are **stdin**, **stdout**, and **stderr**, respectively.

Actually, it is more accurate to think of these "files" as *pipes* connecting two points. Each pipe accepts data at one end, and transfers the data to its destination at the other end. These pipes have only limited ability to store data. Once a certain number of bytes have been written into the pipe, data must be read from the other end before the pipe can accept more data. Writing data into a pipe is analogous to pumping water into a pipeline. The pipeline is able to hold some water, but if the valve at the receiving end of the pipe is shut, the pipeline is soon unable to hold any more water. Opening the valve is analogous to reading data from the pipe. Once water has been removed from the pipeline, more water can be pumped in at the source.

Once a certain volume of water has been allowed to flow out of a pipeline, that same water no longer exists in the pipeline. This is also true for data that has been received from **stdin**, **stdout**, and **stderr**. Reading data from **stdin**, for instance, removes that data from **stdin**. You can see that **stdin**, **stdout**, and **stderr** are very different from ordinary files. Not only can they store small amounts of data, but that data exists only until it is read (unless it is "pushed back" -- see *Character Push-Back* later in this article).

Stdin is opened for reading. This means that your program can only receive data from **stdin**; it cannot write data into it. By default, **stdin**'s source of data is your terminal's keyboard. Thus, whatever you type at your keyboard provides the data that flows through **stdin** and becomes available to your program at the other end. By default, **stdin** is buffered via a buffer containing exactly **BUFSIZ** bytes, where BUFSIZ is a constant defined in **stdio.h**. For Series 200 and Series 500 computers, BUFSIZ is 1024. Due to terminal driver characteristics, data you type in at your keyboard is not available to a program until you press **RETURN** (or its equivalent).

Stdout is opened for writing, which means that your program is the source of data for **stdout**. Your program cannot, however, read data from **stdout**. By default, the destination of **stdout** is your terminal's screen. Thus, data fed into **stdout** appears on your screen. **Stdout** is typically used for all output that arises from successful execution of a program (status reports, lists of tasks being performed, etc.). Like **stdin**, **stdout** is buffered via a buffer containing BUFSIZ bytes.

Stderr is also opened for writing, allowing your program to feed data into it, but disallowing reading. Just like **stdout**, **stderr**'s destination is your terminal's screen by default. **Stderr** is typically used to output data which arises from an erroneous condition in a program, such as error messages, warnings, etc. **Stderr** is unbuffered by default, which means that data written to **stderr** is transferred to its destination one byte at a time.

The buffering for these pipes, as well as for any open file, can be modified – see the *Stream Status and Control Routines* section later in this tutorial.

Of course, your program would be severely limited in its I/O capabilities if it had only these three pipes to work with. Therefore, ordinary text files can be opened for reading, or created/opened for writing, appending, or both reading and writing. Directories can also be opened, but only for reading. These features are discussed later in this article. For now, the use of **stdin** and **stdout** is described (**stderr** is also left for later discussion).

Input/Output Using Stdin and Stdout

This section describes those routines which are capable of I/O using **stdin** and **stdout** only. The routines discussed are *getchar* and *putchar* (single character I/O), *gets* and *puts* (string I/O), and *scanf* and *printf* (formatted I/O of all types).

Single-character Input/Output

This section describes the two basic input and output routines, *getchar* and *putchar. Getchar* is a macro defined in **stdio.h** which reads one character from **stdin**. Similarly, *putchar* is also a macro defined in **stdio.h**. *Putchar* writes one character on **stdout**.

As an example, consider the following program, which simply reads **stdin** and echos whatever it finds to **stdout**. The program terminates when it receives an at-sign (@) from **stdin**.

```
#include <stdio.h>
main()
{
   int c;

   while((c = getchar()) != '@')
        putchar(c);
   putchar('\n');
}
```

Why is *c* declared an **int** instead of a **char**? For most applications, **char** works fine. In certain cases, however, sign extension, bit shifting, and similar operations cause strange results with **chars**. Therefore, **int** is used here, and in all following examples, to be safe.

The final *putchar* statement in the program is used to output a new-line so that your shell prompt appears at the beginning of a new line, instead of at the end of the last line of output. Type it in and give it a try! Remember that your input is not available to the program until you press **RETURN**.

Getchar and *putchar* are most useful in *filters* — — programs which accept data and modify it in some way before passing it on. Suppose you want to write a program which puts parentheses around each vowel encountered in the input. It's easy to do with these routines:

```
#include <stdio.h>
main()
{
    int c;

    while((c = getchar()) != '\n') {
        if(vowel(c)) {
            putchar('(');
            putchar(c);
            putchar(')');

        }else
             putchar(c);
    }
vowel(c)
char c;
{
    if(c=='a' || c=='A' || c=='e' || c=='E' || c=='i' || c=='I'
    || c=='o' || c=='O' || c=='u' || c=='U')
        return(1);
    else
        return(0);
}
```

The vowel test is placed in the function **vowel**, since it tends to clutter up the main program. This program terminates when it encounters a new-line.

String Input/Output

The *gets* function reads a string from **stdin** and stores it in a character array. The string is terminated by a new-line in the input, which *gets* replaces with a NULL character in the array. Its companion function, *puts*, copies a string from a character array to **stdout**. The string is terminated by a NULL character in the array, which *puts* replaces with a new-line in the output.

The simple "echo" program from the last section can be rewritten using *gets* and *puts*.

```
#include <stdio.h>
main()
{
    char line[80], *gets();

    while((gets(line)) != NULL)
        puts(line);
}
```

This program, as written, runs forever. To terminate it, press **BREAK** (or its equivalent). Later, when string comparison and string length routines are introduced, an intelligent termination condition can be written for this program.

Formatted Input/Output

The *scanf* and *printf* routines are powerful tools enabling you to read and write data in formatted form, respectively.

Scanf

Scanf is the formatted-input library routine. Its syntax is:

```
scanf(format, [item[, item ...]]);
```

where *format* is a character pointer to a character string (or the character string itself enclosed in double quotes), and *item* is the *address* of a variable.

The purpose of the format is to specify how the data to be read is presented on **stdin**, and what types of data are found there. The format consists of two things: *conversion specifications*, and literal characters.

Conversion Specifications

A conversion specification is a character sequence which tells *scanf* how to interpret the data received at that point in the input. For example, if a conversion specification says "treat the next piece of data as a decimal integer", then that data is interpreted and stored as a decimal integer.

In the format, a conversion specification is introduced by a percent sign (%), optionally followed by an asterisk (*) (called the *assignment suppression character*), optionally followed by an integer value (called the *field width*). The conversion specification is terminated by a character specifying the type of data to expect. These terminating characters are called *conversion characters*.

When a conversion specification is encountered in a format, it is matched up with the corresponding item in the item list. The data formatted by that specification is then stored in the location pointed to by that item. For example, if there are four conversion specifications in a format, the first specification is matched up with the first item, the second specification with the second item, and so on.

The number of conversion specifications in the format is directly related to the number of items specified in the item list. With one exception, there must be at least as many items as there are conversion specifications in the format. If there are too few items in the item list, an error occurs; if there are too many, the excess items are simply ignored. The one exception occurs when the assignment suppression character (*) is used. If an asterisk occurs immediately after the percent sign (before the field width, if any), then the data formatted by that conversion specification is discarded. No corresponding item is expected in the item list. This is useful for skipping over unwanted data in the input.

Conversion Characters

There are eight conversion characters available. Three of them are used to format integer data, three are used to format character data, and two are used for floating-point data.

The integer conversion characters are:

d a decimal integer is expected;

o an octal integer is expected;

x a hexadecimal integer is expected;

The character conversion characters are:

c a single character is expected;

s a character string is expected;

[a character string is expected;

The floating-point conversion characters are:

e, f a floating-point number is expected;

Integer Conversion Characters

The **d**, **o**, and **x** conversion characters read characters from **stdin** until an inappropriate character is encountered, or until the number of characters specified by the field width, if given, is exhausted (whichever comes first).

For **d**, an inappropriate character is any character *except* +, −, and 0 thru 9. For **o**, an inappropriate character is any character *except* +, −, and 0 thru 9. That's right − 8 and 9 are allowed in octal numbers! If you enter, say, 1294 to be interpreted by the **o** conversion character, it still interprets the entire number as octal, and converts the digits to the octal digit range. Thus, 1294 actually gets stored as 1314 (octal). For **x**, an inappropriate character is any character *except* +, −, 0 thru 9, and the characters a − f and A thru F. Note that negative octal and hexadecimal values are stored in their 2's complement form with sign extension. Thus, they may look unfamiliar if you print them out later (using *printf* − see below).

These integer conversion characters can be capitalized or preceded by a lower-case L (**l**) to indicate that a **long int** should be expected rather than an **int**. They can also be preceded by **h** to indicate a **short int**. The corresponding items in the item list for these conversion characters must be pointers to integer variables of the appropriate length.

Character Conversion Characters

The **c** conversion character reads the next character from **stdin**, no matter what that character is. The corresponding item in the item list must be a pointer to a character variable. If a field width is specified, then the number of characters indicated by the field width are read. In this case, the corresponding item must refer to a character array large enough to hold the characters read.

Note that strings read using the **c** conversion character are **not** automatically terminated with a NULL character in the array. Since all C library routines which utilize strings assume the existence of a NULL terminator, be sure you add the NULL character yourself. Otherwise, library routines are not able to tell where the string ends, and you'll get puzzling results.

The **s** conversion character reads a character string from **stdin** which is delimited by one or more space characters (blanks, tabs, or new-lines). If no field width is given, the input string consists of all characters from the first non-space character up to (but not including) the first space character. Any initial space characters are skipped over. If a field width is given, then characters are read, beginning with the first non-space character, up to the first space character, or until the number of characters specified by the field width is reached (whichever comes first). The corresponding item in the item list must refer to a character array large enough to hold the characters read, plus a terminating NULL character which is added automatically.

An important point to remember about the **s** conversion character is that it *cannot* be made to read a space character as part of a string. Space characters are always skipped over at the beginning of a string, and they terminate reading whenever they occur in the string. For example, suppose you want to read the first character from the following input line:

```
"          Hello, there!"
```

(10 spaces followed by "Hello, there!", the double quotes being added for clarity). If you use **%c**, you get a space character. However, if you use **%1s**, you get "H" (the first non-space character in the input).

The **[** conversion character also reads a character string from **stdin**. However, this character should be used when a string is *not* to be delimited by space characters. The left bracket is followed by a list of characters, and is terminated by a right bracket. If the first character after the left bracket is a circumflex (^), then characters are read from **stdin** until a character is read which matches one of the characters between the brackets. If the first character is *not* a circumflex, then characters are read from **stdin** until a character *not* occurring between the brackets is found. The corresponding item in the item list must refer to a character array large enough to hold the characters read, plus a terminating NULL character which is added automatically.

The three string conversion characters provide you with a complete set of string-reading capabilities. The **c** conversion character can be used to read *any* single character, or to read a character string *when the exact number of characters in the string is known beforehand.* The **s** conversion character enables you to read any character string *which is delimited by space characters, and is of unknown length.* Finally, the **[** conversion character enables you to read character strings *that are delimited by characters other than space characters, and which are of unknown length.*

Floating-point Conversion Characters

The **e** and **f** conversion characters read characters from **stdin** until an inappropriate character is encountered, or until the number of characters specified by the field width, if given, is exhausted (whichever comes first).

Both **e** and **f** expect data in the following form: an optionally signed string of digits (possibly containing a decimal point), followed by an optional exponent field consisting of an **E** or **e** followed by an optionally signed integer. Thus, an inappropriate character is any character *except* +, −, ., 0 thru 9, E, or e.

These floating-point conversion characters may be capitalized, or preceded by a lower-case L (**l**), to indicate that a **double** value is expected rather than a **float**. The corresponding items in the item list for these conversion characters must be pointers to floating-point variables of the appropriate length.

Literal Characters

Any characters included in the format which are *not* part of a conversion specification are *literal characters.* A literal character is expected to occur in the input at exactly that point. Note that since the percent sign is used to introduce a conversion specification, you must type two percent signs (%%) to get a literal percent sign.

Examples

Suppose that you have to read the following line of data:

```
NAME: Joe Kool; AGE: 27; PROF: Elec Engr; SAL: 39550
```

To get the vital data, you must read two strings (containing spaces), and two integers. You also have data that should be ignored, such as the semicolons and the identifying strings ("NAME:"). How do you go about reading this?

First, note that the identifying strings are always delimited by space characters. This suggests use of the **s** conversion character to read them. Second, you can never know the exact sizes of the NAME and PROF fields, but note that they are both terminated by a semicolon. Thus, you can use [to read them. Finally, the **d** conversion character can be used to read both integers. (Note: on 16-bit processors, you probably need to use a **long int** to read the salaries. Thus, **D** or **ld** should be used instead of **d**.)

The following code fragment successfully reads this data:

```
char name[40], prof[40];
int age, salary;
   ...
scanf("%*s%*[ ]%[^;]%*c%*s%d%*c%*s%*[ ]%[^;]%*c%*s%d",name,&age,\
prof,&salary);
```

For easier understanding, break the format into pieces:

%*s This reads the string "NAME:". Since an asterisk is given, the string is simply read and discarded.

%*[ ] This gets rid of all blanks occurring between "NAME:" and the employee's name. Note that this gets rid of one or more blanks, giving the format some flexibility.

%[^;] This reads all characters from the current character up to a semicolon, and assigns the characters to the array *name*.

%*c This gets rid of the semicolon left over after reading the name.

%*s This reads the next identifying string, "AGE:", and discards it.

%d This reads the integer age given, and assigns it to *age*. The semicolon after the age terminates %d, because that character is not appropriate for an integer value. Note that the address of *age* is given in the item list (&age) instead of the variable name itself. If this is not done, a memory fault occurs at run-time.

%*c This gets rid of the semicolon following the age.

%*s This reads the next identifying string, "PROF:", and discards it.

%*[ ] This removes all blanks between "PROF:" and the next string.

%[^;] This reads all characters up to the next semicolon, and assigns them to the character array *prof*.

%*c This gets rid of the semicolon following the profession string.

%*s This reads the final identifying string, "SAL:", and discards it.

%d This reads the final integer and assigns it to the integer variable *salary*. Again, note that the address of *salary* is given, not the variable name itself.

Although somewhat confusing to read, this format is quite flexible, since it allows for multiple spaces between items and varying identifying strings (i.e. "PROFESSION:" could be specified instead of "PROF:"). The following *scanf* call reads the same data, but is much less flexible:

```
scanf("NAME: %[^;]; AGE:%d; PROF: %[^;]; SAL: %d",name,&age,prof,&salary);
```

Here, literal characters are used to exactly match the characters in the input line. This works fine if you can be sure that the data always appears in this form. If one typing variation is made, however, such as typing "SALARY:" instead of "SAL:", the *scanf* fails.

Scanf waits for more data as long as there are unsatisfied conversion specifications in the format. Thus, a *scanf* call like

```
scanf("%f%f%f", &float1, &float2, &float3);
```

where *float1*, *float2*, and *float3* are all variables of type **float**, allows you to enter data in several ways. For example,

```
14.77 29.8 13.0
```

is read correctly by *scanf*, as is

```
14.77 RETURN   29.8 RETURN   13.0 RETURN
```

Note: using decimal points in floating-point data is recommended whenever floating-point variables are being read. However, *scanf* converts integer data to floating-point if the conversion specification so demands. Thus, "13.0" in the previous example could have been entered as "13" with no side effects.

As a final example, consider the input string

```
abcdef137 d14.77ghijklmnop
```

Suppose that the following code fragment is used to read this string:

```
char arr1[10], arr2[10], arr3[10], arr4[10];
float float1;
scanf("%4c%[^3]%6c%f%[ghijkl]",arr1,arr2,arr3,&float1,arr4);
```

What values are stored in the variables listed? (Give this some thought before reading on.) As before, break up the format into separate conversion specifications, and see what data is demanded by each.

%4c reads four characters, and assigns them to *arr1*. Thus, the string "abcd" is assigned to *arr1*. Note that an extra character, NULL, is appended to the end of the string.

%[^3] reads all characters from the current character up to the character "3". This assigns "ef1", along with an added NULL character, to the array *arr2*.

%6c reads the next six characters and stores them in the array *arr3*. Thus, "37 d14" is assigned to *arr3*, terminated by a NULL character.

%f reads a floating-point value which, due to the lack of a field width, is terminated by the first "inappropriate" character. Thus, the value ".77" is assigned to *float1*.

%[ghijkl] reads all characters up to the first character not occurring between the brackets. This stores the string "ghijkl", along with an appended NULL character, in the array *arr4*.

Note that there are some characters left in **stdin** that were not read. What happens to these characters? Do they just go away? No! Any characters left unread in the input remain there! This can cause unexpected errors. Suppose that, later in the above program fragment, you want to read a string from **stdin** using **%s**. No matter what string you type in as input, it will never be read, because the **%s** conversion specification is satisfied by reading "mnop" – the characters left over from the previous read operation! To solve this, always be sure you have read the entire current line of input before attempting to read the next. To fix this in the previous *scanf* example, just add a **%*s** conversion specification at the end of the format. This reads and discards the left-over characters.

Printf

Printf is the other half of the formatted I/O team. It enables you to output data in formatted form. Its syntax is identical to that of *scanf*:

```
printf(format, [item[, item ...]\¦]);
```

where the *format* is a pointer to a character string (or the character string itself enclosed in double quotes) which specifies the format and content of the data to be printed. Each *item* is a variable or expression specifying the data to print.

Printf's format is similar in many respects to that of *scanf*. It is made up of conversion specifications and literal characters. As in *scanf*, literal characters are all characters that are not part of a conversion specification. Literal characters are printed on **stdout** exactly as they appear in the format.

Literal Characters

Included in the list of literal characters are *escape sequences*, which are sequences beginning with a backslash (\e) which stand for other characters. The following list shows the escape sequences defined for *printf* (and *scanf*, though less frequently used):

\b backspace;

\n new-line (carriage-return/line-feed sequence); output begins at the beginning of a new line;

\r carriage-return without a line-feed; output begins at the beginning of the current line (data already printed on that line is over-printed);

\t tab;

\\ literal backslash;

nnn the character represented by the octal number *nnn* in the ASCII character set. *Nnn* must begin with a zero. For example, **007** is an ASCII bell, which beeps the bell on your terminal.

Conversion Specifications

A conversion specification for *printf* is very similar to that of *scanf*, but is a bit more complicated. The following list shows the different components of a conversion specification in their correct sequence:

1. A percent sign (%), which signals the beginning of a conversion specification; to output a literal percent sign, you must type two percent signs (%%);
2. Zero or more *flags*, which affect the way a value is printed (see below);
3. an optional decimal digit string which specifies a minimum *field width*;
4. an optional *precision* consisting of a dot (.) followed by a decimal digit string;
5. an optional **l** (lower-case L) or **h**, indicating a **l**ong or **sh**ort integer argument;
6. a *conversion character*, which indicates the type of data to be converted and printed.

As in *scanf*, a one-to-one correlation must exist between each specification encountered and each item in the item list.

The available *flags* are:

- `-` causes the data to be left-justified within its output field. Normally, the data is right-justified.
- `+` causes all signed data to begin with a sign (+ or −). Normally, only negative values have signs.
- `blank` causes a blank to be inserted before a positive signed value. This is used to line up positive and negative values in columnar data. Otherwise, the first digit of a positive value is lined up with the negative sign of a negative value. If the "blank" and "+" flags both appear, the "blank" flag is ignored.
- `#` causes the data to be printed in an "alternate form". Refer to the descriptions of the conversion characters below for details concerning the effects of this flag.

A *field width*, if specified, determines the *minimum* number of spaces allocated to the output field for the particular piece of data being printed. If the data happens to be smaller than the field width, the data is blank-padded on the left (or on the right, if the − flag is specified) to fill the field. If the data is larger than the field width, the field width is simply expanded to accommodate the data. An insufficient field width *never* causes data to be truncated. If no field width is specified, the resulting field is made just large enough to hold the data.

The *precision* is a value which means different things depending on the conversion character specified. Refer to the descriptions of the conversion characters below for more details.

Note: a field width or precision may be replaced by an asterisk (*). If so, the next item in the item list is fetched, and its value is used as the field width or precision. The item fetched must be an integer.

Conversion Characters

conversion character specifies the type of data to expect in the item list, and causes the data to be formatted and printed appropriately. The integer conversion characters are:

d an integer *item* is converted to signed decimal. The precision, if given, specifies the minimum number of digits to appear. If the value has fewer digits than that specified by the precision, the value is expanded with leading zeros. The default precision is one (1). A null string results if a zero value is printed with a zero precision. The # flag has no effect.

u an integer *item* is converted to unsigned decimal. The effects of the precision and the # flag are the same as for **d**.

o an integer *item* is converted to unsigned octal. The # flag, if specified, causes the precision to be expanded, and the octal value is printed with a leading zero (a C convention). The precision behaves the same as in **d** above, except that printing a zero value with a zero precision results in only the leading zero being printed, if the # flag is specified.

x an integer *item* is converted to hexadecimal. The letters **abcdef** are used in printing hexadecimal values. The # flag, if specified, causes the precision to be expanded, and the hexadecimal value is printed with a leading "0x" (a C convention). The precision behaves as in **d** above, except that printing a zero value with a zero precision results in only the leading "0x" being printed, if the # flag is specified.

X same as **x** above, except that the letters **ABCDEF** are used to print the hexadecimal value, and the # flag causes the value to be printed with a leading "0X".

The character conversion characters are as follows:

c the character specified by the **char** *item* is printed. The precision is meaningless, and the # flag has no effect.

s the string pointed to by the character pointer *item* is printed. If a precision is specified, characters from the string are printed until the number of characters indicated by the precision has been reached, or until a NULL character is encountered, whichever comes first. If the precision is omitted, all characters up to the first NULL character are printed. The # flag has no effect.

The floating-point conversion characters are:

f the **float** or **double** *item* is converted to decimal notation in *style f*, that is, in the form

[-]ddd.ddd

where the number of digits after the decimal point is equal to the precision. If no precision is specified, six (6) digits are printed after the decimal point. If the precision is explicitly zero, the decimal point is eliminated entirely. If the # flag is specified, a decimal point always appears, even if no digits follow the decimal point.

e the **float** or **double** *item* is converted to scientific notation in *style e*; that is, in the form

[−]d.ddd**Ae** ± ddd

where there is always one digit before the decimal point. The number of digits after the decimal point is equal to the precision. If no precision is given, six (6) digits are printed after the decimal point. If the precision is explicitly zero, the decimal point is eliminated entirely. The exponent always contains exactly three digits. If the # flag is specified, the result always contains a decimal point, even if no digits follow the decimal point.

E same as **e** above, except that **E** is used to introduce the exponent instead of **e** (*style E*).

g the **float** or **double** *item* is converted to either *style f* or *style e*, depending on the size of the exponent. If the exponent resulting from the conversion is less than −4 or greater than the precision, *style e* is used. Otherwise, *style f* is used. The precision specifies the number of significant digits. Trailing zeros are removed from the result, and a decimal point appears only if it is followed by a digit. If the # flag is specified, the result always has a decimal point, even if no digits follow the decimal point, and trailing zeros are *not* removed.

G same as the **g** conversion above, except that *style E* is used instead of *style e*.

The *items* in the item list may be variable names or expressions. Note that, with the exception of the **s** conversion, pointers are **not** required in the item list (contrast this with *scanf*'s item list). If the **s** conversion is used, a pointer to a character string must be specified.

Examples

Here are some examples of *printf* conversion specifications and a brief description of what they do:

%d output a signed decimal integer. The field width is just large enough to hold the value.

% − *d output a signed decimal integer. The left-justify flag (−) and the blank flag are specified. The asterisk causes a field width value to be extracted from the item list. Thus, the item specifying the desired field width must occur before the item containing the value to be converted by the **d** conversion character.

% + 7.2f output a floating-point value. The + flag causes the value to have an initial sign (+ or −). The value is right-justified in a 7-column field, and has exactly two digits after the decimal point. This conversion specification is ideal for a debit/credit column on a finance worksheet. (If the + sign is not necessary, use the blank flag instead.)

Consider the following program, which reads a number from **stdin**, and prints that number, followed by its square and its cube:

```
#include <stdio.h>
main()
{
    double x;

    printf("Enter your number: ");
    scanf("%F", &x);
    printf("Your number is %g\n", x);
    printf("Its square is %g\nIts cube is %g\n", x*x, x*x*x);
}
```

The **g** conversion character is used so that the decision about whether or not to use an exponent is automated. Note that the item list contains expressions to calculate *x* squared and *x* cubed. Also note that the address of the variable is required in order to read a value for it, but printing requires the variable name itself.

How about a program that accepts a decimal integer, and then prints the integer itself, its square, and its cube in decimal, octal, and hexadecimal? Easy enough:

```
#include <stdio.h>
main()
{
   long n, n2, n3;

/* get value */

   printf("Enter your number: ");
   scanf("%D", &n);

/* print headings */

    printf("\n\n              Decimal     Octal     Hexadecimal\n");

 /* do the computation */

   n2 = n * n;
   n3 = n * n * n;
   printf("n itself:    %7ld   %9lo     %6lx\n", n, n, n);
   printf("n squared:   %7ld   %9lo     %6lx\n", n2, n2, n2);
   printf("n cubed:     %7ld   %9lo     %6lx\n", n3, n3, n3);
}
```

This program prints the headings "Decimal", "Octal", and "Hexadecimal", and then prints out the data in tabular form. Programs which print tabular data always require some tinkering with the formats to make things come out right. Type this in and try it yourself.

Strings are especially easy to manipulate using *printf*. The following simple program illustrates this:

```
#include <stdio.h>
main()
{
   char first[15], last[25];

   printf("Enter your first and last names: ");
   scanf("%s%s", first, last);
   printf("\nWell, hello %s, it's good to meet you!\n", first);
   printf("%s, huh?  Are you any relation to that famous\n", last);
   printf("computer programmer, Mortimer Zigfelder %s?\n", last);
   printf("No, sorry, that was my mistake.  I was thinking of\n");
   printf("O'%s, not %s.\n", last, last);
 }
```

This program shows how easily strings can be inserted in text. Try variations of your own.

Input/Output from/to Strings

Two library routines, *sscanf* and *sprintf*, enable you to read data from a string, and write data into a string. These routines behave identically to *scanf* and *printf*, respectively, except that *sscanf* reads data from a character string instead of from **stdin**, and *sprintf* writes data into a string instead of on **stdout**.

Reading Data from a String

Sscanf enables you to read data directly from a string. The syntax for an *sscanf* call is

```
sscanf(string, format, [item[, item ...]]);
```

where *string* is the name of a character array containing the data to be read, and *format* and *item* are familiar terms from the previous section. Thus, the only difference between *sscanf* and *scanf*, syntactically speaking, is *sscanf*'s inclusion of a new parameter, *string*.

The following program simply reads a string of your choosing from **stdin**, stores it in the character array *string*, and prints out the first word of that string:

```
#include <stdio.h>
main()
{
   char string[80], word[25], *gets();

/* get the string */

   printf("Enter your string: ");
   gets(string);

/* get the first word */

   sscanf(string, "%s", word);
   printf("The first word is %s.\n", word);
}
```

Of course, *sscanf* is rarely used in this way. *Sscanf* is more often used as a means of converting ASCII characters into other forms, such as integer or floating-point values. For example, the following program uses *sscanf* to implement a five-function calculator:

```
#include <stdio.h>
main()
{
   char line[80], *gets(), op[4];
   long n1, n2;
   double arg1, arg2;

/* print prompt (>) and get input */

   printf("\n> ");
   gets(line);
```

```
/* begin loop */

        while(line[0] != 'q') {
            sscanf(line, "%*s%s", op);
            if(op[0] == '+') {
                sscanf(line, "%F%*s%F", &arg1, &arg2);
                printf("Answer:  %g\n\n", arg1+arg2);
        } else if(op[0] == '¦-') {
            sscanf(line, "%F%*s%F", &arg1, &arg2);
            printf("Answer:  %g\n\n", arg1¦-arg2);
        } else if(op[0] == '*') {
            sscanf(line, "%F%*s%F", &arg1, &arg2);
            printf("Answer:  %g\n\n", arg1*arg2);
        } else if(op[0] == '/') {
            sscanf(line, "%F%*s%F", &arg1, &arg2);
            printf("Answer:  %g\n\n", arg1/arg2);
        } else if(op[0] == '%') {
            sscanf(line, "%D%*s%D", &n1, &n2);
            while(n1 >= n2)
                n1 ¦-= n2;
            printf("Answer:  %ld\n\n", n1);
        } else
            printf("Can't recognize operator: %s\n\n", op);
        printf("> ");
        gets(line);
    }
}
```

The calculator program accepts input lines having the form

value <operator> *value*

where *value* is any number, and <operator> is the symbol +, −, *, /, or %, standing for addition, subtraction, multiplication, division, or remainder, respectively. All functions except remainder are handled internally in floating-point, but values for these functions can be typed with or without a decimal point. Values for the remainder function must not have a decimal point. There must be at least one space between each value and the operator.

Note the use of *sscanf* in this program. The entire input line is read using *gets*. Then, the different parts of the input line are read from *line* using *sscanf*. Notice that the input line is stored as an ASCII string in *line*, but portions of it are converted to floating-point or integer values, depending on the operator.

Examples of valid entries are

```
15.778 * 3.89
27 % 8
17 + 39.72
```
etc.

The program terminates when it reads a line beginning with "q", such as "quit".

There are two things that differ between reading data from **stdin**, and reading data from a string. First, you remember that reading data from **stdin** causes that data to "go away" – – it is no longer contained in **stdin**. This is *not* true for a string. Since the data is stored in a string, it is always there, even if that data has been read several times. Second, since the data read from **stdin** disappears as you read it, the next read operation from **stdin** always begins where the previous read operation terminated. This is *not* true when you read from a string using *sscanf*. Each successive read operation begins *at the beginning of the string*. Thus, if you want to read five words from a string stored in a character array, you must read them in a single *sscanf* call. If you try to read one word in five separate *sscanf* calls, each call starts reading at the beginning of the string, and you end up reading the same word five times!

Writing Data Into a String

The *sprintf* routine enables you to write data into a character string. Its syntax is

```
sprintf(string, format, [item[, item ...]]);
```

which is identical to that of *sscanf*. *String* is the name of the character string into which the data is written. *Format* and *item* are familiar terms from the previous discussion of *printf*. In fact, the only difference between *sprintf* and *printf* is that *sprintf* writes data into a character array, while *printf* writes data on **stdout**.

The following program acts as a "formatter" for personal data. Suppose that this program is used to provide a "friendly" user interface to gather personal data. The data received is then reformatted into a string which is passed along to another program, such as a data base maintainer. The string contains the data entered by the user, but in a form using strict field widths for the various pieces of data. The data base program requires these field widths in order for the data to be processed correctly, but there is no reason to burden the user with this requirement. This "formatter" program lets the user enter data in a convenient form (without the fixed field restrictions imposed by the data base).

```
#include <stdio.h>
main()
{
    char name[31], prof[31], hdate[7], curve[3], string[81];
    char *format = "%30s%2d%30s%6ld%6s%2d%2s";
    int age, rank;
    long salary;

/* start asking questions */

    printf("\nName (30 chars max): ");
    gets(name);
    while(name[0] != ']') {
        printf("Age: ");
        scanf("%d%*c", &age);
        printf("Job title (30 chars max): ");
        gets(prof);
        printf("Salary (6 digits max, no comma): ");
        scanf("%D%*c", &salary);
        printf("Hire date (numerical MMDDYY): ");
        gets(hdate);
        printf("Percentile ranking (omit \"%%\"): ");
        scanf("%d%*c", &rank);
        printf("Pay curve: ");
        gets(curve);
```

```
/* format string */

        sprintf(string,format,name,age,prof,salary,hdate,rank,curve);
        printf("\n%s\n", string);

/* start next round */

        printf("\nName (30 chars max): ");
        gets(name);
    }
}
```

This program asks you questions to obtain typical company information such as name, age, job title, salary, hire date, ranking, and pay curve. This data is then packed into a 78-character string using *sprintf*. The string is printed on your screen in this program, but in an actual working environment, this string would probably be passed directly to the data base program. Note that *sprintf*'s format is specified as an explicit character pointer. When lengthy, unchanging formats are used, this is often more convenient than typing the entire format string, especially if the item list is long.

As an exercise, consider the *scanf* calls in the previous program. Notice that a **%*c** conversion specification is included in the formats of the *scanf*s which are reading integer values (age, salary, rank). Why is this necessary? If you aren't sure, take the **%*c**'s out of those formats, re-compile the program, run it, and note its behavior. (Hint: remember that a new-line character terminates the read operation for **%d** and **%D** conversions, and leaves the new-line unread in **stdin**.)

Input/Output Using Ordinary Files

So far, you have been using library routines which can perform I/O only by using **stdin** and **stdout**. This section introduces routines that enable you to open existing ordinary files for reading, writing, or both, and to create ordinary files. Routines that enable you to perform I/O to and from ordinary files are also described.

Opening Ordinary Files

Before a file can be read from or written to, it must be **opened**. A file is opened using the *fopen* library routine. The syntax of an *fopen* call is

```
fopen(<filename>, <type>);
```

where <filename> is a character pointer to a character string specifying the name of the file to be opened, and <type> is a character pointer to a one- or two-character string specifying the I/O operation for which the file is opened. The available <type>s are:

r opens the file for reading at the beginning of the file. The file must already exist, or an error occurs.

w opens the file for writing at the beginning of the file. If the file exists, its previous contents are destroyed. If the file does not exist, it is created.

a opens the file for writing at the end of the file (appends data to the end of the file). If the file does not exist, it is created for writing.

r+ opens the file for both reading and writing, starting at the beginning of the file. The file must already exist, or an error occurs.

w+ opens the file for both reading and writing, starting at the beginning of the file. If the file already exists, its previous contents are destroyed. If the file does not exist, it is created.

a+ opens the file for both reading and writing, starting at the end of the file. If the file does not exist, it is created.

When a file is opened for an append operation (<type> is "a" or "a+"), it is impossible to overwrite the existing file contents. *Fseek* can be used to reposition the file pointer to any position in the file, but when output is written to the file, the pointer is disregarded. When the append operation (which begins at the end of the existing file) is completed, the file pointer is repositioned to the end of the appended output.

In exchange for a *filename* and a *type*, *fopen* opens a "pathway" between your program and the file. This "pathway" is called a *stream*. If you open the file for reading, then the stream provides one-way data transfer from the file to your program. If you open the file for writing, then data transfer flows from your program to the file. Finally, if the file is opened for both reading and writing, the resulting stream is bi-directional.

Fopen also associates a *buffer* with the stream. This gives the stream the ability to store a small amount of data. By default, the capacity of the buffer is equal to **BUFSIZ** bytes, where BUFSIZ is a constant defined in **stdio.h**. For the Series 200 and Series 500 computers, BUFSIZ is defined to be 1024.

The buffer size can be increased, decreased, or set to zero by using *setbuf* or *setvbuf*. If the buffer size is allowed to remain at default size, a maximum of BUFSIZ bytes of data can be present on the stream at any given time. If the buffer size is reduced to zero, then the stream can transfer only one byte at a time.

Since *fopen* takes care of all the intricacies of building a stream and allocating a buffer, all you need to know is how to find your end of the stream. *Fopen* provides you with this information by returning to you a value called a *file pointer* (often called a *stream pointer*). A file pointer "points" to the newly-created stream, and keeps track of where the next I/O operation takes place (in the form of a byte offset relative to the beginning of the associated buffer).

Is all this talk about streams and data transfer from a source to a destination beginning to sound familiar? Do you remember the "pipeline and water" analogy given at the beginning of this section? These two discussions should sound almost identical, because **stdin**, **stdout**, and **stderr** are actually file pointers to pre-opened streams! **Stdin** is a file pointer to a stream which transfers data from your tty (terminal) file to your program. **Stdout** and **stderr** are file pointers to two *different* streams which both transfer data from your program to your tty file. Be sure to note that **stdout** and **stderr** are different streams flowing in the same direction between the same two points!

Once you have a file pointer in your possession, you need never refer to the open file by its name again. A file pointer provides access to all the information needed by other standard I/O routines to read from or write to the file.

The following program fragment shows how the *fopen* routine is used:

```
#include <stdio.h>
main()
{
    FILE *fp;

    fp = fopen("/users/tom/bin/datafile", "r");
    if(fp == NULL) {
        printf("Can't open datafile.\n");
        exit(1);
    }
    ...
}
```

This *fopen* call, if successful, opens */users/tom/bin/datafile* for reading. The file pointer returned by *fopen* is stored in *fp*. Note that *fp*'s value is checked to see if it is NULL. This is because *fopen* returns a NULL pointer if the indicated file cannot be opened. It is good practice to check the value of a file pointer – – this is the only error indication facility that *fopen* provides.

The previous example also introduces a new type declaration, **FILE**. The FILE declaration is defined in **stdio.h**. In the example above, it defines *fp* as a variable containing a file pointer. Note that explicit declarations of functions returning file pointers is unneccessary – – **stdio.h** declares all such functions for you.

Before moving on, keep in mind that several things can stop you from successfully opening a file. First, HP-UX limits the number of files simultaneously open in a process (refer to the System Administrator Manual supplied with your system to find your system's limit). Remember that **stdin**, **stdout**, and **stderr** are automatically opened for you, so the maximum you can explicitly open is three fewer than the system limit. Second, you must have permission to open the file for the particular *type* you have specified (this permission is granted or denied by the file's mode). Third, trying to open a non-existent file using *type* **r** or **r+** always fails. Fourth, if the *filename* is specified incorrectly, contains a non-existent directory name, or contains an intermediate component which is not a directory, the open fails. This is not a complete list, but it contains some of the common reasons why an attempt to open a file might fail.

Single-character Input/Output

Now that you know how to open files and obtain file pointers, you have a whole new set of I/O routines at your disposal, enabling you to perform all kinds of I/O operations. In fact, there are about three times as many available routines that utilize file pointers as there are routines that are limited to **stdin** and **stdout** only!

In this section, only those routines that read or write one character at a time are discussed. These routines are *getc*, *putc*, *fgetc*, and *fputc*. *Getc* and *putc* are macros defined in **stdio.h** which read one character from the specified stream, and write one character on the specified stream, respectively. They have the following syntax:

```
getc(stream);
putc(c, stream);
```

where *stream* is a file pointer obtained from *fopen*, and *c* is a variable of type **char** (or **int**) indicating the character to write on the indicated stream. A simple version of the HP-UX *cat* command can be written using these routines:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
    int c;
    FILE *fp;

    if(argc != 2) {
        printf("Usage:  cat file\n");
        exit(1);
    }

    fp = fopen(argv[1], "r");
    if(fp == NULL) {
        printf("Can't open %s.\n", argv[1]);
        exit(1);
    }

    while((c = getc(fp)) != EOF)
        putc(c, stdout);
    putc('\n', stdout);

    exit(0);
}
```

This program accepts a single argument which is assumed to be the name of a file whose contents are to be printed on the user's terminal. The specified file is opened for reading, and the resulting file pointer *fp* is used in *getc* to read a character from the file. Each character read is written on **stdout** using *putc* (note that **stdout**, as well as **stdin** and **stderr**, are perfectly legal file pointers). The reading and writing loop is terminated when the constant EOF is returned from *getc*, indicating that the end of the file has been reached. This constant is defined in **stdio.h**.

Note that *getc* and *putc* can be made to behave exactly like the *getchar* and *putchar* routines discussed earlier by specifying the appropriate file pointer. In other words,

```
getc(stdin);
```

is identical to

```
getchar();
```

and

```
putc(c, stdout);
```

is identical to

```
putchar(c);
```

Thus, the *putc* call in the previous program could just as easily have been

```
putchar(c);
```

without altering the behavior of the program. However, if the destination of the data is somewhere other than the user's terminal, the flexibility of *putc* is required. Take, for example, the following program, which is a simple version of the HP-UX *cp* command:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
    int c;
    FILE *from, *to;

    if(argc != 3) {
        printf("Usage:  cp fromfile tofile\n");
        exit(1);
    }

    from = fopen(argv[1], "r");
    if(from == NULL) {
        printf("Can't open %s.\n", argv[1]);
        exit(1);
    }
    to = fopen(argv[2], "w");
    if(to == NULL) {
        printf("Can't create %s.\n", argv[2]);
        exit(1);
   }

   while((c = getc(from)) != EOF)
        putc(c, to);

   exit(0);
}
```

This program accepts two arguments. The first is the name of the file to be copied, and the second is the name of the file to be created. The first file is opened for reading, and the second file is created for writing. The data from the first file is then copied directly to the newly-created file.

The *fgetc* and *fputc* routines are actual functions, not macros. Their syntax and usage is identical to that of *getc* and *putc*, so no examples are given here illustrating their use. However, here are some distinctions between the macro and function versions of these routines to help you decide which to use:

- A function call takes time, since the function call still exists at run-time. A macro call, however, takes no time at all, because the macro call is replaced with the actual code making up the macro during compilation, before run-time. Thus, generally speaking, programs containing macros run faster than programs containing the equivalent function calls.
- A function's code is localized in one section of the program. Each function call causes a jump to that section to execute the function. A macro call, however, is replaced with its code everywhere that macro call appears. Thus, programs containing macro calls generally require more space than programs containing the equivalent function calls.
- You may take the address of a function, and pass it as an argument. You cannot do this with a macro.

Given these guidelines, decide which routines to use based on your own constraints.

Character Push-Back

The *ungetc* routine enables you to push back a single character onto an input stream. This character is then returned by the next *getc* call (or equivalent).

Ungetc's syntax is as follows:

```
ungetc(c, stream);
```

where *c* is the character to be pushed back, and *stream* is the input stream where the push-back is to occur. Note that *c must* be a character that has been previously read from *stream*.

The following program simply reads one character from **stdin**, pushes it back onto **stdin**, re-reads the character, and checks to make sure that this character and the character originally pushed back are the same. A message is printed on **stdout** stating the outcome of the comparison.

```
#include <stdio.h>
main()
{
   int c1, c2;

   c1 = getchar();
   ungetc(c1, stdin);
   c2 = getchar();
   if(c1 == c2)
      printf("They're the same!\n");
   else

      printf("Oops!  They're different!\n");
}
```

One character's worth of push-back is guaranteed as long as something has been read from the stream prior to the push-back attempt, and provided that the stream is buffered. More characters could possibly be pushed back, but determining exactly how many characters of push-back you can safely perform is quite possibly not worth the effort. However, for completeness, the following statement is included as a method for determining the number of characters of push-back available at any given time:

```
numpb = ftell(stream) % BUFSIZ + 1;
```

where *ftell* is a function discussed in a later section, *stream* is a file pointer, and BUFSIZ is a constant defined in **stdio.h** containing the size of the buffer in bytes. After execution, *numpb* contains the number of characters of push-back available at that time.

String Input/Output

The *fgets* and *fputs* routines enable you to read or write strings from or to specified streams. Their syntax is as follows:

```
fgets(string, n, stream);
fputs(string, stream);
```

where *string* is a pointer to a character string, and *stream* is a file pointer to the input or output stream.

Fgets reads a character string from the specified *stream*, and stores it in the character array pointed to by *string*. *Fgets* reads $n-1$ characters, or up to a new-line character, whichever comes first. If a new-line character is encountered, it is retained as part of the string (contrast this with *gets*, which replaces the new-line with a NULL character). *Fgets* appends a NULL character to the string.

Fputs writes the character string pointed to by *string* on the specified *stream*, stopping when a NULL character is encountered. *Fputs* does *not* append a new-line character to the string when it is written. This is because *fputs* is intended for use with *fgets*, which incorporates a new-line character into the string if a new-line is encountered in the input.

The *cp* program written earlier can be re-written using *fgets* and *fputs*:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   char c, line[256], *fgets();
   FILE *from, *to;

   if(argc != 3) {
      printf("Usage:  cp fromfile tofile\n");
      exit(1);
   }

   from = fopen(argv[1], "r");
   if(from == NULL) {
      printf("Can't open %s.\n", argv[1]);
      exit(1);
   }
```

```
    to = fopen(argv[2], "w");
    if(to == NULL) {
        printf("Can't create %s.\n", argv[2]);
        exit(1);
    }

    while(fgets(line, 256, from) != NULL)
        fputs(line, to);

    exit(0);
}
```

This program functions exactly like the previous version of *cp* above. Note that *fgets*'s return value is compared to NULL in the **while** loop, since *fgets* returns the NULL pointer when it reaches the end of its input.

This program can easily be converted to a simple *cat* command. It only requires four changes. Can you see what they are? First, change the *argc* comparison such that it reads

```
if(argc != 2) ...
```

(You might also want to change the associated usage message!) Second, remove the *to* file pointer, since you don't need it anymore. Third, remove the block of code which uses *fopen* to open the new file, and assigns a value to *to*. Fourth, change the *fputs* call such that it reads

```
fputs(line, stdout);
```

Here's the new *cat* command:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
    char c, line[256], *fgets();
    FILE *from;

    if(argc < 2) {
        printf("Usage:  cat file\n");
        exit(1);
    }

    from = fopen(argv[1], "r");
    if(from == NULL) {
        printf("Can't open %s.\n", argv[1]);
        exit(1);
    }

    while(fgets(line, 256, from) != NULL)
        fputs(line, stdout);

    exit(0);
}
```

Formatted Input/Output

Just as there are versions of *scanf* and *printf* which perform string I/O, so there are versions which enable I/O using files. *Fscanf* enables you to read data of all types from a specified stream, and *fprintf* provides the capability of writing data on a stream. Their syntax is as follows:

```
fscanf(stream, format, [item[, item ...]]);
fprintf(stream, format, [item[, item ...]]);
```

Stream is a file pointer to an open stream. *Format* and *item* should be familiar terms from previous discussions.

The following program illustrates the use of the *fscanf* and *fprintf* routines:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int count = 0;
   FILE *file;

   if(argc != 2) {
      fprintf(stderr, "Usage:  wdcnt filename\n");
      exit(1);
   }

   file = fopen(argv[1], "r");
   if(file == NULL) {
      fprintf(stderr, "Can't open %s.\n", argv[1]);
      exit(1);
   }

   while(fscanf(file, "%*s") != EOF)
      count++;

   printf("Number of words found:  %d\n", count);

   exit(0);
}
```

This program, named *wdcnt* (for "word count"), counts the number of "words" in the file specified as its only argument. A word is defined as a string of non-space characters.

Note how *fprintf* is used in this program. You learned in a prior discussion that **stderr** is typically used to output error messages or warning statements. In this program, *fprintf* is used to direct error messages to **stderr**. You don't lose anything by doing this, since data written on **stderr** appears on your terminal by default. However, you gain some important flexibility. Now that error output is written on a different stream than normal output, the error output (or the normal output) can be *redirected* to another destination. For example, invoking the previous program as

```
$ wdcnt <file1> 2>errmsgs
```

causes all output arising from erroneous conditions to be collected in the file **errmsgs**. For the *wdcnt* program, this is somewhat trivial, since the program terminates upon any error. However, for programs which output any number of warnings without terminating, this is a very useful capability. Not only does it keep normal, desired output from getting cluttered up with error messages, but it enables you to save output for later examination at your leisure. Thus, it is good programming practice to write error messages and warnings on **stderr**, and use **stdout** (or whatever your destination file is) to output normal data.

Binary Input/Output

The routines described in this section deal with data in its binary form – that is, the data is never converted to ASCII for user viewing. These routines are used to transfer raw data between two points, such as from a variable to a data file, or vice versa.

Two routines, *getw* and *putw*, are used to read or write an integer word (an **int**) to or from a stream, respectively. Their syntax is as follows:

```
getw(stream);
putw(w, stream);
```

where *stream* is a file pointer to the input or output stream, and *w* is the integer word to be output by *putw*.

The following program "sorts" a data file which has presumably been created earlier, and contains raw integer data. The program divides this data file into two new data files, one containing integer data whose absolute value is less than or equal to 32767, the other containing data whose absolute value is larger than 32767.

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int word;
   FILE *dfile, *datale, *datagt;

   if(argc != 2) {
      fprintf(stderr, "usage:  intsort filename\n");
      exit(1);
   }

   dfile = fopen(argv[1], "r");
   if(dfile == NULL) {
      fprintf("Can't open %s.\n", argv[1]);
      exit(1);
   }

   datale = fopen("dfle", "w");
   if(datale == NULL) {
      fprintf("Can't create dfle file.\n");
      exit(1);
   }

   datagt = fopen("dfgt", "w");
```

```
	if(datagt == NULL) {
	   fprintf("Can't create dfgt file.\n");
	   exit(1);
	}

	while((word = getw(dfile)) != EOF) {
	   if(word <= 32767 && word >= -32767)
	      putw(word, datale);
	   else
	      putw(word, datagt);
	}

	exit(0);
}
```

This program reads a word from the specified data file. If its absolute value is less than or equal to 32767, the word is written on a file called **dfle** in the user's current directory. Otherwise, the word is written on a file called **dfgt** in the current directory.

Note that this program works only on machines that use four-byte integers. Also, the comparison between *word* and the constant EOF is faulty, since EOF is defined to be -1, a valid integer. The section entitled *Stream Status Inquiry Routines* describes standard I/O routines which fix this problem.

Both of these routines transfer four bytes at a time. Again, there is no ASCII conversion associated with these routines, so if you attempt to print the contents of a file containing integer data output by *putw*, you will get garbage. Note that it makes little sense to input binary data from **stdin**, as in

```
getw(stdin);
```

unless **stdin** is redirected from a file containing binary data. Using *getw* to read data from your keyboard is futile. If you type in a valid-looking integer, like "1728", *getw* reads the ASCII values of those characters and stores them as an integer. This results in data being read which is *very* different from what you probably intended.

Two other routines, called *fread* and *fwrite*, provide much more flexible binary data input and output. Their syntax is as follows:

```
fread((char *)ptr, sizeof(*ptr), nitems, stream);
fwrite((char *)ptr, sizeof(*ptr), nitems, stream);
```

where *ptr* is a pointer to the beginning of a block (array) of data. This argument is cast as a character pointer because these routines expect a pointer of this type. The second argument specifies the number of bytes per *unit* of data (four bytes per **int**, one byte per **char**, *x* bytes per **struct**, etc.). The C operator **sizeof** is usually used to obtain this value. The third argument, *nitems*, is an integer specifying the number of units of data to read or write. For example, if *ptr* points to the beginning of a structure, **sizeof**(*ptr*) tells how many bytes make up that structure, and *nitems* tells how many structures to read. Actually, the second and third arguments above may be reversed in the argument list with no ill effects, because internally these routines simply multiply the two integers together to obtain the total number of bytes to read. Finally, *stream* is a file pointer to the input or output stream.

As an example, suppose you have a program which keeps track of certain employee data. Each employee is to be described in a single structure. Here is a simple program to do that:

```
#include <stdio.h>
struct emp {
          char        name[40]; /* name */
          char        job[40];  /* job title */
          long        salary;   /* salary */
          char        hire[6]   /* hire date */
          char        curve[2]  /* pay curve */
          int         rank;     /* percentile ranking */
}
#define EMPS 400                /* no.  of employees */
main()
{
     int items;
     struct emp staff[EMPS];
     FILE *data;

     data = fopen("/usr/lib/employees/empdata", "r");
     if(data == NULL) {
        fprintf(stderr, "Can't open employee data file.\n");
        exit(1);
     }

     items = fread((char *)staff, sizeof(staff[0]), EMPS, data);
     if(items != EMPS) {
        fprintf(stderr, "Insufficient data found.\n");
        exit(1);
     }

     fclose(data);
     archive("/usr/lib/employees/empdata");

/* Employee information processing goes here. */

     ...
/* Processing is done.  Write out new employee records. */

     data = fopen("/usr/lib/employees/empdata", "w");
     if(data == NULL) {
        fprintf(stderr, "Can't create new employee file.\n");
        exit(1);
     }

     items = fwrite((char *)staff, sizeof(staff[0]), EMPS, data);
     if(items != EMPS) {
        fprintf(stderr, "Write error!\n");
        exit(1);
     }

     exit(0);
}
archive(filename)
char *filename;
{
     ...
}
```

This program reads the employee information contained in the binary file */usr/lib/employees/empdata*. The data in this file consists of concatenated streams of bytes describing each employee of a certain 400-employee company. The bytes are written such that, when read correctly, the bytes correspond exactly with the **emp** structure defined in the program. The *staff* array is an array of structures containing one structure for each employee.

In the *fread* call, the **sizeof**(*staff*[0]) expression returns the number of bytes in the **emp** structure. Since the same number of bytes are in each employee structure, any element of the *staff* array could have been specified as the **sizeof** argument; *staff*[0] is used in this example. (By counting the number of bytes in each structure member, you can get an approximation of the number of bytes returned by the **sizeof** operator: 40 + 40 + 8 + 6 + 2 + 4 = 100 bytes. This may vary due to padding performed by a programming language, or by machine architecture.) Specifying EMPS as the *nitems* argument tells *fread* to read 400 such structures. Thus, 100 x 400 = 40000 bytes are read, filling in the information for the members of each structure contained in the *staff* array.

The *archive* function is not shown here, but simply saves the old employee information in **empdata** in an employee information archive of some kind. After the information is archived, the **empdata** file is overwritten with the new, updated employee information.

A new routine, called *fclose*, is introduced here. *Fclose* simply closes the stream associated with the file pointer specified. This is necessary in order to re-open the file for writing. Once it is open for writing, *fwrite* is used to overwrite its previous contents with the new data.

One final note about these two routines: they return the number of items of data which have been read or written. Thus, you can compare this number with whatever you specified for *nitems* to see if everything you wanted read or written actually was. This return value is used twice in the above program to flag probable read and write errors.

The *fread* and *fwrite* routines can be made to read *any* type of data. The following examples show some *fread* calls which read several different types of data:

To read a long integer:

```
long nint;
fread((char *)&nint, sizeof(nint), 1, stream);
```

To read an array of 100 long integers:

```
long nint[100];
fread((char *)nint, sizeof(nint[0]), 100, stream);
```

To read a double precision floating-point value:

```
double fpoint;
fread((char *)&fpoint, sizeof(fpoint), 1, stream);
```

To read an array of 50 floating-point values:

```
float fpoint[50];
fread((char *)fpoint, sizeof(fpoint[0]), 50, stream);
```

To get the equivalent *fwrite* calls, just substitute "fwrite" in place of "fread" in the previous examples. You can see how much more flexible *fread* and *fwrite* are than *getw* and *putw*. Whereas *getw* and *putw* are limited to reading or writing a single four-byte integer per call, *fread* and *fwrite* can be made to read or write any number of variables of any type.

Stream Status and Control Routines

This section discusses standard I/O routines which enable you to:

- Determine whether or not an error has occurred on an open stream (*feof*, *ferror*, *clearerr*);
- Re-position the location of the next I/O operation on an open stream (*rewind*, *ftell*, *fseek*);
- Control various attributes of an open stream, such as buffering, flushing, etc. (*fclose*, *setbuf*, *fflush*, *freopen*);
- Convert a file pointer to a file descriptor, and vice versa (*fileno*, *fdopen*).

Stream Status Inquiry Routines

This section describes three routines, *feof*, *ferror*, and *clearerr*, which enable you to determine the status of an open stream at any given time.

Feof is a macro defined in **stdio.h** which returns a non-zero value if the end-of-file has been reached on an input stream. Its syntax is as follows:

```
feof(stream);
```

Do you remember the example program which illustrated the use of *getw* and *putw*? It was noted that comparing *getw*'s return value to the constant EOF was faulty, because *getw* returns an integer, and EOF is defined to be a valid integer (−1). How then do you determine if end-of-file has been reached when routines like *getw* are being used? You use *feof*.

The example program for *getw*/*putw* can be changed to use *feof*:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int word;
   FILE *dfile, *datale, *datagt;

   if(argc != 2) {
      fprintf(stderr, "usage:  intsort filename\n");
      exit(1);
   }

   dfile = fopen(argv[1], "r");
   if(dfile == NULL) {
      fprintf("Can't open %s.\n", argv[1]);
      exit(1);
   }

   datale = fopen("dfle", "w");
```

```
    if(datale == NULL) {
       fprintf("Can't create dfle file.\n");
       exit(1);
    }

    datagt = fopen("dfgt", "w");
    if(datagt == NULL) {
       fprintf("Can't create dfgt file.\n");
       exit(1);
    }

    for(;;) {
       if((word = getw(dfile)) != EOF) {
          if(word <= 32767 && word >= ¦-32767)
             putw(word, datale);
          else
             putw(word, datagt);
        } else {
             if(feof(dfile))
                break;
             else
                putw(word, datale);
        }
    }

    exit(0);
}
```

An infinite loop is set up around the *getw*/*putw* process. Whenever *getw* returns an integer equal to EOF, *feof* is used to find out if end-of-file has been reached. If it has, the loop (and the program) terminates; if not, the integer is written on **dfle**, and the loop continues.

Ferror is a routine which examines the specified stream to determine whether or not a read or write error has occurred. Its syntax is

```
ferror(stream);
```

Ferror, like *feof*, is intended to clarify ambiguous return values from standard I/O routines. Actually, only *getw* and *putw* require the use of *ferror* to determine if an error has occurred. Both of these routines return EOF on end-of-file or error. Since these routines deal with integer data, however, you need *feof* and *ferror* to determine if the EOF returned actually indicated an error or an end-of-file, or if it's just a -1.

If an error has occurred on a stream, *ferror* returns a non-zero value.

Whenever an error occurs on an open stream, a flag is set to indicate the error. It is this flag that *ferror* checks to determine whether or not an error has occurred. This flag is *not* reset when it is checked. Thus, if an error has occurred, the error flag for that stream remains set. This could lead to misleading information if an *ferror* call indicates that an error has occurred, when in reality the error occurred long ago. The *clearerr* routine clears (or resets) the error indication flag for the specified stream. This routine should be used whenever an error has been indicated, so that the same error is not indicated at a later time. *Clearerr*'s syntax is

```
clearerr(stream);
```

Because *ferror* and *clearerr* are used infrequently in typical programs, no examples are given specific to their use. The *feof* example above illustrates the general scenario in which all three of these routines are used.

Re-positioning Stream I/O Operations

There are three routines, *rewind*, *ftell*, and *fseek*, which enable you to move the location of the next I/O operation on an open stream.

Rewind simply positions the next I/O operation at the beginning of the file. Its syntax is

```
rewind(stream);
```

For example, suppose a particular application program can put a password on a data file it uses. This password is stored in encrypted form on the first line of the file. The line is recognized as a password line if the first two characters are "*P". If the file has no password line, then access to the file is unrestricted. If a password line is found, the user is prompted for the password before access is permitted. The following code can be used to look for a password line:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   FILE *pswd;
   char line[256];

   if(argc != 2) {
      fprintf(stderr, "Usage: getpswd file\n");
      exit(1);
   }

   pswd = fopen(argv[1], "r");
   if(pswd == NULL) {
      fprintf(stderr, "Can't open %s.\n", argv[1]);
      exit(1);
   }

   fgets(line, 256, pswd);
   if(line[0] == '*' && line[1] == 'P') {

   /* ask for and check password */

   } else
      rewind(pswd);

      ...     /* application program goes here */

   exit(0);
}
```

If the first two characters of the first line are "*P", then code is executed which asks for and checks a password. However, if the first line is not a password line, the file is assumed to be unprotected, and the line just read is probably part of the data. Thus, the file must be rewound so the data contained in the first line is available to the application program.

The *ftell* routine returns a long integer specifying the current position of the next I/O operation on an open stream. This position is expressed as a byte offset relative to the beginning of the open file. Its syntax is as follows:

```
ftell(stream);
```

The *fseek* routine enables you to re-position the next I/O operation on an open stream to any location you wish. Its syntax is

```
fseek(stream, offset, ptrname);
```

where *stream* is a file pointer to the open stream, *offset* is a long integer specifying the number of bytes to skip over, and *ptrname* is an integer indicating the reference point in the file from which *offset* bytes are measured. The possible values for *ptrname* are:

0 move *offset* bytes from the beginning of the file;

1 move *offset* bytes from the current position in the file;

2 move *offset* bytes from the end of the file.

Offset can be either negative or positive, indicating backward or forward movement in the file, respectively.

The following program illustrates the use of the *ftell* and *fseek* library routines. The program prints each line of an *n*-line file in this order: line 1, line *n*, line 2, line $n-1$, line 3, ...

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
    char line[256];
    int newlines;
    long front, rear, ftell();
    FILE *fp;

    front = 0;
    rear = 0;

    if(argc < 2) {
        fprintf(stderr, "Usage:  print filename\n");
        exit(1);
    }

    fp = fopen(argv[1], "r");
    if(fp == NULL) {
        fprintf(stderr, "Can't open %s.\n", argv[1]);
        exit(1);
    }
    newlines = countnl(fp) % 2;

    fseek(fp, 0, 2);
    rear = ftell(fp);
```

```
    while(front < rear) {
          fseek(fp, front, 0);
          fgets(line, 256, fp);
          fputs(line, stdout);
          front = ftell(fp);
          findnl(fp, rear);
          rear = ftell(fp);
          if(newlines == 1) {
                if(rear <= front)
                      break;
          }
          fgets(line, 256, fp);
          fputs(line, stdout);
    }

    exit(0);
}
countnl(fp)
FILE *fp;
{
    char c;
    int count = 0;

    while((c = getc(fp)) != EOF) {

if(c == '\n')

    count++;
    }
    rewind(fp);
    return(count);
}

findnl(fp, offset)
FILE *fp;
long offset;
{
    char c;

    fseek(fp, (offset-2), 0);
    while((c = getc(fp)) != '\n') {

fseek(fp, -2, 1);
    }
}
```

This program uses *ftell* and *fseek* to print lines from a file starting at the beginning and the end of the file, and converging toward the center. The **countnl** (count new-lines) function counts the number of lines in the file so the program can decide whether or not to print a line in the final loop (this prevents the middle line being printed twice in files with an odd number of lines). The **findnl** (find new-line) function seeks backwards in the file for the next new-line. When found, this positions the next I/O operation such that *fgets* gets the next line back from the end of the file.

Note the use of *fseek* in this program. All three types of seeks are represented here. The first *fseek* of the program is done relative to the end of the file. All other *fseeks* in the main program are done relative to the beginning of the file. Finally, **findnl** contains an *fseek* which is relative to the current position.

Recall the employee data routine, where each employee is described by the structure

```
struct emp {
        char      name[40]; /* name */
        char      job[40];  /* job title */
        long      salary;   /* salary */
        char      hire[6];  /* hire date */
        char      curve[2]; /* pay curve */
        int       rank;     /* percentile ranking */
}
```

That routine simply read in the data for 400 employees all at once. Suppose you want the program to be selective, so that you can specify (by employee number, 1 – 400) *which* employee's information you want. This is easily done using *fseek*. The following program fragment shows how:

```
    ...

    int empno, bytes;
    long total;
    FILE *data;
    struct emp empinfo;

    /* check for usage error and open data file */
    ...

    sscanf(argv[1], "%d", &empno);
    bytes = sizeof(empinfo);
    total = (empno - 1) * bytes;
    fseek(data, total, 0);
    fread((char *)&empinfo, sizeof(empinfo), 1, data);

    /* print out desired information */
    ...

    exit(0);
}
```

In this program, argv[1] contains, via a command-line argument, the employee number about whom information is desired. This employee number is converted to integer form using *sscanf.* The number of bytes per employee structure is obtained using **sizeof**, and is stored in *bytes.* The total number of bytes to skip in the data file is found by multiplying the employee number (minus one) times the number of bytes per employee structure. This is stored in *total.* Then, *fseek* is used to seek past the specified number of bytes, relative to the beginning of the data file. This leaves the next I/O operation positioned at the start of the specified employee's information. The information is read using *fread.*

Note

If you have a stream which is open for both reading and writing, a read operation cannot be followed by a write operation without one of the following occurring first: a *rewind*, an *fseek*, or a read operation which encounters end-of-file. Similarly, a write operation cannot be followed by a read operation unless a *rewind* or *fseek* is performed.

Stream Control Routines

The routines described here help you control certain attributes of file pointers. The routines described are *fclose*, *setbuf*, *setvbuf*, *fflush*, and *freopen*.

fclose

You have already seen *fclose* in action in the previous example program which read an employee data file. *Fclose* flushes the buffer associated with the specified stream, and, if the buffer was allocated automatically by the standard I/O system, frees the space allocated to that buffer. The stream is then closed, breaking the connection between your file pointer and the stream.

You may be wondering why so many example programs have been written that open files but never explicitly close them. There are two reasons why this is permissible. First, you'll notice that all programs in this tutorial that open files end with a call to *exit*. The *exit* system call automatically performs an *fclose* for every open file in that process. Second, when a program is compiled with *cc* (or *fc*, or *pc*), an *exit* call is automatically compiled in with your code. Keep in mind, however, that it is generally bad programming practice to rely on the system to clean up after you! If you explicitly open any files, you should explicitly close them when you are done. If this is too much trouble, at least include an *exit* call at each termination point in the program. (All future example programs in this article will contain *fclose* calls.)

setbuf

Setbuf and *setvbuf* routines enable you to assign your own buffering to an open stream. *Setbuf* syntax is

```
setbuf(stream, buffer);
```

where *stream* is a file pointer to an already-open stream, and *buffer* is a pointer to a character array or is NULL.

Normally (i.e. without user intervention), a standard I/O buffer is obtained through a call to *malloc*(3C) (*memallc*(2) on the Series 500) upon the first call to *getc* or *putc* (which all I/O routines eventually call). The standard I/O system normally buffers I/O in a buffer which is BUFSIZ bytes long. Exceptions are *Stdout*, which, when directed to a terminal, is line-buffered, and **stderr**, which is normally unbuffered.

Setbuf enables you to change the buffer used for all standard I/O routines. For example, the following code fragment causes the array *buffer* to be used for buffering:

```
...

FILE *fp;
char buffer[BUFSIZ];

fp = fopen(argv[1], "r");
...

setbuf(fp, buffer);
...
```

This fragment shows the correct order of events. First, the file is opened (it need not be opened for reading), then the buffering is assigned using *setbuf.* From that point on, any input taken from *fp* is buffered through the array *buffer.*

Buffering can be eliminated altogether by specifying the NULL pointer in place of the buffer name, as in

```
setbuf(fp, NULL);
```

This causes input or output using *fp* to be completely unbuffered.

Setbuf is limited to buffer sizes of either BUFSIZ bytes or zero. *Setbuf* assumes that the character array pointed to by "buffer"is BUFSIZ bytes. Passing *setbuf* a (non-NULL) pointer to a smaller array can cause severe problems during operation because the standard I/O routines may overwrite memory following the end of the too-small buffer.

Note: Using an *automatic* array as a standard I/O buffer can be dangerous. Automatic variables are only defined in the code block in which they are declared. Thus, buffering which relies on an automatic array is only in effect during the current code block (main program or function). If you pass a file pointer to another function, and the stream pointed to by that file pointer is buffered using an automatic array, then memory faults or other errors can occur. Here's the rule: if you use an automatic array for stream buffering, the stream should be used and closed only in the code block containing the array declaration. To avoid this restriction, use **external** arrays for buffering:

```
...
external char buffer[BUFSIZ];
...
setbuf(fp, buffer);
```

setvbuf

Setvbuf, like *setbuf,* enables you to assign a character array for vuffering, but also provides the means to specify the size of the buffer to be used and the type of buffering to be done. *Setvbuf* syntax is

```
setvbuf(stream, buffer, type, size)
```

where *stream* is a file pointer to an already-open stream, *buffer* is a pointer to a character array or is NULL, *type* tells how *stream* is to be buffered, and *size* defines how large the *buffer* is. Acceptable values for *type* (defined in stdio.h) include:

–IOFBF	Input/output is fully buffered.
–IOLBF	Output is line buffered. The buffer is flushed each time a new line is written, the buffer is full, or input is requested.
–IONBF	Input/output is completely unbuffered.

If *type* –IONBF is specified, *stream* is totally unbuffered. Since no buffer is needed, values for *buffer* and *size* are ignored. For example, the following two calls, though different, are functionally identical:

```
setvbuf(fp, NULL, -IONBF, 0)
setbuf(fp, NULL)
```

When *type* is –IOFBF or –IOLBF, buffering for *stream* is determined by *buffer* and *size*. If *buffer* is not the NULL pointer, it must point to a character array of *size* bytes. All buffering of *stream* is then handled through this array.

```
          ...
          FILE *fp;
          char buffer [256]
          char *filename;
          int ..., retcode;
          fp=fopen(filename, "w");
retcode=setvbuf(fp, buffer, =IOFBF, 256);
          if (retcode !=0) error c);
```

This fragment causes stream *fp* to be buffered through the 256-byte array *buffer*. Serious run-time errors can occur if the buffer array is not the size specified in the call to *setvbuf* (here 256 bytes). As with *setbuf*, it is dangerous to use an automatic array for the buffer. Note that the return value of *setvbuf* can be used to verify that the request was completed successfully.

If *buffer* is the NULL pointer and type is specified as –IOFBF or –IOLBF, *setvbuf* automatically allocates a buffer of *size* bytes through a call to *malloc* (3c) on Series 200 computers or *memallc* (2) on Series 500 computers. If *size* is zero, a buffer of size BUFSIZ will be used. This behavior can be used to change the buffer size for a stream even if you still want the standard I/O system to automatically allocate the buffer. This is particularly useful when a buffer larger than the specified BUFSIZ is desired.

```
          ...
          FILE * fp;
          char * filename;
          int retcode;
          ...
          fp = fopen(filename, "rt")
retcode=setvbuf(fp, NULL, -IOFBF, 2048);
          if(retcode !=0) error( );
          ...
```

This fragment buffers stream *fp* through a 2048-byte buffer that is allocated by the system.

fflush

The *fflush* routine forces all buffered data for an output stream to be written out to that file. Its syntax is

```
fflush(stream);
```

where *stream* is a file pointer to an output stream.

Fflush is performed automatically by *fclose* (and, therefore, by *exit*). Therefore, there is often no reason to call *fflush* explicitly. Situations do arise, however, where it is necessary to manually *fflush* a stream. For example, data written to a terminal is line-buffered by default, which means that the system waits for a new-line before writing the buffer onto the terminal screen. This is often satisfactory, but there are times when you want whatever has been written so far to be written to the screen without waiting for the new-line. In such situations, *fflush* must be used.

Another situation when explicit *fflush*ing is necessary arises whenever you have written less than a buffer-full of data to a file, and you want the contents of that file processed by another function, or by an HP-UX command. Since less than a buffer-full of data was written, the data is still in the buffer; the file is still empty. Performing an *fflush* causes the buffered data to be written out to the file, enabling other functions or commands to utilize the file's contents.

freopen

The final routine in this section is *freopen*. As its name implies, *freopen* enables you to, in a single step, close a stream and then re-open it with a different type and/or file name. Its syntax is

```
freopen(filename, type, stream);
```

where *filename* is a pointer to a character string specifying the name of the source or destination file for the newly-created stream. *Type* is identical to that of *fopen* discussed earlier. *Stream* is a file pointer to the old stream, which is closed and then re-opened. The name of the file pointer remains the same.

For example, the following program accepts lines of data from your terminal and writes them into a file. When only a new-line is typed from the terminal, the program quits reading data, and echos the contents of the file to the terminal.

```
#include <stdio.h>
main()
{
  FILE *fp, *oldfp;
  char line[80], *fgets( );

  fp = fopen("datafile", "w");
  if(fp == NULL) {
     fprintf(stderr, "Can't create datafile.\n");
     exit(1);
  }

  fgets(line, 80, stdin);
  while(line[0] != "\n" {
     fputs(line, fp);
     fgets(line, 80, stdin);
  }

  oldfp = freopen("datafile", "r", fp);
  if(oldfp == NULL) {
     fprintf(stderr, "Can't re-open datafile.\n");
     exit(1);
  }
```

```
    while(fgets(line, 80, fp) != NULL)
        fputs(line, stdout);

    fclose(fp);
    exit(0);
}
```

Just like *fopen*, *freopen* returns a NULL pointer if an error occurs. If successful, *freopen* returns the value of the old file pointer.

Freopen is commonly used to attach the names **stdin**, **stdout**, and **stderr** to other files, so that the source or destination of these file pointers can be redirected. For example,

```
freopen("/usr/lib/data/datafile", "r", stdin);
```

attaches **stdin** to the data file */usr/lib/data/datafile*. Other functions can now be called which read from **stdin**, and the result is that their source of input has been redirected. Similarly,

```
freopen("/users/bill/archives/cal.a", "a", stdout);
```

attaches **stdout** to the indicated file, thus redirecting any future **stdout** data to that file.

Converting Between File Pointers and File Descriptors

A file pointer is actually a pointer to a structure containing information about a stream. This information includes a pointer to the beginning of the buffer, a pointer to the current location in the buffer, a flag specifying whether the stream is open for reading, writing, or both, a count of the characters in the buffer, and an integer called a *file descriptor*.

System calls, such as *open* and *creat*, return a file descriptor when a file is opened. System calls use file descriptors to refer to open files in much the same way that library routines use file pointers. (The main difference between using a file descriptor and using a file pointer is that a file descriptor has no associated buffering.) Since a program often contains both system calls and library routines, a way of converting between file pointers and file descriptors is provided.

Note

Extreme care should be exercised when converting between file pointers and file descriptors. Whenever you convert a file pointer to a file descriptor, you should perform an *fflush* first.

In general, you should never convert file pointers to file descriptors unless you need a file descriptor for a system call that provides a utility not available in the C library package (such as *dup*(2) or *fcntl*(2)). Similarly, file descriptors should never be converted to file pointers unless a file descriptor has been created by a system call which provides a utility not provided in the C library package, and you want to assign system buffering to it.

Two routines, *fileno* and *fdopen*, provide a way to convert between the two types of parameters. *Fileno* is a macro which, given a file pointer, returns the associated file descriptor. Its syntax is

```
fileno(stream);
```

where *stream* is a file pointer to an open stream whose associated file descriptor is desired. Thus,

```
...
FILE *fp;
int fd;
...
fp = fopen("file1", "r");
fd = fileno(fp);
```

returns the integer file descriptor in *fd*, associated with the file pointer *fp*.

The *fdopen* routine enables you to convert a file descriptor into a file pointer. Its syntax is

```
fdopen(fildes, type);
```

where *fildes* is an integer file descriptor obtained from the *open*, *dup*, *creat*, or *pipe* system calls. *Type* is the same as that for *fopen* discussed earlier. Thus,

```
...
int fd;
FILE *fp;
...
/* obtain fd via appropriate system call */
...
fp = fdopen(fd, "r");
if(fp == NULL) {
	fprintf(stderr, "Can't convert file descriptor.\n");
	exit(1);
}
...
```

converts the file descriptor *fd* into a file pointer, *fp*. *Fdopen* returns a NULL pointer if the operation fails.

Fdopen can be useful for opening a file in a way unlike any of the standard types of *fopen*.

```
...
# include <fcntl.h>
...
int fd;
FILE *fp
char *filename;
...
fd= open(filename, O_WRONLY|O_CREAT, 0666);
fp= fdopen(fd,"w");
fseek(fd,0L,2)
```

This code fragment uses the *open* system call to open a file for general write access, then uses *fdopen* to assign buffering to the file. The constants O_WRONLY and O_CREAT are defined in the include file */usr/include/fcntl.n*, and are described in *open* (2). (`O_WRONLY` causes *open* to open the file for writing only; `O-CREAT` creates the file if it does not already exist.) This technique opens the file in a way that does not correspond exactly to any of the available types in *fopen*: "w" would truncate the current file contents, "r+" would fail if the file does not already exist (and would allow reading of the file), and "a" does not permit seeking backwards and rewriting the current file contents.

Interprocess Communication

So far, you've been communicating between an active process (your program) and a passive object (a file). What if you want to communicate between two active processes? Suppose you want to create a stream between two programs, with one program (process) pumping data onto the stream, and the other reading data from the other end. How is this done?

The *popen* routine exists for this purpose. Its syntax is

```
popen(command, type);
```

where *command* is a pointer to a character string specifying a command line. *Type* is a pointer to a single-character string which is either **r** (for reading) or **w** (for writing).

For example, suppose you are writing a program which processes text in some way. Your program handles normal text perfectly, but unfortunately your source files are all coded in *troff* constructs. If you could only filter out all those pesky *troff* constructs, your program would work fine. Cheer up! It's easily done. There is an HP-UX command called *deroff* which filters out *troff* constructs. All you have to do is make sure that all input to your program passes through *deroff* first. Here's how:

```
#include <stdio.h>
main()
{
    FILE *popen(), *fp;

    fp = popen("deroff /users/bin/text/*.tx", "r");
    if(fp == NULL) {
        fprintf(stderr, "Can't create stream.\n");
        exit(1);
    }

    /* begin processing text; read text from fp! */
    ...
    pclose(fp);
```

Popen returns a file pointer to the newly-opened stream. If an error occurs, a NULL pointer is returned. When successfully executed, *popen* enables your program to read from the file pointer *fp*, the data from which is the standard output from the *deroff* command. In this example, *deroff* is invoked such that it processes all files in */users/bin/text* which end with ".tx". Note that *popen*'s return value must be declared explicitly because it is not declared in **stdio.h.**

Because *deroff* processes **stdin** if no arguments are given, the following *popen* call

```
fp = popen("deroff", "r");
```

enables your program to receive filtered text from **stdin** instead of from ordinary files. The result of executing the previous example is exactly the same as if you had typed

```
deroff /users/bin/text/*.tx | your program
```

at your keyboard in response to a shell prompt.

Streams that are opened by *popen* must be closed with *pclose*. Thus,

```
pclose(fp);
```

closes the stream created in the previous example.

If a *type* of **w** is specified instead of **r**, then the data flow is reversed, with the result that your program supplies the data for the specified *command*.

Note that, though *popen*'s return value is called a file pointer, it is actually somewhat different than the file pointers you are already familiar with. In general, a file pointer returned by *popen* should not be used in those previously-discussed library routines which modify file pointers returned by *fopen*. Also, file pointers opened by *popen* must be closed with *pclose*; *fclose* is not sufficient.

So far, *popen* has been characterized as a "filter-maker", in that streams to or from a command have been created so that data can be modified in some way before being passed on. Sometimes, however, *popen* is used to execute a command which supplies information valuable to the program. For example, the *find* command accepts dot (.) as a valid directory name. Upon receipt of a dot, *find* discovers the actual path name of dot by creating a stream from the *pwd* command, as follows:

```
char dir[100];
FILE *popen(), *fp;

fp = popen("pwd", "r");
if(fp == NULL) {
      fprintf(stderr, "Can't execute pwd.\n");
      exit(1);
}
fgets(dir, 100, fp);
...
pclose(fp);
```

The preceding example reads the output of the *pwd* command into the character array *dir*, thus supplying the current value of dot. The following program creates a list of the login names of users currently logged in:

```
#include <stdio.h>
main()
{
    char name[10], line[80], *fgets();
    FILE *popen(), *fp;

    fp = popen("who", "r");
    if(fp == NULL) {
        fprintf(stderr, "Can't execute who.\n");
        exit(1);
    }

    printf("Users currently logged in:\n");
    while(fgets(line, 80, fp) != NULL) {
        sscanf(line, "%s", name);
        printf("\t%s\n", name);
    }

    pclose(fp);
    exit(0);
}
```

A stream is created for reading from the *who* command. Each line from *who* is read, and the first field from each line is read and printed.

You may have only one *popen*-ed stream in a process at any given time.

Math Routines

Part 2

Described in this section are absolute value, power, square root, logarithmic, trigonometric, and other functions performing many different kinds of mathmatical calculations.

An include file named **math.h** exists for use with these routines. **Math.h** contains type declarations of all the math routines which do not return an **int**, and a definition of the constant **HUGE**. Many math routines return a "huge" value when an error occurs, so HUGE is set equal to this "huge" value, enabling you to check for errors easily. You need not include **math.h** in your program **if** you remember to explicitly declare each math routine's return type, and **if** you don't need HUGE.

Some of the math routines reside in the standard C library, */lib/libc.a*. This library also contains all the standard I/O routines and the system calls described in section 2 of the HP-UX Reference manual. This library is loaded automatically by the C compiler, *cc*, so you need not worry about explicitly telling the linker (*ld*) to search this library to find the functions contained in it. However, many math routines reside in the library */lib/libm.a*, which is *not* automatically loaded. Thus, if you try to compile a program containing a math routine from *libm.a*, you get a complaint from *ld*.

This is fixed in the following way. Suppose you have a program named *yourprog.c*, and this program contains a math function from *libm.a*. To compile the program, type

```
$ cc yourprog.c -lm
```

The **−l** option causes *ld* to look for and search a library named /lib/lib*x*.a, where *x* is the letter specified after the **−l** option. Thus, this command line tells *ld* to search /lib/libm.a.

How do you know which functions reside in which library? The HP-UX Reference manual provides guidance here. /lib/libc.a contains all of section 2, plus all routines in section 3 having the suffixes (3C) and (3S). /lib/libm.a contains all the routines in section 3 having the suffix (3M). To aid you in deciding how to compile your programs, the routines discussed below include references to the HP-UX Reference manual.

Absolute Value Functions

The *abs* (*abs*(3C)) and *fabs* (found under *floor*(3M)) functions return the absolute value of their integer or floating-point argument, respectively. For example, the following program calculates integer absolute values until a zero is entered from the keyboard:

```
main()
{
	int value;

	printf("Enter value: ");
	scanf("%d", &value);
	while(value != 0) {
		printf("Absolute value of %d is %d.\n", value, abs(value));
		printf("Enter value: ");
		scanf("%d", &value);
	}
	exit(0);
}
```

The floating-point equivalent of the previous program is shown below:

```
main()
{
	double value, fabs();

	printf("Enter value: ");
	scanf("%lf", &value);
	while(value != 0.0) {
		printf("Absolute value of %.12g is %.12g.\n", value, fabs(value));
		printf("Enter value: ");
		scanf("%lf", &value);
	}
	exit(0);
}
```

The first program above can be compiled without the **−l** option, but the second must be compiled using the **−lm** option.

Power, Square Root, and Logarithmic Functions

This section describes the following five functions, all of which are found under *exp*(3M) in the HP-UX Reference manual:

exp(x)	returns e to the *x* power.
log(x)	returns the natural logarithm of *x* (ln(x)).
log10(x)	returns the common logarithm of *x* (log(x)).
pow(x, y)	returns *x* to the *y* power.
sqrt(x)	returns the square root of *x*.

All functions return **double** values, and expect **double** arguments. Since their syntaxes are similar, the following logarithm calculator example suffices for all five of these functions:

```
#include <math.h>
main(argc, argv)
int argc;
char *argv[];
{
   double value;

   sscanf(argv[1], "%lf", &value);
   printf("Natural logarithm of %.12g = %.12g\n", value, log(value));
   printf("Common logarithm of %.12g = %.12g\n", value, log10(value));
}
```

This program accepts its single argument, and returns the natural and common logarithms of that argument.

All five of these functions must be compiled using the **−lm** option to *cc*.

Trigonometric Functions

A full set of trigonometric functions are provided in the math library. They are as follows:

sin(x)	returns the sine of the radian argument *x*.
cos(x)	returns the cosine of the radian argument *x*.
tan(x)	returns the tangent of the radian argument *x*.
asin(x)	returns the arc sine of *x* in the range -pi/2 to pi/2, where $-1 <= x <= 1$.
acos(x)	returns the arc cosine of *x* in the range 0 to pi, where $-1 <= x <= 1$.
atan(x)	returns the arc tangent of *x* in the range -pi/2 to pi/2.
atan2(y, x)	returns the arc tangent of *y/x* in the range -pi to pi.
sinh(x)	returns the hyperbolic sine of the radian argument *x*.
cosh(x)	returns the hyperbolic cosine of the radian argument *x*.
tanh(x)	returns the hyperbolic tangent of *x*.

The following program uses some of these routines, as well as two routines from the previous section, to obtain the dimensions and angles of a right triangle:

```
#include <stdio.h>
#include <math.h>
main()
{
    double sideA, sideB, sideC, anga, angb, tempC;
    double pi = fabs(acos(-1.));
    double torads = pi/180.;
    double todegs = 180./pi;
    double angc = 90.;

    printf("Using the following conventions for sides and angles:\n");
    triangle();
    printf("\nEnter all known information:\n");
    printf("\tA = ");
    scanf("%lf", &sideA);
    printf("\tB = ");
    scanf("%lf", &sideB);
    printf("\tC = ");
    scanf("%lf", &sideC);
    printf("\tAngle a = ");
    scanf("%lf", &anga);
    printf("\tAngle b = ");
    scanf("%lf", &angb);
    if(sideA && sideB && sideC) {
          tempC = sqrt(pow(sideA, 2.) + pow(sideB, 2.));
          if(fabs(sideC - tempC) > 0.001) {
                printf("Sides invalid.\n");
                exit(1);
          }
          anga = acos(sideB/sideC) * todegs;
          angb = 90. - anga;
```

```
} else if(sideA && sideB) {
    sideC = sqrt(pow(sideA, 2.) + pow(sideB, 2.));
    anga = acos(sideB/sideC) * todegs;
    angb = 90. - anga;
} else if(sideB && sideC) {
    sideA = sqrt(pow(sideC, 2.) - pow(sideB, 2.));
    anga = acos(sideB/sideC) * todegs;
    angb = 90. - anga;
} else if(sideA && sideC) {
    sideB = sqrt(pow(sideC, 2.) - pow(sideA, 2.));
    anga = acos(sideB/sideC) * todegs;
    angb = 90. - anga;
} else if(sideA) {
    if(anga && angb) {
        sideC = sideA/cos(angb*torads);
        sideB = sqrt(pow(sideC, 2.) - pow(sideA, 2.));
    } else if(anga) {
        sideC = sideA/sin(anga*torads);
        sideB = sqrt(pow(sideC, 2.) - pow(sideA, 2.));
        angb = 90. - anga;
    } else if(angb) {
        sideC = sideA/cos(angb*torads);
        sideB = sqrt(pow(sideC, 2.) - pow(sideA, 2.));
        anga = 90. - angb;
    } else {
        printf("Insufficient information.\n");
        exit(1);
    }
} else if(sideB) {
    if(anga && angb) {
        sideC = sideB/sin(angb*torads);
        sideA = sqrt(pow(sideC, 2.) - pow(sideB, 2.));
    } else if(anga) {
        sideC = sideB/cos(anga*torads);
        sideA = sqrt(pow(sideC, 2.) - pow(sideB, 2.));
        angb = 90. - anga;
    } else if(angb) {
        sideC = sideB/sin(angb*torads);
        sideA = sqrt(pow(sideC, 2.) - pow(sideB, 2.));
        anga = 90. - angb;
    } else {
        printf("Insufficient information.\n");
        exit(1);
    }
} else if(sideC) {
    if(anga && angb) {
        sideA = sideC * cos(angb*torads);
        sideB = sideC * sin(angb*torads);
    } else if(anga) {
        sideA = sideC * sin(anga*torads);
        sideB = sideC * cos(anga*torads);
        angb = 90. - anga;
    } else if(angb) {
        sideA = sideC * cos(angb*torads);
        sideB = sideC * sin(angb*torads);
        anga = 90. - angb;
```

```
        } else {
             printf("Insufficient information.\n");
             exit(1);
        }
   } else {
        printf("Insufficient information.\n");
        exit(1);
   }

   printf("\n\tSide A = %.2f\t\tAngle a = %.2f degrees\n", sideA, anga);
   printf("\tSide B = %.2f\t\tAngle b = %.2f degrees\n", sideB, angb);
   printf("\tSide C = %.2f\n", sideC);
}
triangle()
{
   FILE *fopen(), *tri;
   char line[50], *fgets();

   tri = fopen("triangle", "r");
   if(tri == NULL) {
        printf("Cannot open triangle file.\n");
        exit(1);
   }

   while(fgets(line, 50, tri) != NULL)
        fputs(line, stdout);
   fclose(tri);
}
```

The *triangle* function prints out the contents of a file in the current directory called **triangle**. The contents of this file should contain an ASCII approximation of a right triangle:

```
          /|
         / |
        /  |
       / a |
      /    |
   C /     | B
    /      |
   /       |
  /        |
 / b    c _|
/________|_|
      A
```

This triangle, made up of slashes, vertical bars, and underscores, shows the naming convention for the sides and angles. The program then asks for the known data; enter a value of zero for those parameters that are unknown. The dimensions and angles are then calculated based on the data you have supplied. If there is insufficient information, you are told about it.

The hyperbolic functions are found under *sinh*(3M) in the HP-UX Reference manual. All others are found under *trig*(3M). Thus, the **−lm** argument must be used when compiling code containing these functions.

Miscellaneous Functions

Calculating Upper and Lower Bounds

Two functions, *floor* and *ceil* (see *floor*(3M)), enable you to obtain integers (returned as **doubles**) defining an upper and a lower bound for a number or a series of numbers. *Floor* returns a double precision representation of the the largest integer which is still not greater than *floor*'s argument. Similarly, *ceil* returns a double precision representation of the smallest integer which is still greater than *ceil*'s argument.

The following program returns the floor and ceiling values for the number specified as its argument:

```
#include <math.h>
main(argc, argv)
int argc;
char *argv[];
{
   double value;

   sscanf(argv[1], "%lf", &value);
   printf("Floor = %g; Ceiling = %g\n", floor(value), ceil(value));
}
```

If you type this in and run it, you see that *floor* and *ceil* provide two **double** values representing the smallest range in which the numbers used to obtain that range will fit. For example, if you have a program which reads three values from a source file, and these values are 4.79, 19.6, and 21.1, you can get the smallest possible range in which these numbers fit by running *floor* on each number (and keeping the smallest floor value), and then running *ceil* on each number (and keeping the largest ceiling value). For the above three numbers, this yields a floor value of 4, and a ceiling value of 22.

Code containing these functions must be compiled using the **−lm** *cc* option. **Math.h** need not be included if you remember to explicitly declare that these functions return **double** values.

Calculating Remainders

This section covers two functions, *fmod* and *modf*. The *fmod* function (see *floor*(3M)) returns the remainder (in double precision form) resulting from dividing *fmod*'s first argument by its second. For example,

```
fmod(10., 4.)
```

divides 10 by 4, and returns the remainder (2, in this case). The following program accepts two numbers, divides the first by the second, and displays the results in a form showing the number of times the divisor goes evenly into the dividend, and the remainder, if any.

```
#include <math.h>
main(argc, argv)
int argc;
char *argv[];
{
    int result;
    double number, div, rem;

    sscanf(argv[1], "%lf", &number);
    sscanf(argv[3], "%lf", &div);

    result = number/div;
    printf("%g = (%d)(%g)", number, result, div);
    if((rem = fmod(number, div)) != 0.0)

printf(" + %g\n", rem);
}
```

This program is set up so that it can be invoked in sentence style. If you name the compiled version of this program "divide", then you can say

```
$ divide 33.27 by 11
```

Since argv[2] is ignored in the code, "by" is harmless, and the two numbers are parsed correctly.

Code containing a call to *fmod* must be compiled with the **−lm** *cc* option. However, you need not include **math.h** in your program, as long as you declare *fmod*'s return type appropriately.

The other function, *modf* (see *frexp*(3C)), is not really a remainder function in the same sense that *fmod* is a remainder function. In *fmod*, a division actually takes place. In *modf*, however, no division takes place. *Modf* simply accepts a **double** value, and splits it into its integer and fractional parts. Its syntax is

```
modf(value, iptr);
```

where *value* is the number to be split into two parts, and *iptr* is a pointer to a **double** variable where the integer part of *value* is to be stored. *Modf*'s return value is the signed fractional part of *value.*

The following program shows *modf* in action:

```
main(argc, argv)
int argc;
char *argv[];
{
    double value, iptr, frac, modf();

    sscanf(argv[1], "%lf", &value);
    frac = modf(value, &iptr);
    printf("Integer part: %g; Fractional part: %g\n", iptr, frac);
}
```

The program accepts one argument, the value, and then prints the integer and fractional parts of that value. Note that the address of *iptr* is passed to *modf*, because *modf* expects the address of a **double** variable where the integer part can be stored.

Code containing calls to *modf* does not require the **−lm** option during compilation. Also, the **math.h** include file is of no use to *modf*, so it can be omitted.

Calculating A Hypotenuse

The *hypot* function (see *hypot*(3M)) returns the square root of the sum of the squares of its two arguments, yielding the length of the hypotenuse of a right triangle, or the Euclidian Distance.

Thus, in the previous program which calculated the sides and angles of a right triangle, the line of code which read

```
sideC = sqrt(pow(sideA, 2.) + pow(sideB, 2.));
```

could be replaced with

```
sideC = hypot(sideA, sideB);
```

thus eliminating one function call (*hypot* contains a call to *sqrt*).

Code containing calls to *hypot* must be compiled using the **−lm** option to *cc*.

Generating Random Numbers

The *rand* and *srand* routines (see *rand*(3C)) exist for the generation of random numbers. *Rand* is the random number generator itself, and *srand* enables you to specify a starting point (or *seed*) for *rand*.

The following program simply sets up an infinite loop and lets *rand* run for awhile (to terminate it, just press BREAK, or its equivalent):

```
main()
{
   unsigned value;

   srand(1);
   for(;;) {
        value = rand();
        printf("Random number is %u\n", value);
        sleep(1);
   }
}
```

Note that *rand* and *srand* deal only with *unsigned* integers. If you let this program run for awhile, you'll notice that the random values returned are quite large, and don't often venture below 1000. If your application requires smaller random numbers, divide the value returned by *rand* by some appropriate divisor until a number in the desired range is obtained.

Srand initializes the random number generator to a particular starting point. In the above program, 1 is used, but you can specify any positive integer you like.

The *sleep* library routine causes the program to "pause" for the number of seconds specified (1, in this case).

Floating-Point Exponentiation Routines

Two routines, *frexp* and *ldexp* (see *frexp*(3C)), are covered in this section. *Frexp* accepts a **double** value, and returns two values, *x* and *n*, such that

```
value = x * 2^n
```

where *x* is a **double** quantity of magnitude less than 1, and *n* is an integer exponent. *Frexp*'s syntax is

```
frexp(value, eptr);
```

where *value* is the value to be processed, and *eptr* is a pointer to an integer variable where the exponent *n* is to be stored. The quantity *x* is returned as *frexp*'s return value.

The following program accepts a number argument and uses *frexp* to output that number's representation in the form shown above:

```
main(argc, argv)
int argc;
char *argv[];
{
   double value, x, frexp();
   int eptr;

   sscanf(argv[1], "%lf", &value);
   x = frexp(value, &eptr);
   printf("%g = %g * 2^%d\n", value, x, eptr);
}
```

Ldexp accepts a **double** *value* and an integer exponent *exp*, and returns a **double** quantity equal to

<value> * 2^ <exponent>

The following program accepts two number arguments, *value* and *exp*, and outputs the result:

```
main(argc, argv)
int argc;
char *argv[];
{
   double value, result, ldexp();
   int exp;

   sscanf(argv[1], "%lf", &value);
   sscanf(argv[2], "%d", &exp);
   result = ldexp(value, exp);
   printf("%g * 2^%d = %g\n", value, exp, result);
}
```

Neither of these routines require **math.h** or the use of the **−lm** *cc* option.

Character Conversion and Classification

Part 3

This section discusses those routines found under *conv*(3C) and *ctype*(3C) which enable you to convert between upper- and lower-case, and classify characters as digits, non-printing, upper-case, etc.

Converting Between Uppercase and Lowercase

Four routines are documented under *conv*(3C) which enable you to convert between upper- and lowercase. They are *toupper*, *tolower*, *_toupper*, and *_tolower*.

Toupper and *tolower* are functions which accept a single integer argument in the range −1 through 255. If the integer taken as a character represents a lower-case character, *toupper* returns the corresponding upper-case character. Similarly, *tolower* returns the corresponding lower-case character. Both routines return the argument unchanged if it does not represent a lower-case character (*toupper*) or an upper-case character (*tolower*).

_toupper and *_tolower* are macros defined in **ctype.h**. *_toupper* accepts a single integer argument which *must* represent a lower-case character; the corresponding upper-case character is returned. Similarly, *_tolower must* be given an upper-case character, and returns the corresponding lower-case character. If an argument is specified which is not a lower-case character (*_toupper*) or an upper-case character (*_tolower*), garbage is returned.

The macro versions of these routines are faster than the functions, so if you can guarantee that only lower-case or upper-case characters are passed to the macros, you should probably use them. However, the function versions are handy for tasks like

```
...
for(i=0; array[i] != NULL; i++)
     array[i] = toupper(array[i]);
...
```

which converts every lowercase character found in *array* to uppercase. The functions enable you to be more lenient about the arguments passed to them. In the above program fragment, no argument checking is needed; if the argument isn't a lowercase character, it is returned unchanged.

Character Classification

The *ctype*(3C) entry in the HP-UX Reference lists routines which test their single argument and return a non-zero value if the test is positive, and 0 otherwise.

All of these routines are macros defined in **ctype.h**. Because their syntaxes are identical, the following example suffices for all *ctype* macros:

```
...
for(i=0; array[i] != NULL; i++) {
     if(islower(array[i]))
          array[i] = _toupper(array[i]);
}
...
```

This program fragment shows one way to change all occurrences of a lower-case character in *array* to upper-case using the macro *_toupper*. The macro *islower* is used to make sure that only lower-case characters are passed to *_toupper*.

String Manipulation

String(3C) in the HP-UX Reference manual documents an extensive list of string manipulation routines enabling you to perform several operations on character strings. This section describes the *string*(3C) package in detail.

Concatenating Strings

Strcat and *strncat* enable you to append a copy of one string onto the end of another. Their syntaxes are:

```
strcat(s1, s2);
strncat(s1, s2, n);
```

where *s1* and *s2* are character pointers to NULL-terminated character strings. *Strcat* appends the entire string pointed to by *s2* (up to the first NULL character encountered) onto the end of string *s1*. *Strncat* does the same thing, except that at most *n* characters are appended to *s1* (or up to a NULL character, whichever comes first). (Note that string *s2* need not be NULL-terminated when using *strncat* if *n* is less than or equal to the length of *s2*.) Both routines return a character pointer to the NULL-terminated result.

Neither of these routines checks to make sure that there is room in *s1* for the additional characters of *s2*. Thus, to be safe, *s1* should **always** be a declared array having plenty of space for the additional characters of *s2*, plus a terminating NULL character.

Copying Strings

Strcpy and *strncpy* copy one string of characters into another. Their syntaxes are:

```
strcpy(s1, s2);
strncpy(s1, s2, n);
```

where *s2* is a character pointer to the string to be copied, and *s1* is a character pointer to the beginning of the string into which the contents of string *s1* are copied. *Strcpy* copies the entire string, up to (and including) the first NULL encountered. *Strncpy* copies up to *n* characters, or up to (and including) the first encountered NULL, whichever occurs first. (String *s2* need not be NULL-terminated when using *strncpy* if *n* is less than or equal to the length of *s2*.) Both routines return the value of *s1*.

The following program uses the *strcat* routine discussed earlier and *strcpy* to build a character string representing the lower-case alphabet, one character at a time.

```
#include <stdio.h>
main()
{
    int b = 'b', z = 'z', i;
    char alpha[30], chr[4];

    chr[1] = NULL;
    strcpy(alpha, "a");
    printf("%s\n", alpha);

    for(i = b; i <= z; i++) {
          chr[0] = i;
          strcat(alpha, chr);
          printf("%s\n", alpha);
    }
}
```

The array *chr* is always going to be a two-character array consisting of the next character in the alphabet followed by NULL. Thus, the second element of *chr* is set to NULL early in the program. The first *chr* element is then successively set to the next lower-case character in the **for** loop, and the resulting two-character string is concatenated onto the end of the alphabet assembled so far in *alpha*. Note the use of *strcpy* to initialize *alpha*. Remember that C transforms one or more characters enclosed in double quotes into a character pointer to those characters followed by a NULL. Thus, the *strcpy* statement above copies the character "a" followed by a NULL character into *alpha*.

There are some things to be aware of when using *strcat*, *strncat*, *strcpy*, and *strncpy*. These routines all modify string *s1* in some way, but none of them check for overflow in that string. Therefore, be sure there is enough room in *s1* to hold the added or copied characters plus a terminating NULL. Also, be sure you use a character *array* for *s1* (*not* just a character pointer), especially when using *strcat* or *strncat*. This is because an explicitly-declared array has sufficient memory allocated to it to contain all of its elements, but a character pointer simply points to a single location in memory. Concatenating a string to the end of a string contained in an array is guaranteed to work, provided the array is large enough. However, concatenating a string to a string of characters referenced by a simple character pointer is dangerous, since the concatenated characters could overwrite data in memory. For example,

```
char array[100], *ptr = "abcdef";
...
strcat(array, ptr);
```

works fine, since you are guaranteed that 100 storage elements have been set aside for the array. However,

```
char *ptr1 = "abcdef", *ptr2 = "ghijkl";
...
strcat(ptr1, ptr2);
```

is asking for trouble. Although C makes sure that there is enough room for the initializing strings ("abcdef" and "ghijkl" in this example), there are no guarantees that there is enough room to add characters to the end of one of these strings. Therefore, the last fragment could easily overwrite valid data occurring after the string pointed to by *ptr1*.

Since string *s2* is not modified, you can use arrays or character pointers with no ill effects.

Comparing Strings

Strcmp and *strncmp* compare two strings and return an integer indicating the result of the comparison. Their syntaxes are:

```
strcmp(s1, s2);
strncmp(s1, s2, n);
```

where *s1* and *s2* are character pointers to the NULL-terminated character strings to be compared. *Strcmp* compares the entire strings, stopping as soon as the result is determined. *Strncmp* compares at most *n* characters of both strings (neither string need be NULL-terminated if *n* is less than or equal to the length of the shorter string). The integer returned uses the following convention:

<0	*s1* is lexicographically less than *s2*;
=0	*s1* and *s2* are equal;
>0	*s1* is lexicographically greater than *s2*.

The following program fragment uses *strncmp* to analyze the contents of a file coded with the man macros (see *man*(7)). It reads each line of the file and keeps a count of the number of times selected macros are used, and prints a summary of its findings at the end.

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
    char *fgets(), line[100];
    FILE *fp;
    int nsh, npp, ntp, nrs, nre, npd, nip, nmisc, nlines;

    nsh = npp = ntp = nrs = nre = npd = nip = nmisc = nlines = 0;

    if(argc != 2) {
          fprintf(stderr, "Usage: count file\n");
          exit(2);
    }

    fp = fopen(argv[1], "r");
    if(fp == NULL) {
          fprintf(stderr, "Can't open %s.\n", argv[1]);
          exit(1);
    }

    while(fgets(line, 100, fp) != NULL) {
          if(strncmp(line, ".SH", 3) == 0)
                nsh++;
          else if(strncmp(line, ".PP", 3) == 0)
                npp++;
          else if(strncmp(line, ".TP", 3) == 0)
                ntp++;
          else if(strncmp(line, ".RS", 3) == 0)
                nrs++;
          else if(strncmp(line, ".RE", 3) == 0)
                nre++;
```

```
        else if(strncmp(line, ".PD", 3) == 0)
             npd++;
        else if(strncmp(line, ".IP", 3) == 0)
             nip++;
        else if(line[0] == '.')
             nmisc++;
        nlines++;
    }
    printf("No. of lines: %d\n\n", nlines);
    printf("No. of .SH's: %d\n", nsh);
    printf("No. of .PP's: %d\n", npp);
    printf("No. of .TP's: %d\n", ntp);
    printf("No. of .RS's: %d\n", nrs);
    printf("No. of .RE's: %d\n", nre);
    printf("No. of .PD's: %d\n", npd);
    printf("No. of .IP's: %d\n", nip);
    printf("No. of misc. macros: %d\n", nmisc);

    fclose(fp);
    exit(0);
}
```

In the above program, *strncmp* is used to compare the first three characters of each line read. If the first three characters match a particular macro, the appropriate counter is incremented. If the line begins with ".", but is not one of the macros being searched for, the "miscellaneous" counter is incremented. The total number of lines in the file is also given.

Finding the Length of a String

The *strlen* routine returns an integer specifying the number of non-NULL characters in a string. Its syntax is:

```
strlen(s);
```

where *s* is a character pointer to the NULL-terminated string whose length is to be taken. For example, if you execute

```
len = strlen(string);
```

then the integer *len* contains the total number of non-\s-1NULL\s+1 characters in the string pointed to by *string*. Thus,

```
string[len]
```

points to the terminating NULL in *string*

Finding Characters in Strings

The *strchr*, *strrchr*, and *strpbrk* routines enable you to locate a particular character within a string.

Strchr and *strrchr* return a character pointer to an occurrence of a specified character in a string. Their syntaxes are:

```
strchr(s, c);
strrchr(s, c);
```

where *s* is a character pointer to the string of interest, and *c* is a variable of type **char** specifying the character to search for.

Strchr returns a character pointer to the first occurrence of character *c* in string *s*. Similarly, *strrchr* returns a character pointer to the last occurrence in string *s*. Both routines return a NULL if the character does not occur in the string pointed to by *s*. For example,

```
char *ptr, *strchr(), string[100];
...
while((ptr = strchr(string, '@') != NULL)
   *ptr = '#';
```

replaces all occurrences of "@" in the array *string* with "#", starting from the beginning of the array and working toward the end. The same operation can be done using

```
while((ptr = strrchr(string, '@')) != NULL)
   *ptr = '#';
```

which replaces all @'s with #'s, starting from the end of the array, working backward toward the beginning.

The *strpbrk* routine returns a character pointer to the first occurrence in string *s1* of any character contained in string *s2*, or NULL if none of the characters in *s2* occur in *s1*. Its syntax is:

```
strpbrk(s1, s2);
```

For example, suppose you have to read lines of input in which are embedded numerical data which must be read. For simplicity, assume that the following conventions are used:

- Positive numbers do not begin with "+";
- Fractional numbers always begin with zero, as in 0.25;
- The first occurrence of a digit in the string signals the beginning of the number to be read.

Given these rules, the following code fragment does the job:

```
char line[100], *chrs = "-0123456789", *ptr;
float value;
...
ptr = strpbrk(line, chrs);
sscanf(ptr, "%f", &value);
...
```

The character pointer *chrs* is initialized to point to a string of characters which might introduce the embedded number. *Strpbrk* then finds the first occurrence of one of these characters in *line*, and returns a pointer to that location in *ptr*. Finally, *ptr* is passed to *sscanf*, which interprets *ptr* as if it were a pointer to the beginning of a string from which input is to be taken. The number is read correctly because *ptr* points to the beginning of a number, and because the %f conversion terminates at the first inappropriate character.

Miscellaneous String Routines

Finding Characters Common to Two Strings

The *strspn* and *strcspn* routines return an integer giving the length of the initial segment of string *s1* which consists entirely of characters found in string *s2*. *Strcspn* is similar, but returns an integer giving the length of the initial segment of *s1* which consists entirely of characters *not* found in string *s2*. Their syntaxes are:

```
strspn(s1, s2);
strcspn(s1, s2);
```

For example, suppose you have the following two strings:

```
"A tattle-tale never wins."
```

for string *s1*, and

```
" -Aatle"
```

for *s2*. Executing

```
strspn(s1, s2);
```

with the strings shown returns a value of 14, since the first 14 characters in *s1* all occur in *s2* – "A tattle-tale ". If you execute

```
strcspn(s1, s2);
```

using the same strings, you get 0, because there is no initial segment of *s1* which contains characters not found in *s2*.

Breaking a String into Tokens

A *token* is a string of characters delimited by one or more token delimiters. The *strtok* routine divides string *s1* into one or more tokens. The token separators consist of any characters contained in string *s2*. Its syntax is:

```
strtok(s1, s2);
```

where *s1* is a character pointer to the string which is to be broken up into tokens, and *s2* is a character pointer to a string consisting of those characters which are to be treated as token separators.

Strtok returns the next token from *s1* each time it is called. The first time *strtok* is called, both *s1* and *s2* must be specified. On subsequent calls, however, *s1* need not be specified (a NULL is specified in its place). *Strtok* remembers the string from call to call. String *s2* must be specified each call, but need not contain the same characters (token separators) each time.

Strtok returns a pointer to the beginning of the next token, and writes a NULL character into *s1* immediately following the end of the returned token. *Strtok* returns a NULL when no tokens remain.

For example, suppose you are reading lines from */etc/gettydefs*, which is the speed table for *getty*(8) – see *gettydefs*(5). The lines in this file contain several fields delimited by pound signs (#). Thus, the following code could be used to read the fields of each line:

```
int count = 0;
char *delims = "#", *token, *arg1, *strtok(), line[256];
arg1 = line;
...
while((token = strtok(arg1, delims) != NULL) {
   count++;
   printf("field %d: %s\n", count, token);
   if(count == 1)
      arg1 = NULL;
}
```

This code sees to it that *strtok*'s first argument is NULL after the first call. Also, note that *delims* did not change from call to call, but it could have. This greatly increases the power of *strtok*, since it enables you to change the token delimiters between calls.

Date and Time Manipulation

Part 4

Ctime(3C) describes a set of routines which enable you to access the date and time as maintained by the system clock. This package knows about daylight saving time, and automatically converts between standard time and daylight saving time when appropriate.

Most of the *ctime* routines require the quantity returned by *time*(2), which is the number of seconds that have elapsed since 00:00:00 GMT (Greenwich Mean Time), January 1, 1970.

The *ctime* routine converts the *time*(2) value into a 26-character ASCII string of the form

```
Fri May 11 09:53:03 1984\n\0
```

where "\n" is a new-line character, and "\0" is a terminating NULL character. *Ctime*'s syntax is:

```
ctime(value);
```

where *value* is a pointer to a long integer value representing the number of elapsed seconds since 00:00:00 GMT, January 1, 1970 (as returned by *time*(2)). Note that *value* is a pointer to the quantity returned by *time*(2), not just the quantity itself. Using *time*(2) and *ctime*, you can write your own simplified version of the *date*(1) command:

```
#include <stdio.h>
main()
{
    char *str, *ctime();
    long time(), nseconds;

    nseconds = time((long *)0);
    str = ctime(&nseconds);
    printf("%s", str);
}
```

The rest of the routines in *ctime*(3C) require the include file *time.h*, which contains the definition of a structure called *tm*. This structure is made up of several variables which contain the various components of the date and time. It looks as follows:

```
struct tm {
        int  tm_sec;
        int  tm_min;
        int  tm_hour;
        int  tm_mday;
        int  tm_mon;
        int  tm_year;
        int  tm_wday;
        int  tm_yday;
        int  tm_isdst;
};
```

The meaning associated with each structure member is:

tm_sec the "seconds" portion of the system's 24-hour clock time;

tm_min the "minutes" portion of the system's 24-hour clock time;

tm_hour the "hours" portion of the system's 24-hour clock time;

tm_mdaythe day of the month, in the range 1 thru 31;

tm_mon the month of the year, in the range 0 thru 11 (0 = January);

tm_year the current year − 1900;

tm_wdaythe day of the week, in the range 0 thru 6 (0 = Sunday);

tm_yday the day of the year, in the range 0 thru 365;

tm_isdst a flag which is non-zero if daylight saving time is in effect.

The *localtime* and *gmtime* routines accept a pointer to a quantity such as returned by *time*(2), and fill in the various components of the *tm* structure. *Localtime* corrects the time for the local time zone and possible daylight saving time, while *gmtime* converts directly to GMT time (this is the time used by HP-UX). Both routines return a pointer to a structure of type *tm* which can be used to access the various components of the *tm* structure.

For example, the following code fragment assigns values to the *tm* structure members for the local time zone:

```
#include <time.h>
...
struct tm *ptr, *localtime();
long time(), nseconds;
...
nseconds = time((long *)0);
ptr = localtime(&nseconds);
```

Once this code is executed, you can use *ptr* to access the different components of the local time. For example, **ptr−>tm_mon** references the month of the year, and **ptr−>tm_wday** references the day of the week. (*Gmtime* is used in exactly the same way, so this example suffices for it also).

The *asctime* routine converts the time contained in a *tm* structure into \s-1ASCII\s+1 representation such as that returned by *date*(1) and *ctime*. Its syntax is:

```
asctime(ptr);
```

where *ptr* is a pointer to a structure of type *tm* whose members have previously been assigned values with *localtime* or *gmtime*, or explicitly by you. *Asctime* returns a character pointer to the same NULL-terminated 26-character string as returned by *ctime*.

Asctime provides a way for you to obtain the current time, modify it explicitly in some way, and then print the result in ASCII form. The *date* command shown earlier can be re-written using *localtime* and *asctime*:

```
#include <stdio.h>
#include <time.h>
main()
{
    long time(), nseconds;
    struct tm *ptr, *localtime();
    char *string, *asctime();

    nseconds = time((long *)0);
    ptr = localtime(&nseconds);

/* the user may modify the current time in tm here */

    string = asctime(ptr);
    printf("%s", string);
}
```

This program illustrates a rather indirect way to obtain the date, but it does enable you to modify the date stored in *tm* before you print it out. If all you want to do is print the date, the quickest way is to use the *time*/*ctime* combination.

Of all the *ctime* routines, perhaps the most useful is *localtime*. It enables you to break the current time up into referencable chunks which can then be examined for such applications as personal calendar programs, program schedulers, etc. Many of the *tm* values can be used as indices into arrays containing strings identifying months and days. For example, declaring an external array like

```
char *month[] = { "January", "February", "March", "April",
                  "May", "June", "July", "August", "September",
                  "October", "November", "December"
                };
```

enables you to use **tm_mon** as an index into this array to obtain the actual month name. The same thing can be done with **tm_wday** if you initialize an array containing the names of the days of the week. The *ctime*(3C) package makes it easy to design programs which depend upon the time or date. Try creating your own versions of *calendar*(1), *at*(1), or even *cron*(8)!

Table of Contents

Lint C Program Checker
Introduction....1
Error Detection....1
Problem Detection....2
Problem Code: Unused Variables and Functions....2
Suppressing Lint....3
Problem Code: Set/Used Information....3
Problem Code: Unreachable Code....4
Suppressing Lint....4
Problem Code: Function Value....4
Problem Code: Type Matching....5
Suppressing Lint....6
Problem Code: Portability....6
Suppressing Lint....7
Problem Code: Strange Constructions....7
Suppressing Lint....8
Problem Code: Obsolete Constructions....8
How to Use Lint....9
Directives....9
Option List....10

Lint
C Program Checker

Introduction

Lint is a program checker and verifier for C source code. Its main purpose is to supply the programmer with warning messages about problems with the source code's style, efficiency, portability, and consistency. Once the C code passes through the compiler with no errors, *lint* can be used to locate areas, undetected by the compiler, that may require corrections.

Error messages and *lint* warnings are sent to the standard error file (the terminal by default). Once the code errors are corrected, the C source file(s) should be run through the C compiler to produce the necessary object code.

Error Detection

Lint can detect all of the code errors that the C compiler detects. An example of an error message would be:

    illegal initialization

These errors must be corrected before the compiler can be used to produce object code.

Although *lint* can be used for error detection, it cannot recover from all of the code errors it finds. If *lint* encounters an error that it can not recover from, it sends the message:

    cannot recover from earlier errors – goodbye!

and then terminates.

Lint limits the number of code errors that it detects to 30. Once 30 errors have been found in the source file(s), any additional error causes the message:

    too many errors

to be sent to the standard error file, and *lint* terminates. Because of this limitation and *lint*'s inability to recover from some errors, the compiler should be used for error detection. Once the error-causing code has been corrected, *lint* can be used on the source code for finding some of its inefficiencies and bugs.

Problem Detection

The main purpose of *lint* is to find problem areas in C source code. The detected code may not be considered an error by the C compiler; it can be converted into object code. However, *lint* considers the code to be inefficient, nonportable, bad style, or a possible bug.

Comments about problems that are local to a function are produced when they are detected. They have the form:

warning: <message text>

Information about external functions and variables is collected and analyzed after *lint* has processed the files handed to it. At that time, if a problem has been detected, it sends a warning message with the form:

<message text>

followed by a list of external names causing the message and the files where the problem occurred.

Code causing *lint* to issue a warning message should be analyzed to determine the source of the problem. Sometimes the programmer has a valid reason for writing the problem code. Usually, though, this is not the case. *Lint* can be very helpful in uncovering subtle programming errors.

Lint checks the source code for certain conditions, about which it issues warning messages. These can be grouped into the following categories:

1. variable or function is declared but not used;
2. variable is used before it is set;
3. portion of code is unreachable;
4. function values are used incorrectly;
5. type matching does not adhere strictly to C rules;
6. code has portability problems;
7. code construction is strange;
8. code construction is obsolete.

The code that you write may have constructions in it that *lint* objects to but that are necessary to its application. Warning messages about problem areas that you know about and do not plan to correct provide useless information and make helpful messages harder to find. There are two methods for suppressing warning messages from *lint* that you do not need to see. The use of *lint options* is one. The *lint* command can be called with any combination of its defined option set. Each option has *lint* ignore a different problem area. The other method is to insert *lint directives* into the source code. *Lint* directives are discussed later.

Problem Code: Unused Variables and Functions

Lint objects if source code declares a variable that is never used or defines a function that is never called. Unused variables and functions are considered bad style because their declarations clutter the code. They can also be the cause of a program bug if their use is essential.

An unused local variable can result in one of two *lint* warning messages. If a variable is defined to be static and is not used *lint* responds with:

warning: static variable <name> unused

Unused automatic variables cause the message:

warning: <name> unused in function <name>

A function or external variable that is unused causes the message:

name defined but never used

followed by the function or variable name and the file in which it was defined. *Lint* also looks at the special case where one of the parameters of a function is not used. The warning message is:

warning: argument unused in function: <arg_name> in <func_name>

If functions or external variables are declared but never used or defined *lint* responds with

name declared but never used or defined

followed by a list of variable and function names and the names of files where they were declared.

Suppressing Lint

Sometimes it is necessary to have unused function parameters to support consistent interfaces between functions. The **–v** option can be used with *lint* to have warnings about unused parameters suppressed. However, the **–v** option does not suppress comments when parameters are defined as register variables. Unused register variables result in an inefficient use of the computer's resources, since quick-access hardware is often allocated for their storage.

If *lint* is run on a file which is linked with other files at compile time, many external variables and functions can be defined but not used, as well used but not defined. If there is no guarantee that the definition of an external object is always seen before the object is used, it is declared **extern**. The **–u** option can be used to stop complaints about all external objects, whether or not they are declared **extern**. If you want to inhibit complaints about only the **extern** declared functions and variables, use the **–x** option.

Problem Code: Set/Used Information

A probable bug exists in a program if a variable's value is used before it is assigned. Although *lint* attempts to detect occurrences of this, it takes into account only the physical location of the code. If code using a static or external variable is located before the variable is given a value the message sent is:

warning: <name> may be used before set

Since static and external variables are always initialized to zero this may not point out a program bug. *Lint* also objects if automatic variables are set in a function but not used. The message given is:

warning: <name> set but not used in function

Problem Code: Unreachable Code

Lint checks for three types of unreachable code. Any statement following a **goto**, **break**, **continue**, or **return** statement must either be labeled or reside in an outer block for *lint* to consider it reachable. If neither is the case, *lint* responds with:

warning: statement not reached

The same message is given if *lint* finds an infinite loop. It only checks for the infinite loop cases of **while(1)** and **for(;;)**. The third item that *lint* looks for is a loop that cannot be entered from the top. If one is found then the message sent is:

warning: loop not entered from top

Lint's detection of unreachable code is by no means perfect. Warning messages can be sent about valid code. It can also overlook commenting on code that cannot be reached. An example of this is the fact that *lint* does not know if a called function ever returns to the calling function (e.g. **exit**). *Lint* does not identify code following such a function call as being unreachable.

Suppressing Lint

Programs that are generated by *yacc* or *lex* can have many unreachable **break** statements. Normally, each one causes a complaint from *lint*. The **–b** option can be used to force *lint* to ignore unreachable **break** statements.

Problem Code: Function Value

The C compiler allows a function containing both the statement

return();

and the statement

return(expression);

to pass through without complaint. *Lint*, however, detects this inconsistency and responds with the message:

warning: function <name> has return(e); and return;

Problem Code: Type Matching

The C compiler does not strictly enforce the C language's type matching rules. At the loss of some type checking, the C compiler gains speed. An important role of *lint* is to enforce the type checking that the compiler neglects. It does this in four areas:

1. pointer types;
2. **long** and **int** type matching;
3. enumerations;
4. operations on structures and unions.

The types of pointers used in assignment, conditional, relational, and initialization statements must agree exactly. For example, the code:

```
int *p;
char *q;
.
.
.
p = q;
```

would cause *lint* to respond with the message:

```
warning: illegal pointer combination
```

Adding and subtracting integers and pointers are legal. Any other binary operation on them results in the message:

```
warning: illegal combination of pointer and integer: op <operator>
```

An example of code causing this message would be:

```
int s, *t;
.
.
.
t = s;
```

Assignments of long integer variables to integer variables are possible in the C language. However, on some machines the amount of storage supplied for the two types differs, and so the accuracy of a value could be lost in the conversion. *Lint* detects these assignments as possible program bugs. If a long integer is assigned to an integer, *lint* responds with:

```
warning: conversion from long may lose accuracy
```

Lint checks enumerations to see that variables or members are all of one type. Also, the only enumeration operations it allows are assignment, initialization, equality, and inequality. If *lint* finds code breaking any of these guidelines, it sends the message:

warning: enumeration type clash, operator <operator>

Structure and union references are subject to more type checking by *lint* than by the C compiler. *Lint* requires that the left operand of –> be a pointer to a structure or a union. If it isn't a pointer, *lint*'s response is:

warning: struct/union or struct/union pointer required

The left operand of **.** must be a structure or a union, which *lint* also indicates with the message above. The right operand of –> and **.** must be a member of the structure or union implied by the left operand. If it isn't then *lint*'s message is:

warning: illegal member use <name>

where <name> is the right operand.

Suppressing Lint

You may have a legitimate reason for converting a long integer to an integer. *Lint*'s **–a** option inhibits comments about these conversions.

Problem Code: Portability

Lint aids the programmer in writing portable code in five areas:

1. character comparisons;
2. pointer alignments;
3. uninitialized external variables;
4. length of external variables;
5. type casting.

Character representation varies on different machines. Characters may be implemented as signed values or as unsigned values. As a result, certain comparisons with characters give different results on different machines. The expression

c < 0

where *c* is defined as type character, is always true if characters are unsigned values. If, however, characters are signed values the expression could be either true or false. Where character comparisons could result in different values depending on the machine used, *lint* outputs the message:

warning: nonportable character comparison

Legal pointer assignments are determined by the alignment restrictions of the particular machine used. For example, one machine may allow double precision values to begin on any integer boundary, but another may restrict them to word boundaries. If integer and word boundaries are different, code containing an assignment of a double pointer to an integer pointer could cause problems. *Lint* attempts to detect where the effect of pointer assignments is machine dependent. The warning that it sends is:

```
warning: possible pointer alignment problem
```

Another machine dependent area is the treatment of uninitialized external variables. If two files both contain the declaration

```
int a;
```

either one word of storage is allocated or each occurrence receives its own word of storage, depending on the machine. If the files that *lint* is processing contain multiple definitions of the same uninitialized external variable, *lint* responds with:

```
warning: <name> redefinition hides earlier one
```

The amount of information about external symbols that is loaded depends on the machine being used: the number of characters saved and whether or not upper/lower case distinction is kept. *Lint* truncates all external symbols to six characters and allows only one case distinction. (It changes upper case characters to lower case.) This provides a worst-case analysis so that the uniqueness of an external symbol is not machine dependent.

The effectiveness of type casting in C programs can depend on the machine that is used. For this reason, *lint* ignores type casting code. All assignments that use it are subject to *lint*'s type checking (see *Problem Code: Type Matching*).

Suppressing Lint

The **–p** option stops comments about two types of portability problems:

1. pointer alignment problems,
2. multiple definitions of external variables.

Lint's objections to legal casts can also be suppressed. To do so, use its **–c** option.

Problem Code: Strange Constructions

A *strange construction* is code that *lint* considers to be bad style or a possible bug.

Lint looks for code that has no effect. An example is:

```
*p++;
```

where the * has no effect. The statement is equivalent to "p+ +;". In cases like this the message:

    warning: null effect

is sent.

The treatment of unsigned numbers as signed numbers in comparisons causes *lint* to report:

    warning: degenerate unsigned comparison

The following code would produce such a message:

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

Lint also objects if constants are treated as variables. If the boolean expression in a conditional has a set value due to constants, such as

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

lint's response is:

    warning: constant in conditional context

If the NOT operator is used on a constant value, the response is:

    warning: constant argument to NOT

To avoid operator precedence confusion, *lint* encourages using parentheses in expressions by sending the message:

    warning: precedence confusion possible; parenthesize!

Lint judges it bad style to redefine an outer block variable in an inner block. Variables with different functions should normally have different names. If variables are redefined, the message sent is:

    warning: <name> redefinition hides earlier one

Suppressing Lint

To stop *lint*'s comments about strange constructions, use its **–h** option.

Problem Code: Obsolete Constructions

C contains two forms of old syntax which, through the evolution of the language, are now officially discouraged. One is a group of assignment operators. Previously acceptable =+, =–, =*, =/, =%, =<<, =>>, =&, =^, and =| have been changed to +=, –=, *=, /=, %=, <<=, >>=, &=, ^=, and |=. If *lint* sees the older form, it responds with:

```
warning: old-fashioned assignment operator
```

The second syntax change deals with initialization. An older version of C allowed:

```
int a 0;
```

to initialize *a* to zero. Initialization now requires that an equals sign appear between the variable and the value it is to receive:

```
int a = 0;
```

Lint's response to the earlier version is:

```
warning: old-fashioned initialization: use =
```

How to Use Lint

To use *lint*, you must be logged into the HP-UX system and have a shell prompt on your screen. From here you can run *lint* on a single C source file:

```
$ lint filename.c
```

or on several source files which are to be linked together:

```
$ lint file1.c file2.c file3.c
```

The reappearance of your shell prompt after invoking *lint* tells you that *lint* has finished processing your files. If no messages were sent to your standard error file, *lint* found nothing wrong with your code.

Directives

The alternative to using options to suppress *lint*'s comments about problem areas is to use directives. Directives appear in the source code in the form of code comments. *Lint* recognizes five directives.

/*NOTREACHED*/	stops an unreachable code comment about the next line of code.
/*NOSTRICT*/	stops *lint* from strictly type checking the next expression.
/*ARGSUSED*/	stops a comment about any unused parameters for the following function.
/*VARARGS*n**/	stops *lint* from reporting variable numbers of parameters in calls to a function. The function's declaration follows this comment. The first *n* parameters must be present in each call to the function; *lint* comments if they aren't. If "/*VARARGS*/" appears without the *n*, none of the parameters need be present.
/*LINTLIBRARY*/	must be placed at the beginning of a file. This directive tells *lint* that the file is a library file and to suppress comments about unused functions. *Lint* objects if other files redefine routines that are found there.

Option List

The following is a list of the options available when using *lint*:

–a suppress complaints about assignments of integers to longs and of longs to integers.

–b suppress complaints about unreachable break statements.

–c suppress complaints about legal casts. Without this option typecasting is ignored.

–h suppress complaints about legal but strange constructions (see *Problem Code: Strange Constructions*).

–n do not check the compatibility of code against any libraries (standard and portable *lint* libraries, directive-defined libraries).

–p suppress some portability checks (see *Problem Code: Portability*).

–u suppress complaints about externals (functions and variables) that are used but not defined, or that are defined but not used (see *Problem Code: Unused Variables and Functions*, *Problem Code: Set/Used Information*).

–v suppress complaints about unused function parameters. If a parameter is unused and is also declared as a register variable, the warning is not suppressed.

–x suppress complaints about unused variables with external declarations (see *Problem Code: Set/Used Information*).

–D*name*[= *def*]

define the string *name* to *lint*, as if a **#define** control line were used. If no definition is given, then *name* is given the value 1. This option is also used by the C compiler.

–U*name*

remove any initial definition of *name*, as if a **#undef** control line were used. This option is also used by the C compiler.

–I*dir* change the algorithm for searching for **#include** files whose names do not begin with "/". The *dir* directory is searched before the directories on the standard list. Thus, **#include** files whose names are enclosed in double quotes (" ") are searched for first in the directory of the source file, then in the directory specified by each **–I** option, and finally in the directories on the standard list. If a **#include** file's name is enclosed in angle brackets (<>), the source file's directory is not searched. This option is also used by the C compiler.

Table of Contents

MC68000 Assembler on HP-UX . . . 1
- Instruction Format . . . 1
  - In General . . . 1
  - Symbols . . . 1
  - Local Labels . . . 1
  - Opcodes . . . 1
  - Size Suffixes . . . 2
  - Expressions . . . 2
- Pseudo-Op Syntax and Semantics . . . 3
- Interfacing Assembly Routines . . . 4
  - Linking . . . 4
  - Calling Conventions . . . 4
  - Language Dependencies . . . 7
    - C . . . 7
    - Fortran . . . 7
    - Pascal . . . 7
- Conversion from the Pascal Language System (PLS) . . . 9

MC68000 Assembler on HP-UX

Instruction Format

In General

Assembly instructions are written one per line. Mnemonic operation codes (opcodes) and register symbols must be written in lower case. Upper and lower case characters may not be used interchangeably, that is, it is a case sensitive assembler. Instructions are free format with respect to spaces.

If a label is present, it must start in column one of the line. The opcode must start in column two or later. Blanks are not permitted within the operand field. The first blank encountered after the start of the operand field begins the comment field.

```
Label      move a1,a2            comment field
```

A "*" in column one indicates a comment.

```
*
*          These are comments.
*
```

Symbols

Symbols must begin with an alphabetic character, but may contain letters, numbers, @, $ and _. Symbols may contain any number of characters. The restriction is that each instruction must be contained on one line.

* is a symbol having the value of the program counter.

Register symbols are those used to refer to the predefined registers. They are a0...a7, d0...d7, sp, pc, ccr, and sr.

Local Labels

A local label has the form <*digit*>$. A local label may be used to label any machine instruction. Any number of occurrences of the same local label may occur within an assembly source file. When a local label is referenced, the reference will refer to the nearest declaration of the local label.

Opcodes

Most opcodes and their syntaxes are defined in the *MC68000 User's Manual.* Size suffixes are only allowed for those operations which include a size field in the instruction and for the conditional branch `bcc`. In addition to the opcodes listed in the manual, the Series 200 will recognize some variants. For the `bcc` instruction the form `jcc` may be used. Also, `jbsr` may be used in place of `bsr`. In these cases, the assembler will decide the appropriate size for the instruction. No size suffix can be used.

Size Suffixes

Size suffixes are used in the language to specify the size of the operand in the instruction, including addressable locations and registers. All instructions which can operate on more than one data size will assume the default size of word (16 bits) unless a size suffix is used. Size suffixes can also be appended to address register specifications when used in indexed addressing. Operand sizes are defined as follows:

Suffix	Data Unit	Bits
b	byte	8
w	word	16
l	long	32

Expressions

Expressions are evaluated in left to right order, and parentheses are permitted. Symbols which refer to defined labels are permitted in expressions. The value of these symbols is their relative value within the assembled code. The only operations which can be done on these symbols are addition and subtraction. One label can be subtracted from another; the result is an absolute value. A label can be added to an absolute value but not to another symbol. The allowed operators are:

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
!	Bitwise *or*
&	Bitwise *and*
^	Bitwise *exclusive or*
<	Shift left
>	Shift right

Pseudo-Op Syntax And Semantics

The following is a list of the commands which direct the assembler to take the described actions. For a list of the machine commands, see the *MC68000 User's Manual.*

align <name>,<modulus>
Create a global symbol of type **align**. When the loader sees this symbol it will create a hole beginning at symbol *<name>* whose size will be such that the next symbol will be aligned on a <modulus> boundary.

asciz '<string>'
Put a null terminated <string> into the code at this point.

bss
Put the following assembly into the uninitialized data segment.

comm <name>,<size>
Create a global symbol <name>, put it in the bss segment with size <size>.

data
Place the following assembly in the initialized data segment.

dc[.b|.w|.l] <expr>|'<string>'[,<expr>|'<string>']
Place the list of expressions <expr> or strings <string> into the code at this point. Size suffixes may be used to specify the units of storage into which the values will be placed. Default is word. In the case of string literals, the amount of storage needed will be determined by the assembler and each character will be assigned into a unit.

ds[.b|.w|.l] <expr>
The units of space are specified by the size suffix. The number of units is determined by the expression.

equ <expr>
Assigns the value and attributes of the expression to the label.

even
Forces even word alignment.

globl <name>[,<name>]...
Declares the list of names to be global symbols.

include "<name>"|<<name>>
Specifies a file to be merged into the assembly at the point where the instruction is located. The file will be searched for according to the conventions of C (see manual page for cc).

text
Place the following assembly in the code segment.

Interfacing Assembly Routines

In order to know how to use the assembler effectively, it will be necessary to know how to interface to the various higher level languages that the HP-UX Series 200 supports.

Linking

In order for a symbol to be known externally it must be declared in a `globl` statement. It is not necessary for a symbol defined externally to be declared in a module. If a symbol is not defined, it is assumed to be externally defined. It is, however, recommended that all external symbols be declared in a `globl` statement, since this will avoid possible name confusion with local symbols.

Calling Conventions

All languages currently supported on the Series 200 follow certain conventions regarding the calling of subroutines. These conventions must be followed in order to call or be called by a higher level language.

The calling conventions can be summarized as follows:

- Parameters are pushed in reverse order and taken off in the same order as the procedure call;
- The calling routine pops the parameters from the stack upon return;
- The called routine saves and restores the registers it uses (except d0, d1, a0, a1);
- Function results are generally returned in d0, d1;
- `tst.b` required for all stack space used plus that required for the link of any routine called; and
- `link/unlk` instructions are used to allocate local data space and to reference parameters.

These conventions can be more easily understood by means of an example. The best would be to examine the code output by the compiler to do this. This can be easily done using C since it outputs assembly language instructions. Consider the following C program.

```
main()
{
        test(1,2);
}
test(i,j)
register int i, j;
{
        int k;
        k = i + j;
        return k;
}
```

It will produce the following assembly language instructions.

```
 1              data
 2              text
 3              globl    _main
 4        _main
 5              link     a6,#-__F1
 6              tst.b    -__M1-8(a7)
 7              movem.l  #__S1,-__F1(a6)
 8              move.l   #2,-(sp)
 9              move.l   #1,-(sp)
10              jbsr     _test
11              addq     #8,sp
12              jra      L12
13        L12   unlk     a6
14              rts
15        __F1  equ      0
16        __S1  equ      0
17        __M1  equ      0
18              data
19              text
20              globl    _test
21        _test
22              link     a6,#-__F2
23              tst.b    -__M2-8(a7)
24              movem.l  #__S2,-__F2(a6)
25              move.l   8(a6),d7
26              move.l   12(a6),d6
27              move.l   d7,d0
28              add.l    d6,d0
29              move.l   d0,-4(a6)
30              move.l   -4(a6),d0
31              jra      L14
32              jra      L14
33        L14   movem.l  -__F2(a6),#192
34              unlk     a6
35              rts
36        __F2  equ      12
37        __S2  equ      192
38        __M2  equ      0
39              data
```

Things to note are that when the parameters are pushed by the calling routine (`_main`), the second parameter is pushed first and the first parameter is pushed second (lines 8 and 9). When the called routine (`_test`) goes to access the parameters (lines 25 and 26), it finds the first parameter first on the stack and the second parameter second. Line 25 accesses the first parameter and line 26 accesses the second parameter.

Also note that the stack is popped upon return from the subroutine (line 11) and not by the subroutine itself. Since the called routine makes use of `d6` and `d7`, it pushes those registers on the stack (line 24) and then pops them (line 33) before it returns.

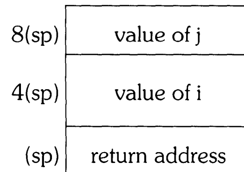
The function result is placed in `d0` before returning (line 30). If the function returned a double precision floating point number, that number would have been placed in `d0` and `d1`.

A `tst.b` instruction (line 23) is needed before any use is made of stack space in any assembly language routine. The `tst.b` makes sure that there is enough stack space for this routine. If the test fails, the operating system can detect this and get more stack space for the process. If the test is not done, the program may die unnecessarily with a segmentation violation. The amount of space that must be tested for is the sum of:

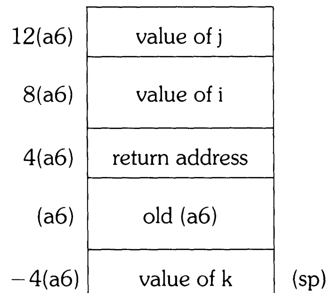
- The amount of space taken by the link instruction;
- The greatest amount of space used for any parameters that may be pushed;
- The constant 8 to account for subroutine jumps and the link which that routine may do.

C and other higher level languages use the link and unlk instructions (lines 22, 34) in all routines. The link instruction is used to allocate local data space and to allow a constant reference point for accessing parameters. The following illustration shows what happens when the link instruction on line 22 is executed.

Before the link:



After the link:



Note how the parameter `i` is accessed on line 25. On line 29 the local variable `k` is set. The link instruction is not necessary in assembly language code. If it is not there, however, the routine will not show up in a stack backtrace from **adb**. If a `link` instruction is done, an `unlk` must be done before returning.

Language Dependencies

C

In C, all variables and functions declared by the user are prefixed with an underbar. Thus, a variable named `test` in C would be known as `_test` at the assembly language level. All global variables can be accessed through this name using a long absolute mode of addressing. C will always push a four-byte quantity on the stack for pointers and any form of integer (char, short, long). C will always push eight bytes for a floating point number (floats are converted to double).

Fortran

Fortran uses the same naming convention as C, and externals can be accessed in the same fashion. Fortran will always push the address of its parameter for user-defined functions.

Pascal

In Pascal, any exported user-defined function is prefixed by the module name surrounded by underbars. For Pascal, then, a function named `funk` in module `test` would be known as `_test_funk` to an assembly language programmer. If a procedure is declared external as in:

```
procedure proc;  external;
```

all calls to `proc` will emit a reference to `_proc`.

Global variables are accessed as a 32-bit absolute relative to the global base. In the example below, the global variable `i1` would be accessed as:

```
move.l  test+0x4,d0
```

Following is the example:

```
Pascal [Rev 2.1Ma  4/19/83] test.p                          Page 1

     1:D         0 $list 'test.l',tables$
     2:D         0 program test;
     3:D         1 var
     4:D      8  1   i1,i2: integer;
     5:D         1 procedure p;
     6:D         2   var
     7:D     -4  2     j: integer;
     8:C         2   begin

Dump of P
 j                 var lev= 0d2 addr=-00000004 local

P dump complete

     9:C         2   end;
    10:C         1 begin

Dump of TEST
 i1                var lev= 0d1 addr=00000004 longabs globalbase = test
 i2                var lev= 0d1 addr=00000000 longabs globalbase = test
 p                 proc lev= 0d1 entry: 00000000

 test              proc lev= 0d0 entry: 00000012


TEST dump complete

    11:C         1 end.
```

Pascal will always push a four-byte quantity on the stack for pointers and integers. For a user-defined function, any parameter greater than four bytes will be passed as an address.

The manual pages for these compilers should be consulted for further information. Assembly listings can be generated by C and Fortran. These can be consulted to get valuable information. The only current means for looking at the code generated by Pascal is through the debugger **adb**.

Conversion from the Pascal Language System (PLS)

A translator (**atrans**) is provided to assist in converting from PLS assembly language to HP-UX assembly language syntax. All code to be ported should be run through the translator first. Lines that will require human intervention will be noted by the translator. To see exactly what the tasks are that it performs, check the manual page.

atrans will not detect or alter parameter passing conventions which are pushed in the opposite order on PLS.

as assumes `rorg 0` for all assemblies. **as** does not generate relative references to external symbols; all external references are absolute. As such, code size can increase when being ported from the PLS to HP-UX.

as does not have support for Pascal modules.

as will accept the same syntax as the PLS assembler for all machine instructions with these exceptions:

Additions:

- **as** will accept `jcc` where `cc` is a condition code accepted by `bcc`. In this case, **as** will decide the length of the instruction required.
- **as** will accept a greater number of operators for expressions. Parentheses are permitted within expressions.
- **as** will accept an immediate operand for the register list in a `movem` instruction. Needed for compiler.
- **as** will allow numeric value for displacement as in `12(pc,d6)`. Needed for compiler.
- **as** will accept <digit>`$` to specify a local label.

Differences:

- **as** is a case-sensitive assembler. All opcodes and register names must be listed in lower case.
- **as** accepts `(pc)` to specify pc-relative references. This is the only way to specify pc-relative.
- The PLS assembler will assume pc with index in some cases for a parameter of the form `8(a0)`. **as** will not.

The greatest differences occur in the pseudo-ops that are supported. The only PLS pseudo-ops that are supported are `dc`, `ds`, `equ`, and `include`. The translator will handle some of the other pseudo-ops, but others will have to be handled by hand.

Table of Contents

Ratfor: A Preprocessor for a Rational Fortran

Introduction 2
Language Description 3
Design 3
Statement Grouping 3
The "else" Clause 4
Nested *ifs* 5
If-else Ambiguity 6
The "switch" Statement 7
The "do" Statement 8
"Break" and "next" 9
The "while" Statement 9
The "for" Statement 11
The "repeat-until" Statement 12
More on *break* and *next* 13
The "return" Statement 13
Cosmetics 14
Free-form Input 14
Translation Services 14
The "define" Statement 15
The "include" Statement 16
Pitfalls, Botches, Blemishes and other Failings 16
Experience 17
Good Things 17
Bad Things 17
Conclusions 18
Appendix: Usage on HP-UX 18

Ratfor: A Preprocessor for a Rational FORTRAN

Although FORTRAN is not a pleasant language to use, its universality and relative efficiency maintain its position in the computer market. The Ratfor language, by providing control flow statements, attempts to conceal the main deficiencies of FORTRAN while retaining its desirable qualities. The Ratfor preprocessor converts input code into FORTRAN output code. The facilities provided include:

- Statement grouping
- `If-else` and `switch` for decision-making
- `While`, `for`, `do`, and `repeat-until` for looping
- `Break` and `next` for controlling loop exits
- Free-form input such as multiple statements/lines, and automatic continuation
- Simple comment convention
- Translation of >, >=, etc., into .gt., .ge., etc.
- `Return` function for functions
- `Define` statement for symbolic parameters
- `Include` statement for including source files.

Introduction

Most programmers agree that FORTRAN is an unpleasant language to program in, yet there are many occasions when they are forced to use it, especially when FORTRAN is the only language thoroughly supported on the local computer, or the application requires intensive computation.

FORTRAN's worst deficiency is probably in control flow statements, conditional branches and loops, that express the logic of program flow. For example, FORTRAN's primitive conditional statements force the user into at least two statement numbers and two implied `GOTO`s to handle a single arithmetic `IF`. This leads to unintelligible code that is eschewed by good programmers.

The Logical `IF` is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the *IF* can only be one FORTRAN statement (with some **further** restrictions!). And of course there can be no *ELSE* part to a FORTRAN `IF`: there is no way to specify an alternative action if the `IF` is not satisfied.

The FORTRAN `DO` restricts the user to going forward in an arithmetic progression. It is fine for "1 to N in steps of 1 (or 2 or ...)", but there is no direct way to go backwards, or even (in ANSI FORTRAN) to go from 1 to N-1. And of course the `DO` is useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that FORTRAN programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with Ratfor (The preprocessor idea is not new, and FORTRAN preprocessors are widely used).

Language Description

Design

Ratfor attempts to retain the merits of FORTRAN (universality, portability, efficiency) while hiding the worst FORTRAN inadequacies. The language **is** FORTRAN except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of Ratfor is to conceal this part of FORTRAN from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without *GOTO*'s. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the "cosmetic" deficiencies of FORTRAN, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects – control flow and cosmetics – Ratfor does nothing about the host of other weaknesses of FORTRAN. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in Ratfor and what should not has been *Ratfor* **doesn't** know any FORTRAN. Any language feature which would require that Ratfor really understand FORTRAN has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the Ratfor language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

Statement Grouping

FORTRAN provides no way to group statements together, short of making them into a subroutine. The standard construction "if a condition is true, do this group of things," for example,

```
if (x > 100)
   { call error("x>100"); err = 1; return }
```

cannot be written directly in FORTRAN. Instead a programmer is forced to translate this relatively clear thought into murky FORTRAN, by stating the negative condition and branching around the group of statements:

```
      if (x .le. 100) goto 10
            call error(5hx>100)
            err = 1
            return
10    ...
```

When the program doesn't work, or when it must be modified, this must be translated back into a clearer form before one can be sure what it does.

Ratfor eliminates this error-prone and confusing back-and-forth translation; the first form **is** the way the computation is written in Ratfor. A group of statements can be treated as a unit by enclosing them in the braces { and }. This is true throughout the language: wherever a single Ratfor statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than *begin* and *end* or *do* and *end*, and of course *do* and *end* already have FORTRAN meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character ">" is clearer than "*.GT.*", so Ratfor translates it appropriately, along with several other similar shorthands. Although many FORTRAN compilers permit character strings in quotes (like `"""x>100"""`), quotes are not allowed in ANSI FORTRAN, so Ratfor converts it into the right number of `H`'s because computers count better than people do.

Ratfor is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
   call error("x>100")
   err = 1
   return
}
```

In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the `if` is a single statement (Ratfor or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
   write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general Ratfor continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

The "else" Clause

Ratfor provides an "*else*" statement to handle the construction "if a condition is true, do this thing, otherwise do that thing."

```
if (a <= b)
     { sw = 0; write(6, 1) a, b }
else
     { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of *a* and *b*, then the larger, and sets *sw* appropriately.

The FORTRAN equivalent of this code is circuitous indeed:

```
        if (a .gt. b) goto 10
            sw = 0
            write(6, 1) a, b
            goto 20
10      sw = 1
            write(6, 1) b, a
20        ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that is not a translation. To understand the FORTRAN version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an *if-else* construction. With the Ratfor version, there is no question about how one gets to the parts of the statement. The *if-else* is a single unit, which can be read, understood, and ignored if not relevant. The program says what it means.

As before, if the statement following an *if* or an *else* is a single statement, no braces are needed:

```
if (a <= b)
   sw = 0
else
   sw = 1
```

The syntax of the *if* statement is

```
if (<legal FORTRAN condition>)
   Ratfor statement
else
   Ratfor statement
```

where the *else* part is optional. The <legal FORTRAN condition> is anything that can legally go into a FORTRAN Logical `IF`. Ratfor does not check this clause, since it does not know enough FORTRAN to know what is permitted. The *Ratfor statement* is any Ratfor or FORTRAN statement, or any collection of them in braces.

Nested *ifs*

Since the statement that follows an *if* or an *else* can be any Ratfor statement, this leads immediately to the possibility of another *if* or *else*. As a useful example, consider this problem:

The variable *f* is to be set to -1 if *x* is less than zero, to $+1$ if *x* is greater than 100, and to 0 otherwise. In Ratfor, we write

```
if (x < 0)
    f = -1
else if (x > 100)
        f = +1
else
    f = 0
```

Here the statement after the first `else` is another `if-else`. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight Fortran will necessarily be indirect because Fortran does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an *else* with an *if* is one way to write a multi-way branch in Ratfor. In general the structure

```
        if (...)
            - - -
        else if (...)
            - - -
        else if (...)
            - - -
        ...
    else
    ---
```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a *switch* statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing *else* part handles the "default" case, where none of the other conditions apply. If there is no default action, this final *else* part is omitted:

```
    if (x < 0)
        x = 0
    else if (x > 100)
        x = 100
```

If-else Ambiguity

There is one thing to notice about complicated structures involving nested *if*s and *else*s. Consider

```
    if (x > 0)
            if (y > 0)
                write(6, 1) x, y
            else
                write(6, 2) y
```

There are two *if*s and only one *else*. Which *if* does the *else* go with?

This is a genuine ambiguity in Ratfor, as it is in many other programming languages. The ambiguity is resolved in Ratfor (as elsewhere) by saying that in such cases the *else* goes with the closest previous *else*ed un-*if*. Thus in this case, the *else* goes with the inner *if*, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
    else
        write(6, 2) y
}
```

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we **must** write

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
}
    else
    write(6, 2) y
```

The "switch" Statement

The *switch* statement provides a clean way to express multi-way branches which branch on the value of some integer-valued expression. The syntax is

```
switch (<expression>) {
    case <expr1>:
    statements
    case <expr2>, <expr> :
        statements
    ...
default:
statements
}
```

Each *case* is followed by a list of comma-separated integer expressions. The <expression> inside *switch* is compared against the case expressions <expr1>, <expr2>, and so on in turn until one matches, at which time the statements following that *case* are executed. If no cases match <expression>, and there is a *default* section, the statements with it are done; if there is no *default*, nothing is done. In all situations, as soon as some block of statements is executed, the entire *switch* is exited immediately. (Readers familiar with C should beware that this behavior is not the same as the C *switch*.)

The "do" Statement

The *do* statement in Ratfor is quite similar to the `DO` statement in FORTRAN, except that it uses no statement number. The statement number, after all, serves only to mark the end of the `DO`, and this can be done just as easily with braces. Thus

```
do i = 1, n {
    x(i) = 0.0
    y(i) = 0.0
    z(i) = 0.0
}
```

is the same as

```
    do 10 i = 1, n
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
10  continue
```

The syntax is:

`do` <legal FORTRAN text>
    Ratfor statement

The part that follows the keyword *do* has to be something that can legally go into a FORTRAN `DO` statement. Thus if a local version of FORTRAN allows `DO` limits to be expressions (which is not currently permitted in ANSI FORTRAN), they can be used in a Ratfor *do*.

The *Ratfor statement* part will often be enclosed in braces, but as with the *if*, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
    x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
    do j = 1, n
        m(i, j) = 0
```

sets the entire array *m* to zero, and

```
do i = 1, n
    do j = 1, n
        if (i < j)
            m(i, j) = -1
        else if (i == j)
            m(i, j) = 0
        else
            m(i, j) = +1
```

sets the upper triangle of *m* to -1, the diagonal to zero, and the lower triangle to $+1$. (The operator == is "equals"; that is, ".EQ.".) In each case, the statement that follows the *do* is logically a **single** statement, even though complicated, and thus needs no braces.

"Break" and "next"

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. `Break` causes an immediate exit from the `do`; in effect it is a branch to the statement **after** the `do`. `Next` is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
    if (x(i) < 0.0)
        next
    <process positive element>
}
```

`Break` and `next` also work in the other Ratfor looping constructions discussed in the next few sections.

`Break` and *next* can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and `Break 1` is equivalent to `break`. `next 2` iterates the second enclosing loop. (Realistically, multi-level `breaks` and `nexts` are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

The "while" Statement

One of the problems with the FORTRAN `DO` statement is that it generally insists upon being done once, regardless of its limits. If a loop begins

```
DO I = 2, 1
```

this will typically be done once with *I* set to 2, even though common sense would suggest that perhaps it shouldn't be. Of course a Ratfor `do` can easily be preceded by a test

```
if (j <= k)
    do i = j, k  {
        - - -
    }
```

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the `DO` statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the FORTRAN `DO`, it is that much harder to write and understand.

To overcome these difficulties, Ratfor provides a `while` statement, which is simply a loop: "while some condition is true, repeat this group of statements". It has no preconceptions about why one is looping. For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria.

```
real function sin(x, e)
     # returns sin(x) to accuracy e, by
     # sin(x) = x - x**3/3! + x**5/5! - ...
     sin = x
     term = x
     i = 3
     while (abs(term)>e & i<100) {
          term = -term * x**2 / float(i*(i-1))
          sin = sin + term
          i = i + 2
     }
     return
     end
```

Notice that if the routine is entered with *term* already smaller than *e*, the loop will be done **zero times**, that is, no attempt will be made to compute `x**3` and thus a potential underflow is avoided. Since the test is made at the top of a `while` loop instead of the bottom, a special case disappears: the code works at one of its boundaries. (The test *i<100* is the other boundary, making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character "#" in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line − one can make marginal remarks, which is not possible with FORTRAN's "C in column 1" convention. Blank lines are also permitted anywhere (they are not in FORTRAN); they should be used to emphasize the natural divisions of a program.

The syntax of the `while` statement is

> `while` (*legal FORTRAN condition*)
> *Ratfor statement*

As with the *if*, *legal* FORTRAN condition is something that can go into a FORTRAN Logical `IF`, and *Ratfor* statement is a single statement, which may be multiple statements in braces.

The `while` encourages a style of coding not normally practiced by FORTRAN programmers. For example, suppose *nextch* is a function which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

```
while (nextch(ich) == iblank)
     ;
```

A semicolon by itself is a null statement, which is necessary here to mark the end of the `while`; if it were not present, the `while` would control the next statement. When the loop is broken, `ich` contains the first non-blank. Of course the same code can be written in FORTRAN as

```
100 if (nextch(ich) .eq. iblank) goto 100
```

but many FORTRAN programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

The "for" Statement

The `for` statement is another Ratfor loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the `while`. A `for` statement allows explicit initialization and increment steps as part of the statement. For example, a `DO` loop is just

```
for (i = 1; i <= n; i = i + 1) ...
```

This is equivalent to

```
i = 1
while (i <= n) {
     ...
     i = i + 1
}
```

The initialization and increment of *i* have been moved into the *for* statement, making it easier to see at a glance what controls the loop.

The *for* and *while* versions have the advantage that they will be done zero times if *n* is less than 1; this is not true of the *do*.

The loop of the sine routine in the previous section can be rewritten with a *for* as

```
for (i=3; abs(term) > e & i < 100; i=i+2) {
     term = -term * x**2 / float(i*(i-1))
     sin = sin + term
}
```

The syntax of the *for* statement is

```
for (<init>; <condition>; <increment>)
     Ratfor statement
```

<init> is any single FORTRAN statement that is executed once before the loop begins.

<increment> is any single FORTRAN statement, that gets done at the end of each pass through the loop, before the test.

<condition> is, again, anything that is legal in a logical `IF`.

Any of <init>, <condition>, and <increment> can be omitted, although the semicolons **must** always be present. A non-existent <condition> is treated as always true, so *for(;;)* is an indefinite repeat (but see the *repeat-until* in the next section).

The *for* statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a `DO` statement, and obscure to write out with `IF`s and `GOTO`s. For example, here is a backwards `DO` loop to find the last non-blank character on a card:

```
for (i = 80; i > 0; i = i - 1)
     if (card(i) != blank)
          break
```

(!= is the same as .NE.). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken *break and next* work in *for*s and *while*s just as in *do*s). If *i* reaches zero, the card is all blank.

This code is rather nasty to write with a regular FORTRAN DO, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. (Forgetting this is a common error.) Thus:

```
      DO 10 J = 1, 80
              I = 81 - J
              IF (CARD(I) .NE. BLANK) GO TO 11
10    CONTINUE
          I = 0
11    ...
```

The version that uses the *for* handles the termination condition properly for free; *i* is zero when we fall out of the *for* loop.

The increment in a *for* need not be an arithmetic progression; the following program walks along a list (stored in an integer array *ptr*) until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
    sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

The "repeat-until" Statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the *repeat-until*:

```
repeat
    Ratfor statement
until (legal FORTRAN condition)
```

The *Ratfor* statement part is done once, then the condition is evaluated.

If it is true, the loop is exited.
– If it is false, another pass is made.

The *until* part is optional, so a bare *repeat* is the cleanest way to specify an infinite loop.

Of course such a loop must ultimately be broken by some transfer of control such as *stop*, *return*, or *break*, or an implicit stop such as running out of input with a READ statement.

It is a matter of observed fact that the *repeat-until* statement is **much** less used than the other looping constructions; in particular, it is typically outnumbered ten to one by *for* and *while*. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

More on *break* and *next*

Break exits immediately from *do*, *while*, *for*, and *repeat-until*. *Next* goes to the test part of *do*, *while* and *repeat-until*, and to the increment step of a *for*.

The "return" Statement

The standard FORTRAN mechanism for returning a value from a function uses the name of the function as a variable. The variable is assigned by the program, and the last value stored in it is the function value upon return. For example, here is a routine *equal* which returns 1 if two arrays are identical, and zero if they differ. The array ends are marked by the special value -1.

```
# equal - compare str1 to str2;
#
return 1 if equal, 0 if not
    integer function equal(str1, str2)
    integer str1(100), str2(100)
    integer i
    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == -1) {
           equal = 1
           return
        }
    equal = 0
    return
    end
```

In many languages (e.g., PL/I) one instead says

```
return ( <expression> )
```

to return a value from a function. Since this is often clearer, Ratfor provides such a *return* statement. In a function *f*, *return (*expression*)* is equivalent to

```
{ F = <expression>; <return> }
```

For example, here is *equal* again:

```
# equal _ compare str1 to str2;
#
return 1 if equal, 0 if not
    integer function equal(str1, str2)
    integer str1(100), str2(100)
    integer i
    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == -1)
        return(1)
    return(0)
    end
```

If there is no parenthesized expression after *return*, a normal `RETURN` is made. (Another version of *equal* is presented shortly.)

Cosmetics

As previously stated, the visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, Ratfor provides a number of cosmetic facilities which may be used to make programs more readable.

Free-form Input

Statements can be placed anywhere on a line. Long statements are continued automatically, as are long conditions in *if*, *while*, *for*, and *until*. Blank lines are ignored. Multiple statements may appear on one line if they are separated by semicolons. No semicolon is needed at the end of a line if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

```
=   +   -   *   ,   |   &   (   _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a FORTRAN label, and placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format("hello")
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

Translation Services

Text enclosed in matching single or double quotes is converted to `nH...` but is otherwise unaltered (except for formatting – it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash (\) serves as an escape character: the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
"\\\'"
```

is a string containing a backslash and an apostrophe. (This is **not** the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character (%) is left absolutely unaltered except for stripping off the (%) and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing FORTRAN program). Use (%) only for ordinary statements; not for the condition parts of *if*, *while*, etc.; or the output may be positioned incorrectly.

The following character translations are made, except within single or double quotes or on a line beginning with a percent sign (%).

Input	Translated output
`==`	.eq.
`!=`	.ne.
`>`	.gt.
`>=`	.ge.
`<`	.lt.
`<=`	.le.
`&`	.and.
`|`	.or.
`!`	.not.
`^`	.not.
	In addition, the following translations are provided for input devices with restricted character sets.
`[`	`{`
`]`	`}`
`$(`	`{`
`$)`	`}`

The "define" Statement

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

`Define` is typically used to create symbolic parameters:

```
define ROWS 100
define COLS 50
dimension a(ROWS), b(ROWS, COLS)
    if (i > ROWS | j > COLS) ...
```

Alternately, definitions can be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this allows multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers. As an example, here is the routine *equal* again, this time with symbolic constants.

```
define YES 1
define NO 0
define EOS -1
define ARB 100
# equal - compare str1 to str2;
#
return YES if equal, NO if not
    integer function equal(str1, str2)
    integer str1(ARB), str2(ARB)
    integer i
    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == EOS)
        return(YES)
    return(NO)
    end
```

The "include" Statement

The statement

```
include file
```

inserts the file found on input stream `file` into the Ratfor input in place of the `include` statement. The standard usage is to place `COMMON` blocks on a file, and *include* that file whenever a copy is needed:

```
subroutine x
    include commonblocks
    ...
    end
subroutine y
    include commonblocks
    ...
    end
```

This ensures that all copies of the `COMMON` blocks are identical

Pitfalls, Botches, Blemishes and other Failings

Ratfor catches certain syntax errors, such as missing braces, `else` clauses without an `if`, and most errors involving missing parentheses in statements. Beyond that, since Ratfor knows no FORTRAN, any errors you make will be reported by the FORTRAN compiler, so you will from time to time have to relate a FORTRAN diagnostic back to the Ratfor source.

Keywords are reserved. Using *if*, *else*, etc., as variable names will typically wreak havoc.

Don't leave spaces in keywords. Don't use the Arithmetic `IF`.

The FORTRAN `nH` convention is not recognized anywhere by Ratfor; use quotes instead.

Experience

Good Things

"It's so much better than FORTRAN" is the most common response of users when asked how well Ratfor meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts FORTRAN from a bad language into quite a reasonable one, assuming that FORTRAN data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in FORTRAN. More important, debugging and subsequent revision are much faster than in FORTRAN. Partly this is simply because the code can be *read*. The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in Ratfor tend to be as readable as programs written in more modern languages like Pascal. Once one is freed from the shackles of FORTRAN's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a Ratfor implementation of a linear table search:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
     ;
if (i > m) {
     m = i
     B(i) = 1
}
else
     B(i) = B(i) + 1
```

Bad Things

The biggest single problem is that many FORTRAN syntax errors are not detected by Ratfor but by the local FORTRAN compiler. The compiler then prints a message in terms of the generated FORTRAN, and in a few cases this may be difficult to relate back to the offending Ratfor line, especially if the implementation conceals the generated FORTRAN. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since Ratfor generates no variables, only a simple pattern of `IF`s and `GOTO`s, data-related errors like missing `DIMENSION` statements are easy to find in the FORTRAN. Furthermore, there has been a steady improvement in Ratfor's ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic `IF`. A few standard FORTRAN constructions are not accepted by Ratfor, and this is perceived as a problem by users with a large corpus of existing FORTRAN programs. Protecting every line with a (%) is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program *struct*, which converts arbitrary FORTRAN programs into Ratfor.

Users who export programs often complain that the generated FORTRAN is "unreadable" because it is not tastefully formatted and contains extraneous `CONTINUE` statements. To some extent this can be ameliorated (Ratfor now has an option to copy Ratfor comments into the generated FORTRAN), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success; since Ratfor is relatively easy to modify, there are now several dialects of Ratfor. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

Conclusions

Ratfor demonstrates that with modest effort it is possible to convert FORTRAN from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in "features"; things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for Ratfor to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he **can** format it neatly. No one should read the output anyway except in the most dire circumstances.

Appendix: Usage on HP-UX

Beware. Local customs vary. Check with a native before going into the jungle.

The program *ratfor* is the basic translator; it takes either a list of file names or the standard input and writes FORTRAN on the standard output. Options include `-6x`, which uses `x` as a continuation character in column 6 (HP-UX uses & in column 1), and *−C*, which causes Ratfor comments to be copied into the generated FORTRAN.

The program *rc* provides an interface to the *ratfor* command which is much the same as *cc*. Thus

    rc [<options>] <files>

compiles the files specified by <files>. Files with names ending in **.r** are Ratfor source; other files are assumed to be for the loader. The flags `-C` and `-6x` described above are recognized, as are

`-c`	compile only; don't load.	
`-f`	save intermediate FORTRAN *.f* files	
`-r`	Ratfor only; implies `-c` and `-f`.	
`-2`	use big FORTRAN compiler	(for large programs)
`-U`	flag undeclared variables	(not universally available)

Other flags are passed on to the loader.

Table of Contents

Overview

Getting Started ... 1
Who Will Use Native Language Support? ... 1
Manual Organization ... 1
Conventions Used In This Manual ... 2
Using Other HP-UX Manuals ... 3

Chapter 1: Introduction to Native Language Support

What Is NLS? ... 5
Scope of Native Language Support ... 6
Aspects of NLS Support ... 6
Pre-localized Commands ... 8
Supported Native Languages and Character Sets ... 9
8-Bit Character Sets ... 9
Native Languages ... 13

Chapter 2: Native Language Support

File Hierarchy ... 15
Configuring Native Languages ... 16
Installation of Optional Languages ... 16
Environment Changes ... 16
Accessing NLS Features ... 17
NLS HP-UX Commands ... 17
Library Support for NLS ... 17

Chapter 3: Programming With Native Language Support

NLS Header Files ... 19
Library Routines ... 19
Convert Date/Time to String ... 19
Convert Floating Point to String ... 20
C Routines to Translate Characters ... 20
C Routines That Classify Characters ... 20
Get Message From Catalog ... 21
Information on User's Native Language ... 21
Print Formatted Output With Numbered Arguments ... 22
Non-ASCII String Collation ... 23
Convert String to Double Precision Number ... 23
Application Guidelines ... 24
Example C Programs ... 24
Example 1 ... 24
Example 2 ... 25

Chapter 4: Message Catalog System
- Introduction 27
- Creating a Message Catalog 29
  - Preview: Incorporating NLS into Commands 29
  - Following the Flow 30
  - Format of Source Message File 33
  - Printmsg, Fprintmsg and Sprintmsg 34
- Accessing Applications Catalogs 35
- File System Organization and Catalog Naming Conventions 35
- Localization 36
  - 6 Steps to Localize an Example Program 36

Appendix A: Pre-localization Commands 39

Appendix B: Native Language Support Library 41

Appendix C: Peripheral Configuration
- European Character Sets 43
- Japanese Character Sets 43
- ISO 7-bit Substitution 43
- Character Set Support by Peripherals 43

Appendix D: Character Sets 45

Glossary 53

Index

Overview

Getting Started

If you're like most people, reading computer manuals is not your favorite pastime. We strongly urge you to read the remainder of this chapter. This manual assumes that you have read these first few pages; if you choose not to do so, you are on your own.

One other note: the best way for us to improve the quality of documentation is through your feedback. Please use one of the reply cards at the back of this manual to tell us what was helpful, what was not, and why. Feel free to comment on depth, technical accuracy, organization, and style. Your comments are appreciated.

Who Will Use Native Language Support?

OEMs (Original Equipment Manufacturers), ISVs (Independent Software Vendors), applications programmers, and Hewlett-Packard Country Software Centers will be the primary users of Native Language Support (NLS). These are the people writing or translating programs for multi-national use.

This manual has been written with these users in mind.

Manual Organization

Overview
Defines the NLS user audience, explains the conventions used in the manual, and identifies other manuals referenced within this one.

Chapter 1: Introduction to Native Language Support
Presents the basic description and scope of Native Language Support. This includes the aspects of NLS (Character Set Support, Local Customs, and Messages), pre-localization, and the character sets as well as native languages supported.

Chapter 2: Native Language Support on HP-UX
Identifies the HP-UX directories and files in which the NLS tools reside, provides an installation guide for the optional languages, and identifies the library calls (and commands) that an applications programmer needs in order to access NLS features.

Chapter 3: Programming With Native Language Support
Presents the header files specific to NLS, a detailed description of the C library routines (with their syntax), and example C programs (with their command lines and output).

Chapter 4: Message Catalog System
Explains how local language message files are created and updated, where they are kept, and by what conventions they are named. This includes a diagram and description of the general flow of the message catalog system, ways to access catalogs by use of library routines, file naming conventions and an example of program output in a local language other than American English.

Appendix A: Pre-localized Commands
Describes the HP-UX commands that currently incorporate Native Language Support.

Appendix B: Native Language Support Library
Overview of NLS library routines and routines affected by NLS.

Appendix C: Peripheral Configuration
Table summary of Series 200/500 peripherals that support alternate character sets.

Appendix D: Character Sets
ASCII, Roman and Katakana character sets with their decimal and binary representations.

Conventions Used In This Manual

The following naming conventions are used throughout this manual.

- Italics indicate files and HP-UX commands, system calls, and subroutines found in the *HP-UX Reference* manual as well as titles of manuals. Italics are also used for symbolic items either typed by the user or displayed by the system as discussed below. Examples include */usr/lib/nls/american/prog.cat*, *date(1)*, and *pty(4)*. The parenthetic number shown for commands, system calls, and other items found in the *HP-UX Reference* is a convention used in that manual.
- **Boldface** is used when a word is first defined and for **general emphasis**.
- Computer font indicates a literal typed by the user or displayed by the system. A typical example is:

  ```
  findstr prog.c > prog.str
  ```

  Note that when a command or file name is part of a literal, it is shown in computer font and not italics. However, if the command or file name is symbolic (but not literal), it is shown in italics as the following example illustrates.

  `findstr` *progname* > *output-file-name*

  In this case you would type in your own *progname* and *output-file-name*.
- Environment variables such as LANG or PATH are represented in uppercase characters.
- Unless otherwise stated, all references such as "see the *nl_toupper(3C)* entry for more details" refer to entries in the *HP-UX Reference* manual. Some of these entries will be under an associated heading. For example, the *nl_toupper(3C)* entry is under the *nl_conv(3C)* heading. If you cannot find an entry where you expect it to be, use the *HP-UX Reference* Manual's Permuted Index.

Using Other HP-UX Manuals

This manual may be used in conjunction with other HP-UX documentation. References to these manuals are included, where appropriate, in the text.

- The *HP-UX Reference* manual contains the syntactic and semantic details of all commands and application programs, system calls, subroutines, special files, file formats, miscellaneous facilities, and maintenance procedures available on the Series 200/500 HP-UX Operating System.
- The *HP-UX Portability Guide* documents the guidelines and techniques for maximizing the portability of programs written on and for HP9000 computers running the HP-UX Operating System. It covers the portability of high level source code (C, Pascal, FORTRAN) and transportability of data and source files between commonly used formats.
- The *HP-UX System Administrator Manual* provides step-by-step instructions for installing the HP-UX Operating System software, explains certain concepts used and implemented in HP-UX, describes system boot and login, and contains the guide for implementing administrative tasks.

Introduction to Native Language Support 1

The features of Hewlett-Packard **Native Language Support (NLS)** enable the applications designer or programmer to adapt applications to an end user's local language needs.

What Is NLS?

A well-written application program manipulates data and presents it appropriately for the users and its own use. Users who are less technically sophisticated benefit from application programs that interact with them in their native language and conform to their local customs. **Native language** refers to the user's first language (learned as a child), such as Finnish, Portuguese, or Japanese. **Local customs** refer to local conventions such as date, time, and currency formats.

Programs written with the intention of providing a friendly user interface often make assumptions about the user's local customs and language. Program interface and processing requirements vary from country to country; sometimes even within a country. Much existing software does not take this into account, making it appropriate for use only in the country or locality for which it was originally written.

The solution to this problem is to design application programs that can be easily localized. **Localization** is the process of adapting a software application or system for use in different countries or local environments. In many cases, a user's native language or data processing requirements may differ dramatically from those in the environment of the software developer. Traditionally, localization has been achieved by modifying a program for each specific country. Applications that have been designed with localization in mind provide a better solution. Localization can then be accomplished with little or no modification of tables and language-dependent features which are totally independent of the compiled code.

An applications designer must write the application program with built-in provisions for localization. Functions that vary with local language or custom cannot be hard-coded. For example, all messages and prompts must be stored in an external file or catalog. Character comparisons and **upshifting** (using the `shift` key, on most keyboards, to get uppercase characters) must be accomplished by external system-level routines or instructions. External files and catalogs can then be translated, and the program localized without rewriting or recompiling the application program.

Native Language Support (NLS) provides the tools for an applications designer or programmer to produce localizable applications. These tools may include architecture and peripheral support, as well as software facilities within the operating systems and subsystems. NLS addresses the internal functions of a program (such as sorting) as well as its user interface (which includes displayed messages, user inputs, and currency formats.)

Scope of Native Language Support

NLS facilities allow application programs to be designed and written with a local language interface for the end user and for locally correct internal processing. The end user then interacts with localized programs produced by applications programmers who have used NLS tools to write the applications.

For the programmer, the interface has not changed. Most HP-UX interfacing, subsystems, programmer productivity tools, and compilers have not been localized. Applications programmers may still use American English to interact with HP-UX and its subsystems. For example, it is possible to write a complete local language application program using C, but the C compiler retains the English-like characteristics. For example, C key words such as *main, if, while*, and *printf* are still in English.

Aspects of NLS Support

The following aspects of native language support are included in HP-UX software. These three aspects, **Character Set Support**, **Local Customs**, and **Messages**, describe the extent of localization of an application. The applications programmer should consider each aspect carefully when creating software that is language independent.

Character Set Support

A major NLS objective is to provide the capabilities for adapting character sets and sequences to local language needs. This takes into account that character code size determines the maximum number of distinct characters contained in a set. The default set is 7-bit ASCII character set; all programs not localized use this character set. 7-bit ASCII is sufficient to span the Latin alphabet used in many European Languages including upper- and lowercase, punctuation, and special symbols.

The 8th bit of a character byte is normally never stripped or modified. So Hewlett-Packard has defined character sets with bytes in the range 0 to 255 for foreign languages instead of ASCII's 0 to 127. Using the extra bit allows expansion to support European languages that have additional characters, accented vowels, consonants with special forms and special symbols. (See *roman8(7)*.) This 8-bit character code handles the phonetic Japanese **Katakana** character set and others. (See *kana8(7)* and the section on *Supported Native Languages and Character Sets.*)

For languages with larger character sets, such as **Kanji** (the Japanese ideographic character set based on Chinese), 16-bit character codes are required. NLS does not presently offer 16-bit character sets.

All sorting, shifting and type analysis of characters is done according to the local conventions for the native language selected. While the ROMAN8 character set has uppercase and lowercase for most alphabetic characters, some languages discard accents when characters are shifted to uppercase. European French discards accents while Canadian French does not. If there is no notion of **case** in the underlying language (such as Katakana) alphabetic characters are not shifted at all.

Each language uses its own distinct **collating sequences** (the sequence in which characters acceptable to the computer are ordered). The ASCII collation order is actually not even adequate for American dictionary usage. Different languages sort characters from the ROMAN8 set in different orders. For example, Spanish requires character pairs such as "ch" and "ll" to be sorted as single characters. Therefore, "ch" falls at the end of the sorted pairs "cg", "ci", and "cz"; and "ll" similarly falls after "lk", "lm", and "lz". Certain **ideographic** character sets, which represent ideas by graphic symbols, can have multiple orderings. An instance of this is Japanese ideograms (use of graphic symbols to represent Kanji) which can be sorted in phonetic order; based on the number of strokes in the ideogram; or according, first, to the radical (root) of the character and, second, to the number of strokes added to the radical.

On the subject of directionality, the assumption that displayed text goes from left to right does not hold for all languages. Some Middle Eastern languages such as Hebrew go from right to left; while some Far Eastern languages use vertical columns, starting from the right.

Local Customs

Some aspects of NLS relate more to the local customs of a particular geographic area. These aspects, even when supported by a common character set, change from region to region. Consequently, date and time, number, currency information, and so on are presented in a way appropriate to the user's language. For instance, although Great Britain, the United States, Canada, Australia, and New Zealand share the English language, other aspects of data representation differ according to local custom.

The representation of numbers, variations in the symbol indicating the radix character (period in the U.S.), modification of the digit grouping symbol (comma in the U.S.), and the number of digits in a group (three in the U.S.), are all based on the user's native customs. For example, the United States and France both represent currency using decimals and commas, but the symbols are transposed (2,345.77 vs. 2.345,77).

Currency units and how they are subdivided vary with region and country. The symbol for a currency unit can change as well as the symbols placement. It can precede, follow, or appear within the numeric value. Similarly, some currencies allow decimal fractions while others use alternate methods for representing smaller monetary values.

Computation and proper display of time, 24 versus 12-hour clocks, and date information must be considered. The HP-UX system clock runs on Greenwich Mean Time (GMT). Corrections to local time zones consist of adding or subtracting whole or fractional hours from GMT. Some regions, instead of using the common Gregorian calender system, number (or name) the years based upon seasonal, astronomical, or historical events. For example, in Arabic, time of day is measured from the previous sunset; in India, the calendar is strictly lunar (with a leap month every few years); in Japan years are based upon the reign of the emperor.

Names for days of the week and months of the year also varies with language. Abbreviations can be other than three characters or disallowed. Ordering of the year, month, and day, as well as the separating delimiters, is not universally defined. For example, October 7, 1986 would be represented as *10/7/1986* in the U.S., *7.10.1986* in Germany, and *1986/10/7* in Japan.

Chapter 3: Programming With NLS describes the library routines used to access these features.

Messages

The need to make messages readable by users is perhaps the most significant justification for implementing Native Language Support. The user can choose the language for prompts, response to prompts, error messages, and mnemonic command names at run time. Thus it is not necessary to recompile source code when a user in yet another country decides he or she wants translated messages. Keep in mind the syntax of another language may force a change in the structure of the sentence if messages are built in segments (using *printf(3S)*). For example, in German, *"output from standard out and file"* becomes *"Aus und sammlung aus dem standarden ausgabe"*, which translates literally to *"out and file from standard output."*
To do this, user messages must be put in a **message catalog** from which they are retrieved by special library calls. *Chapter 4: Message Catalog System* explains how to create and access message catalogs.

Example: a **fully localized** version of *pr* would

- never strip the 8th bit of a character code
- properly format the date in each page header
- use the message catalog system to select user error messages

Pre-localized Commands

Pre-localization is program modification that makes use of language-dependent library routines not limited to 7-bit character processing. These routines are enhanced to ensure the proper handeling of 8-bit data.

Localization consists of taking the **pre-localized command** and adding the necessary message catalogs and tables to make it run in a particular language (such as French).

Pre-localization allows the message catalogs and tables to be specified at run time, rather than having the information hard-coded and compiled into the commands.

A **localized message file** contains messages in the desired native language. Some HP-UX commands have been enhanced to check for localized message files.

To pre-localize source code, original commands are replaced by commands that incorporate NLS prior to compilation of the program source code. These pre-localized commands are listed in *Appendix A: Pre-localized Commands.*

Supported Native Languages and Character Sets

The NLS system is based on 15 native languages and 3 character sets. These character sets are built into the operating system. Tables and files associated with supported languages will be available through Hewlett-Packard sales offices.

Within NLS, each supported language is associated with a 7-bit or 8-bit character set (one character set may support several languages). Before the introduction of NLS, the only widely-supported character set was ASCII, a 128-character set designed to support American English text. ASCII uses only seven bits of an 8-bit byte to encode each character. The eighth or high order bit is usually zero, except in some applications where it is used for other purposes. For this reason, ASCII is referred to as a "7-bit" code.

8-Bit Character Sets

An 8-bit byte can contain any of 256 unique values, making it is possible to build supersets of ASCII which permit encoding and manipulation of characters required by languages other than American English. These supersets are referred to as **8-bit compatible** or **extended** character sets. These sets have five distinct ranges: 0 to 31 and 127 are **control codes**; 32 is **space**; 33 to 126 are **printable characters**; 128 to 160 and 255 are **extended control characters**; and 161 to 254 are **extended printable characters** (see Table 1.1.) New printable characters are added by defining code values in the range 161 to 254.

Table 1.1 8-bit Character Set Structure

			COL BIT	8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	ROW BIT			5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
4	3	2	1		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	CONTROL CODES (0-31)		SP						EXTENDED CHARACTERS (128-160)							
0	0	0	1	1																
0	0	1	0	2																
0	0	1	1	3				USASCII GRAPHIC (printable) CHARACTERS (33-126)								EXTENDED PRINTABLE CHARACTERS (161-254)				
0	1	0	0	4																
0	1	0	1	5																
0	1	1	0	6																
0	1	1	1	7																
1	0	0	0	8																
1	0	0	1	9																
1	0	1	0	10																
1	0	1	1	11																
1	1	0	0	12																
1	1	0	1	13																
1	1	1	0	14																
1	1	1	1	15								127								255

NLS supports two 8-bit character sets: ROMAN8 (see Table 1.2) and KANA8 (see Table 1.3)

Table 1.2 ROMAN8 Character Set

				COL BIT 8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
ROW BIT				5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
4	3	2	1		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	NU	DL	(SP)	0	@	P	`	p				‾	â	Å	Á	Þ
0	0	0	1	1	SH	D1	!	1	A	Q	a	q			À		ê	î	Ã	þ
0	0	1	0	2	SX	D2	"	2	B	R	b	r			Â		ô	Ø	ã	
0	0	1	1	3	EX	D3	#	3	C	S	c	s			È	°	û	Æ	Đ	
0	1	0	0	4	ET	D4	$	4	D	T	d	t			Ê		á	å	đ	
0	1	0	1	5	EQ	NK	%	5	E	U	e	u			Ë	ç	é	í	Í	
0	1	1	0	6	AK	SY	&	6	F	V	f	v			Î	Ñ	ó	ø	Ì	–
0	1	1	1	7	BL	EB	'	7	G	W	g	w			Ï	ñ	ú	æ	Ó	¼
1	0	0	0	8	BS	CN	(	8	H	X	h	x			´	¡	à	Ä	Ò	½
1	0	0	1	9	HT	EM	)	9	I	Y	i	y			`	¿	è	ì	Õ	ª
1	0	1	0	10	LF	SB	*	:	J	Z	j	z			^	¤	ò	Ö	õ	º
1	0	1	1	11	VT	EC	+	;	K	[	k	{			¨	£	ù	Ü	Š	«
1	1	0	0	12	FF	FS	,	<	L	\	l	|			~		ä	É	š	■
1	1	0	1	13	CR	GS	-	=	M	]	m	}			Ù	§	ë	ï	Ú	»
1	1	1	0	14	SO	RS	.	>	N	^	n	~			Û		ö	ß	Ÿ	±
1	1	1	1	15	SI	US	/	?	O	_	o				£		ü		ÿ	

Table 1.3 KANA8 Character Set

COL BIT																	
	8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
ROW BIT	5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
4 3 2 1		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0 0 0 0	0	NU	DL		0	@	P	`	p				ｰ	ﾀ	ﾐ		
0 0 0 1	1	SH	D1	!	1	A	Q	a	q			｡	ｱ	ﾁ	ﾑ		
0 0 1 0	2	SX	D2	"	2	B	R	b	r			｢	ｲ	ﾂ	ﾒ		
0 0 1 1	3	EX	D3	#	3	C	S	c	s			｣	ｳ	ﾃ	ﾓ		
0 1 0 0	4	ET	D4	$	4	D	T	d	t			､	ｴ	ﾄ	ﾔ		
0 1 0 1	5	EQ	NK	%	5	E	U	e	u			･	ｵ	ﾅ	ﾕ		
0 1 1 0	6	AK	SY	&	6	F	V	f	v			ｦ	ｶ	ﾆ	ﾖ		
0 1 1 1	7	BL	EB	'	7	G	W	g	w			ｧ	ｷ	ﾇ	ﾗ		
1 0 0 0	8	BS	CN	(	8	H	X	h	x			ｨ	ｸ	ﾈ	ﾘ		
1 0 0 1	9	HT	EM	)	9	I	Y	i	y			ｩ	ｹ	ﾉ	ﾙ		
1 0 1 0	10	LF	SB	*	:	J	Z	j	z			ｪ	ｺ	ﾊ	ﾚ		
1 0 1 1	11	VT	EC	+	;	K	[	k	{			ｫ	ｻ	ﾋ	ﾛ		
1 1 0 0	12	FF	FS	,	<	L	¥	l	|			ｬ	ｼ	ﾌ	ﾜ		
1 1 0 1	13	CR	GS	-	=	M	]	m	}			ｭ	ｽ	ﾍ	ﾝ		
1 1 1 0	14	SO	RS	.	>	N	^	n	~			ｮ	ｾ	ﾎ	ﾞ		
1 1 1 1	15	SI	US	/	?	O	_	o				ｯ	ｿ	ﾏ	ﾟ		

NLS 8-bit character sets support all ASCII characters (with the exception that the graphic for back slash (“\”) in KANA8 is yen (“¥”)) in addition to the characters needed to support several Western European-based languages and Katakana. More character sets will be implemented in the future.

The use of 8-bit character sets for NLS implies that in character data, all bits of every byte have significance. Application software must take care to preserve the eighth (high order) bit and not allow it to be modified or reused for any special purpose. Also, no differentiation should be made between characters having the eighth bit turned off and those with it turned on, because all characters have equal status in any extended character set.

Peripherals play a key role in a system's ability to represesent a particular language. Sometimes, even within a single document, several character sets are needed. For example, this document's tables needed line drawing characters; another section contains French and Arabic examples; while the technical section uses mathmatical symbols. Hewlett-Packard peripherals (generally) use the above model to handle multiple character sets (see Figure 1.1).

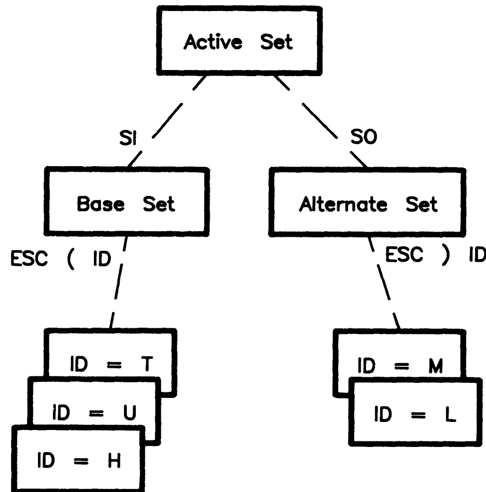


Figure 1.1 8-bit Character Set Support Model

The **Active Set** is the one printed, plotted, or displayed on the terminal. ${}^{S}_{I}$ (shift in) and ${}^{S}_{O}$ (shift out) characters are used to invoke or activate the **Base** or **Alternate** character set. The Base Set is the language-oriented set while the Alternate Set is for special symbols. The escape sequences ${}^{E}_{C}$(*ID* and ${}^{E}_{C}$) *ID* are used to designate, from the collection of available character sets, the Base and Alternate Set. *ID* designates **ID Field** in this context; see Table 1.4 for a table of example character sets with their ID Field number. All sets in this model are 8-bit character sets.

Table 1.4 Character Set ID Numbers

8-bit Character Set Name	ID Field
Start up Base/Default Set	@
Greek8 Character Set	8 B
Hebrew8 Character Set	8 D
Kana8 Character Set	8 H
Line Draw8 Character Set	8 L
Math/Special Symbol8 Set	8 M
Turkish8 Character Set	8 T
Roman8 Character Set	8 U
Arabic8 Character Set	8 V

Native Languages

Each supported native language is based on one of the three character sets. They consist of several language-dependent characteristics defined in various tables and accessed by C library routines and HP-UX commands. These characteristics include rules on upshifting, downshifting, date and time format, currency, and collating sequence.

Hewlett-Packard has assigned a unique **language name** and **language number** to each language included in NLS (see Table 1.5). In some cases, Hewlett-Packard has introduced more than one **supported language** corresponding to a single **natural language**. For example, NLS supports both French (language number 7) and Canadian-French (language number 2) because upshifting is handled differently in French and Canadian-French.

Each of the supported languages can also be considered a **language family** which is applicable in several countries. German (language number 8), for example, can be used in Germany, Austria, Switzerland, and any other place it is requested.

In addition to the native languages supported, an artifical language, native-computer (language number 0), represents the way the computer dealt with language before the introduction of NLS. Whenever language number 0 is used in a native language function, the result is identical to that of the same function performed before the introduction of NLS. NLS library calls with the language parameter equal to 0 will always work correctly, even when no native languages have been configured on the system.

Table 1.5 Supported Native Languages and Character Sets

Language Num	Abbreviation	Language Name
00	n-computer	native computer
01	american	american
02	c-french	canadian french
03	danish	danish
04	dutch	dutch
05	english	english
06	finnish	finnish
07	french	french
08	german	german
09	italian	italian
10	norwegian	norwegian
11	portuguese	portuguese
12	spanish	spanish
13	swedish	swedish
14-40		reserved
41	katakana	katakana
42-80		reserved

Native Language Support 2

File Hierarchy

A set of directories and files has been added to HP-UX in which the NLS tools and language-dependent entities, such as message catalogs and shift tables, reside.

Pre-localized HP-UX commands and C library routines for NLS are in standard directories (*/bin*, */usr/bin*, and */usr/lib*), but there are some special directories and files for NLS language-dependent features.

- The language configuration file, */usr/lib/nls/config*, is a file containing all the native languages that can be configured into a system. Your system has a table like this:

```
00 n-computer
01 american
02 c-french
03 danish
04 dutch
05 english
06 finnish
07 french
08 german
09 italian
10 norwegian
11 portugese
12 spanish
13 swedish
41 katakana
```

  Your computer is always configured for *native-computer*, language number 0 (see Table 1.5). The presence of the actual resources corresponding to each language will vary with the system. This file is used by *langinfo* routines; it must be updated before pre-localized commands can work correctly.

- The following directories are of the form */usr/lib/nls/$LANG* where *$LANG* is a native language (such as *american*).

  /usr/lib/nls/$LANG/collate8 contains the collating sequence for a given language.

  /usr/lib/nls/$LANG/ctype contains information on character set type for the language *$LANG*.

  /usr/lib/nls/$LANG/info.cat contains language-dependent information used by *langinfo*.

  /usr/lib/nls/$LANG/shift has shift tables (uppercase to lowercase or vice-versa).

Configuring Native Languages

To use a language other than *native-computer* (the default lanugage on HP-UX) you must purchase the support software for the optional language and update the environment accordingly.

Installation of Optional Languages

Native Language Support (NLS) comes with only *native-computer* as a language. Other languages (such as German) must be ordered as an option from your Hewlett-Packard sales office.

A **language** includes the tables needed for collating, upshifting, downshifting, character type, language information, and message catalogs. The three character sets already present are standard in HP-UX; only the language tables are optional. Not all character sets are supported on all peripherals, so peripherals which support the desired character set must be obtained.

To install:

- Perform the actual installation using the *optinstall* command, as explained in the chapter of the *HP-UX System Administrator's Manual* entitled *The System Administrator's Toolbox.*
- *Optinstall* automatically installs the language support files in the correct directory as described in the previous section *File Hierarchy.*

After a language has been installed, language-specific information provided by NLS can be used by any application program requesting it.

Environment Changes

To support HP-UX NLS, changes to the user environment within HP-UX were needed. One new environment variable **LANG** (LANGuage) was created and **TZ** (Time Zone) was modified. TZ allows input about different time zones while LANG specifies the language you want to use.

LANG

LANG is a the environment variable that must be set to the native language you desire. LANG contains the language name in American English. It is used to select the character set, lexical order, upshift and downshift tables, and other conventions that vary with language and locality. LANG can be set in */etc/profile* as a default native language, or it can be set by any individual user in *.profile* or *.login.* For *.profile* use:

```
LANG = american
export LANG
```

For *.login* use:

```
setenv LANG american
```

If LANG is not set, all programs using LANG default to the native computer language.

TZ

TZ is a variable that holds time zone information. TZ has been changed to allow fractional offsets from GMT (Greenwich Mean Time). Specification of daylight savings time is taken into account as well as name differences and starting and ending date differences.

Accessing NLS Features

On HP-UX, the use of NLS features is optional. These features must be requested by the applications programmer through library calls or interactively by the user through a localized HP-UX command. The C library routines used for NLS can also be accessed from Pascal and FORTRAN. A description of how to access C library routines from Pascal and FORTRAN is documented in the *HP-UX Portability Guide*.

NLS HP-UX Commands

There are several HP-UX commands that were created specifically to access the message catalog features. They are described in detail in the *Chapter 4: Message Catalog System*.

- *findstr* - find strings in programs not previously localized for inclusion in message catalogs.
- *gencat* - generate a formatted message catalog file.
- *insertmsg* - use *findstr* output to insert calls to *getmsg*.
- *findmsg* - extracts strings from pre-localized C programs for inclusion in message catalogs.
- *dumpmsg* - reverse the effect of *gencat*; take a formatted message catalog and make a modifiable message catalog source file.

Library Support for NLS

There are several C library routines that access the language tables and message catalogs (see *Appendix B: Native Language Support Library*). These are documented in *Chapter 3: Programming With Native Language Support*.

Programming With Native Language Support 3

This chapter describes the NLS header files and the C library routines that are used by Native Language Support (NLS). Two example programs are also provided.

NLS Header Files

There are three header files in */usr/include* specific to NLS: *msgbuf.h, nl_ctype.h,* and *langinfo.h.*

Library Routines

Most NLS library routines have counterparts within the standard HP-UX system. These routines produce similar results; but, instead of assuming particular formats, they use additional parameters to format information how the user prefers to see it.

NLS Library routines are listed below. Routines that have counterparts in the standard C library are mentioned, but not described in detail. Other NLS routines that were added to the C library are described in more detail. Manual pages for all these routines are included in the *HP-UX Reference.* NLS routines are discussed in this chapter in the same sequence as in the *HP-UX Reference*, Section 3.

Convert Date/Time to String

nl_ctime, nl_asctime

Syntax

```
nl_ctime(clock, format, langid)
nl_asctime(tm, format, langid)
```

The *nl_ctime* command extends the capabilities of *ctime* in two ways. First the *format* specification allows the date and time to be output in a variety of ways. *format* uses the field descriptors defined in *date(1).* If *format* is the null string, the *D_T_FMT* string defined by *langinfo(3C)* is used. Second *langid* provides month and weekday names (when selected as alphabetic by the format string) to be in the user's native language. The *nl_asctime* command is similar to *asctime*, but like *nl_ctime* allows the date string to be formatted and the month and weekday names to be in the user's native language. However, like *asctime*, it takes *tm* as its argument. See also *ctime(3C).*

Convert Floating Point to String

nl_gcvt

Syntax

```
nl_gcvt(value, ndigit, buf, langid)
```

The *nl_gcvt* command differs from *gcvt* only in that it uses *langid* to determine what the radix character should be. If *langid* is not valid, or information for *langid* has not been installed, the radix character defaults to a period.
See *ecvt(3C)*.

C Routines to Translate Characters

nl_conv(3C)

This manual page includes *nl_toupper* and *nl_tolower.*

Syntax

```
nl_toupper(c,langid)
nl_tolower(c,langid)
```

These routines are similar to the routines in *conv(3C).* They function the same way, but use a second parameter whose value is expected to be one of the values defined in *langid(7).* If *langid* has not been installed or if shift information for *langid* has not been installed, *toupper* and *tolower* is used for characters below 127, while characters 127 and above are returned unchanged (*toupper* and *tolower* are used with ASCII character set only).
See also *conv(3C).*

C Routines That Classify Characters

nl_ctype(3C)

This manual page includes *nl_isalpha, nl_isupper, nl_islower, nl_isalnum, nl_ispunct, nl_isprint,* and *nl_isgraph.* These routines classify the characters by using the tables in */usr/lib/nls.*

Syntax

All these routines have the same parameter list:

routine`(c, langid)`

where *routine* is any of the routines in *nl_ctype.*

nl_isalpha	*c* is a letter
nl_isupper	*c* is an upper case letter
nl_islower	*c* is a lower case letter
nl_isalnum	*c* is an alphanumeric (letter or digit)
nl_ispunct	*c* is a punctuation character (neither control nor alphanumeric)
nl_isprint	*c* is a printing character
nl_isgraph	*c* is a printing character, like *nl_isprint* except false for space

These routines classify character-coded integer values by table lookup. The command *langid* is as defined in *langid(7)*. Each returns non-zero for true, zero for false. All are defined for the range -1 to 255. If *langid* is not defined or if type information for that language is not installed, *isalpha, isupper,* etc. from *ctype(3C)* is used, returning 0 for values above 127.

If the argument to any of these routines is not in the domain of the function, the result is undefined.

Get Message From Catalog

getmsg(3C)

This added routine is used to retrieve a message from a message catalog.

Syntax

```
getmsg(fd, set_num, msg_num, buf, buflen)
```

where *fd* is the file descriptor pointing to the catalog (file) containing the messages, *set_num* is the set number designating a group of messages in the catalog, *msg_num* is the message number within that set, *buf* is the character array that will hold the returned message, and *buflen* is the number of bytes of the message that can be put into *buf*. The function itself returns a pointer to the character string in *buf*. If *fd* is invalid or *set_num* or *msg_num* are not in the catalog, it returns a pointer to an empty (null) string .

Information on User's Native Language

langinfo(3C)

This includes the routines *langinfo, langtoid, idtolang* and *currlangid*. The command *langinfo* retrieves a null-terminated string containing information unique to a language or cultural area.

Syntax

```
langinfo(langid, item)
langtoid(langname)
idtolang(langid)
currlangid()
```

where *langid* is language information and *item* can be one of the following:

```
D_T_FMT - string for formatting date(1), nl_ctime, and nl_asctime.

DAY_1   - "Sunday" in English
...
DAY_7  - "Saturday" in English
MON_1  - "January"
...       ...
MON_12 - "December"
RADIXCHAR - "decimal point" (',' on the European Continent)
THOUSEP   - separator for thousands
YESSTR    - affirmative response for [y/n] questions
NOSTR     - negative response for [y/n] questions
CRNCYSTR  - symbol for currency preceded by '-' if it precedes the
            number, '+' if it follows the number.
            e.g. "-f" for Dutch, "+ Kr" for Danish.
```

The *idtolang* command takes the integer *langid* and returns the corresponding character string (language name) defined in *langid(7)*. If *langid* is not found, an empty string is returned. The command *langtoid* is the reverse of *idtolang*. The *currlangid* command looks for a LANG string in the user's environment. If it finds it, it returns the corresponding integer (language number) listed in *langid(7)*. Otherwise it returns 0 to indicate a default to ASCII *native-computer*.

Print Formatted Output With Numbered Arguments

printmsg(3C)

This manual page includes *printmsg*, *fprintmsg* and *sprintmsg*, which are derived from their counterparts in *printf(3S)*.

Syntax

```
printmsg (format [ , arg ] ... )
fprintmsg (stream, format [ , arg ] ... )
sprintmsg (s, format [ , arg ] ... )
```

The conversion character *%* used in *printf* is replaced by the sequence *%digit$*, where *digit* is a decimal digit *n* in the range 1-9. The conversion should be applied to the *n*th argument, rather than to the next unused one (you specify which parameter you want this conversion applied to). All other aspects of formatting are unchanged. All conversions must contain the *%digit$* sequence, and it is the user's responsibility to make sure the numbering is correct. All parameters must be used exactly once.
See also *printf(3S)*.

Example

The following example prints a language-independent date and time format.

```
printmsg(format, weekday, month, day, hour, min);
```

For American usage *format* would be a pointer to the string:

```
"%1$s, %2$s %3$d, %4$d:%5$.2d"
```

producing the output:

```
Sunday, July 3, 10:02.
```

For German usage, *format* would be a pointer to the string:

```
"%1$s, %3$d %2$s %4$d:%5$.2d"
```

which outputs:

```
Sonntag, 3 Juli 10:02.
```

Non-ASCII String Collation

nl_string(3C)

This manual page includes *strcmp8, strncmp8, strcmp16*, and *strncmp16*.

Syntax

```
strcmp8(s1, s2, langid)
strncmp8(s1, s2, n, langid)
strcmp16(s1, s2, file_name)
strncmp16(s1, s2, n, file_name)
```

The command *strcmp8* compares string *s1* and *s2* according to the collating sequence specified by *langid* (the language number). An integer greater than, equal to, or less than *0* is returned, depending on whether the collation of *s1* is greater than, equal to, or less than that of *s2*. If *langid* or the collation sequence file is not installed, the native machine collating sequence is used. Trailing blanks in string *s1* or *s2* are ignored. The command *strncmp8* makes the same comparison but looks at only *n* characters. The *strcmp16* and *strncmp16* commands are similar, but use the 16-bit collating sequence table in *file_name*. There can be one of several tables, so the table *file_name*, must be specified rather than simply sending the value *langid*.
See *nl_string(3C)*.

Convert String to Double Precision Number

nl_strtod,nl_atof

Syntax

```
nl_strtod(str, ptr, langid)
nl_atof(str, langid)
```

The *nl_strtod* and *nl_atof* commands are similar to the standard routines, *strtod* and *atof*, but use *langid* to determine the radix character. If *langid* is not valid, or information for *langid* has not been installed, the radix character defaults to a period.
See also *strtod(3C)*.

Application Guidelines

When writing an application program, do not use hard-coded message statements. Store all messages to the user in a separate message catalog where they can be accessed via NLS library commands. This allows users who prefer other native languages to modify the messages to fit their own needs.

The library routines provided for NLS guarantee correct and standard conversions to formats in all supported native languages. You can also create any formats or tables that are beyond those supported by HP to fit your specific needs.

Example C Programs

Here are two example C programs that show how to use some of the Chapter 3 NLS commands.

Example 1

This C program is representative of changes to *ctime* that adapt it for NLS. The commands *nl_conv(3C), nl_ctype(3C), nl_string(3C), nl_strod* and *nl_atod* are handled in a similar manner.

```
#include <langinfo.h>
main ()
     {
      int langid;
      long timestamp;

      langid = currlangid();

      time(&timestamp);
      printf("%s", ctime(&timestamp));
      printf("%s", nl_ctime(&timestamp,"",langid));
     }
```

The command lines used are:

```
LANG = american
export LANG
cc test_ctime.c -o test_ctime
test_ctime
```

The output is:

```
Tue Feb 26 15:56:34 1990
Tue Feb 26 15:56:34 1990
```

The command lines to change the language to French are:

```
LANG = french
export LANG
test_ctime
```

The output is:

```
Tue Feb 26 15:56:34 1990
Mar 1990 Avr 26 15h56
```

Example 2

This C program uses the *printmsg(3C)* routines to output the same message in a variety of ways.

```
#include <stdio.h>
main()
     {
     char *a = "Hello,";
     char *b = "world!";
     char Buf[100];

     printf("Hello, world!\n");
     printf(%s %s\n", a, b);

     printmsg("Hello, world!\n");
     printmsg("%1$s %2$s\n", a, b);
     printmsg("%2$s %1$s\n", a, b);

     fprintf(stdout, "Hello, world!\n");
     fprintf(stdout, "%1$s %2$s\n", a, b);

     fprintmsg(stdout, "Hello, world!\n");
     fprintmsg(stdout, "%1$s %2$s\n", a, b);
     fprintmsg(stdout, "%2$s %1$s\n", a, b);

     sprintf(buf, "Hello, world!\n");
     printf("%s", buf);
     sprintf(buf, "%s %s\n", a, b);
     printf("%s", buf)

     sprintmsg(buf, "Hello, world!\n");
     printf("%s", buf);
     sprintmsg(buf, "%1$s %2$s\n", a, b);
     printf("%s", buf);
     sprintmsg(buf, "%2$s %1$s\n", a, b);
     printf("%s", buf);
     }
```

The command lines used are:

```
cc test_pmsg.c -o test_pmsg
test_pmsg
```

The output is:

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
world! Hello,
Hello, world!
Hello, world!
Hello, world!
Hello, world!
world! Hello,
Hello, world!
Hello, world!
Hello, world!
Hello, world!
world! Hello,
```

Message Catalog System 4

This chapter explains how localized message files are created and updated, where they are kept, and by what conventions they are named.

Introduction

In order to simplify the localization process, applications programmers should write programs that do not require recompiling of code when they are localized. If the code can remain unmodified, the functionality of an application is not affected when translations are made. This reduces support problems because only one version of the application exists. Also this, minimizes the possibility of introducing additional bugs into the product and reduces the localization time.

Localizable programs use text (prompts, commands, messages) from an external message catalog file. This allows text to be translated (part of the localization process) without modifying program source code or recompiling. If the external message catalog file is inaccessible for any reason (such as accidentally removed, not yet created, or whatever), the internally stored messages written in the original language can be used.

A message catalog system is used to separate strings such as prompts and messages from the main code of a program. This makes it very easy for another country to translate the information and have the program run properly without modifying its source code. The HP-UX message catalog system uses HP-UX commands to help create the catalogs and C library routines to access those catalogs. Message catalog commands work only with the C programming language, but the library routines can be accessed from C, Pascal, and FORTRAN programs.

The message catalog commands are:

- *findstr* - find strings for inclusion in message catalogs
- *gencat* - generate a formatted message catalog file
- *insertmsg* - use *findstr* output to insert calls to *getmsg*

The C library routines specific to message catalogs are:

- *getmsg* - get a message from the catalog
- *printmsg, fprintmsg, sprintmsg* - print formatted output with numbered arguments

The steps an applications programmer would take to simplify the localization process are:

- modify existing programs using *findstr, insertmsg, gencat*
- maintain modified programs using *findmsg, gencat*
- translate message files using *dumpmsg, gencat*

Creating a Message Catalog

To make a program easier to localize, string literals such as the error messages and prompts should be placed in a separate file that is accessed by the program at run time. (Hard-coded messages can be left in; they are useful in source for clarifying code.) This way a program can easily access any localized message file without modification of the program. Hewlett-Packard has developed a set of tools to extract print statements from C programs. This set of tools is referred to as the **Message Catalog System**.

Preview: Incorporating NLS into Commands

The general flow of the message catalog system is diagrammed in Figure 4.1. The three HP-UX commands: *findstr, insertmsg,* and *gencat* extract messages from C programs and build a message catalog. The filenames are *prog.c, prog.str, prog.msg,* and *prog.cat.* (They can be named anything you prefer. Names, discounting the *.c* suffix, should be equal to or less than 9 characters in length. The suffixes used here are only a suggested naming convention.)

The name *prog.c* represents any C program containing hard-coded messages. The name *prog.str* represents an intermediate file containing all strings from the source file surrounded by double quotes (" "). The new C program is named *nl_prog.c* (where *prog.c* is the original C program) with references made to a message file instead of hard-coded messages. The final object file produced by compiling *nl_prog.c* is *prog.* The file *prog.msg* contains the numbered messages and sets that are used to generate the final message file. The final message file is *prog.cat.*

Following the Flow

The next sections describe in detail the steps used when creating a message catalog (see Figure 4.1).

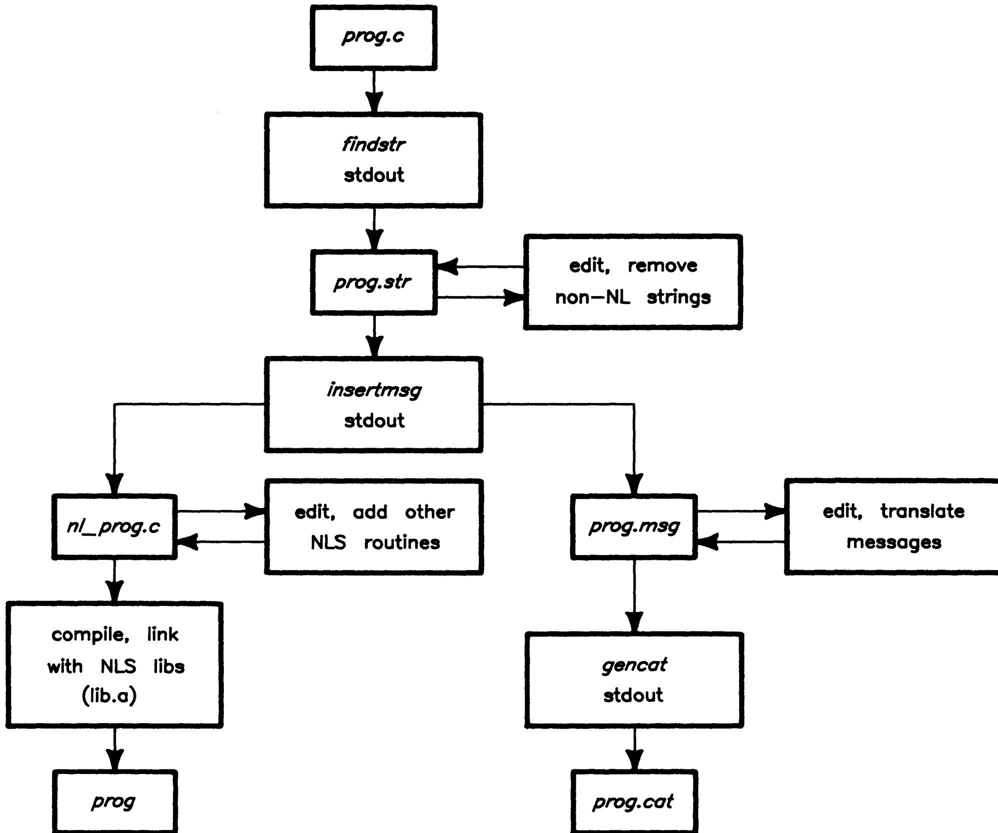


Figure 4.1 Flow of the Message Catalog

findstr

findstr examines files of C source code (*prog.c* in this case) for string constants that do not appear in comments. These strings, along with the surrounding quotes, are placed on standard output. Each extracted string is preceded by the file name, start position in the file, and string length. The output should be redirected to a file for editing.

Syntax

```
findstr prog.c > prog.str
```

prog.str

prog.str, the output from *findstr* which is created when the user redirects output from *findstr* into a file, contains all quoted strings that do not appear in comments from the C program (*prog.c*) used as input to *findstr*. This includes error messages, format statements, system calls, and anything else that is surrounded in double quotes. Preceding the strings is a copy of the filename (*prog.c*), from which the strings came, followed by the *byteposition* and *bytecount*. The file *prog.str* can be called any name. Message files should contain nothing but messages, so you must edit *prog.str* to remove all other types of quoted strings.

Syntax

```
prog.c:byteposition:bytecount:``string''
```

The parameters *byteposition* and *bytecount* apply to the source program at the time *findstr* is run. Any changes to *prog.c* will invalidate these numbers. Do not modify these parameters.

insertmsg

insertmsg uses *prog.c* and *prog.str* to create both the new C source file (*nl_prog.c*) and a file (*prog.msg*) containing the messages for translation into local languages. *prog.msg* is used by *gencat*.

Syntax

```
insertmsg prog.str > prog.msg
```

Here, *prog.str* is the edited output from *findstr* (see above section on *prog.str*). The routine *insertmsg* creates a new file (*nl_prog.c*), for each file named in *prog.str*. For this example, all the lines in *prog.str* refer to *prog.c*.

These lines:

```
#ifdef NLS
#define NL_SETN   1
#include <msgbuf.h>
#else NLS
#define nl_msg(i, s) (s)
#endif NLS
```

are inserted by *insertmsg* at the beginning of each new file (in this case *nl_prog.c*). Then for each line in *prog.str*, it surrounds the string with an expression of the form:

```
nl_msg(1, "Hello, world\n");
```

where *1* is the message number.

This is expanded at run time by a macro in *msgbuf.h*. Then *insertmsg* places a file on the standard output that can be used as input to *gencat* (see section below on *prog.msg*). If *insertmsg* doesn't find the opening or closing double quote where it expects it in *prog.str*, it prints "`insertmsg exiting : lost in strings file`" and dies. If this happens check the strings file to make sure that the lines kept there haven't been altered. Re-running *findstr* on *prog.c* reconstructs *prog.str* to its unedited form.

output from insertmsg

There are two branches from *insertmsg*: the new ".c" file (*nl_prog.c*) and the messages going to **stdout** (assumedly redirected into a file, referred to here as *prog.msg*).

nl_prog.c

This is the new source of your program. It consists of all the source in the original program, with the messages in *prog.str* changed to be of the form shown above, and an additional *#define* and *#include* statement at the beginning of the file.

The programmer must now hand edit the file *nl_prog.c* to insert a call

```
#ifdef NLS
nl_catopen("prog");
#endif NLS
```

where *prog.cat* is the final message file (*.cat* will be appended to *prog* by the *nl_catopen* macro). If a set number other than *1* is desired (for merging several message catalog files, separating them by set number only) change the *NL_SETN* define statement accordingly.

prog.msg

This is what *insertmsg* places on **stdout** to be used as the input to *gencat*. This file needs to be hand edited to define the *$set* number to match the *NL_SETN* in *nl_prog.c*. Messages in this file are automatically numbered from *1* upward, in the same order as they appear in the file *prog.str*. The same number will also be placed in the call to *nl_msg* (the macro placed around the message by *insertmsg*).

Example

```
$set 1
1 Good morning
2 error, monday morning
$set 2
15 Hello, world!
16 Thank goodness its Friday!!
17 CRASH
```

gencat

gencat generates a formatted message catalog (*prog.cat*) from the information in *prog.msg*.

Syntax

```
gencat prog.cat prog.msg ...
```

The *prog.msg* file consist of sets of messages along with comments and are merged into a formatted file (*prog.cat*) that can be accessed by *getmsg*. If *prog.cat* does not exist, it will be created. If it exists, its messages are included in the new *nl_prog.c* unless the set and message numbers collide, in which case the new supersedes the old. See the section on *prog.msg* for details on the input file format. If a message source line has a number but no text then the existing message corresponding to this number is deleted from the catalog.

To delete an entire message set, place the directive

```
$DELSET set_number
```

at the beginning of a line between sets.

prog.cat

prog.cat is the final message catalog, created by *gencat*, which is then accessed by the new source program. *gencat* is a binary file and cannot be read directly by a user.

The file *prog.cat* will be stored as */usr/lib/nls/american/prog.cat* where *american* is the value of LANG when this file is accessed and "prog" is the program name string entered by hand into the *nl_catopen* statement. You must be logged in as super user to place the file in that directory.

Multiple commands may share the same physical file or share the same name in the *nl_catopen* macro. Each message catalog name (program name with *.cat* appended) must be linked to the same file. Messages can be distinguished, either by set number or by message number.

prog

prog is the object file produced by compiling *nl_prog.c*. Do not confuse this file with "prog" called by *nl_catopen* that has *.cat* appended.

Format of Source Message File

The source message catalog consists of the following lines:

$set n *comment*

This line, followed by the message text lines, specifies the set number of the following messages until the next *$set, $delset*, or end of file appears. The *n* denotes the set number (1-255). Set numbers must be in ascending order within a single source file. Any string following the set number is treated as comment.

$delset n *comment*

This line deletes an entire message set from the existing catalog file. The *n* denotes the set number (1-255). Any string following the set number is treated as comment.

$ *comment*

This line is used as a comment line.

m *message text*

The *m* denotes the message number (1-32767). If *message text* exists, the message is stored in the catalog file with the set number specified by *$set* and message number *m*. If the *message text* does not exist, the message corresponding to the set number and message number is deleted from the existing catalog file. Message numbers must be in ascending order within a single set.

Certain special characters are used in the text strings; certain non-graphic characters and the backslash "\" can be specified using the following table (Table 4.1) of escape sequences:

Table 4.1 Escape Sequences

Description	Symbol	Sequence
newline	NL(LF)	\n
horizontal tab	HT	\t
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
bit pattern	ddd	\ddd

The escape sequence *ddd* consists of backslash followed by 1, 2, or 3 octal digits which are used to specify the value of the desired character. If the character following a backslash is not one of those specified, the backslash is ignored. Backslash "\" is also used to continue a string to the next line. The following two lines are considered a single string:

```
1 This line continues \
to the next line.
```

which is equivalent to:

```
1 This line continues to the next line.
```

Note that, in this case, backslash "\" must immediately precede the newline character.

Printmsg, Fprintmsg and Sprintmsg

The commands *printmsg, fprintmsg and sprintmsg* are derived from their counterparts in *printf(3S)*, with the understanding that the conversion character *%* is replaced by the sequence *%digit$*. *Digit* is the decimal *n*, in the range 1-9, and indicates that this conversion should be applied to the *n*th argument, rather than to the next unused one. All conversion specifications must contain the *%digit$* sequence, and numbered correctly. All parameters must be used exactly once. These commands are used to handle the message catalog system with the messages that have two or more parameters.

Accessing Applications Catalogs

Message catalogs are accessed from any supported language program, such as C, Pascal, or FORTRAN, using C library routines. These C library routines consist of some new library functions and some altered, pre-existing C library routines.

All HP-UX shell commands and C library routines that are associated with NLS or that have been changed due to NLS are documented in the *HP-UX Reference* and are listed in *Appendix A: Pre-localized Commands* of this manual.

To use the C library routines from a Pascal or FORTRAN program please refer to the *HP-UX Portability Guide.*

File System Organization and Catalog Naming Conventions

Any application that has been localized into several languages will have separate message catalogs (files) for each language. The routine *nl_catopen* assumes the message file will be under */usr/lib/nls/language/filename.cat* where *language* is the the language contained in the LANG environmental variable and *filename* is the name of the file specified in the call to *nl_catopen* in the source program (usually the program name).

The directory */usr/lib/nls* is writable only by root.

For example, original, unlocalized data might be stored in a file whose full path name is */usr/lib/nls/n-computer/prog.cat.* The file */usr/lib/nls/german/prog.cat* would contain the same data modified for German, and */usr/lib/nls/spanish/prog.cat* would contain Spanish data. It is the responsibility of the application program to determine (at run time) which file to open.

Localization

Suppose you have the following C program, *hello.c*, and you want to localize the output. The source file of *hello.c* looks like this:

```
main()
/* This program prints a greeting and the date */
{
printf("hello, world\n");
system("date");
}
```

6 Steps to Localize an Example Program

1. **Execute** *findstr*, redirecting the output to *hello.str*.

   ```
   $findstr hello.c > hello.str
   ```

2. **Edit** *hello.str*. The file *hello.str* contains all the strings from *hello.c* that are surrounded by double quotes. It contains the following lines:

   ```
   hello.c:67:16:"hello, world\n"
   hello.c:93:6:"date"
   ```

   The file *hello.str* needs to be edited so it contains only messages that should appear on the screen. Notice that *date* is enclosed with double quotes, but should **not** be included in the message file. Edit *hello.str* so it contains only the line:

   ```
   hello.c:67:16:"hello, world\n"
   ```

3. **Execute** *insertmsg*, redirecting output to a file called *hello.msg*.

   ```
   insertmsg hello.str > hello.msg
   ```

   In addition to the messages output to *hello.msg*, *insertmsg* creates the new source file, *nl_hello.c*, which contains the original source plus a new *#define* line and *#include* line, plus an altered message line. Your directory should now contain the following files relating to this example:

   ```
   hello.c     hello.msg     hello.str      nl_hello.c
   ```

4. **Edit** *nl_hello.c*. The file currently looks like:

   ```
   #ifdef NLS
   #define NL_SETN   1 /*set number*/
   #include <msgbuf.h>
   #else NLS
   #define nl_msg(i, s) (s)
   #endif NLS
   main()
   /* This program prints a greeting and the date */
   {
        printf((nl_msg(1, "hello, world\n")));
        system("date");
   }
   ```

   The macro *nl_msg* will be expanded at compile time (see section on *insertmsg*). Both the set number and the message number will be set to *1*.

The file needs to be edited so it refers to the final message file. Decide now what you want to call the final message file (in this example it will be called *hello.cat*) and insert the line:

```
nl_catopen("hello");
```

This line opens a file called *hello.cat* in a directory corresponding to the native language defined in the LANG environmental variable. If LANG is not defined, the hard-coded messages in the source are used. This means that you never need to change the source code. You simply need to change the value of LANG and create a message file stored in */usr/lib/nls/$LANG/hello.cat* if you wish to localize *hello.c* for a new language.

Final source file looks like this:

```
#ifdef NLS
#define NL_SETN   1 /*set number*/
#include <msgbuf.h>
#else NLS
#define nl_msg(i, s) (s)
#endif NLS
main()
/* This program prints a greeting and the date */
     {
     #ifdef NLS
     nl_catopen("hello");
     #endif NLS
     printf((nl_msg(1, "hello,world\n")));
     system("date");
     }
```

5. **Edit** *hello.msg* to define *$set* to match the set number in *nl_hello.c*, if different. It should be the same unless you are creating a message file other than the one created by *insertmsg*. The file *msghello* looks like:

   ```
   $set 1
   1 hello, world\n
   ```

6. **Execute** *gencat* specifying the file *hello.cat* used in step #4 the output file. The input file is *hello.msg*.

   ```
   gencat hello.cat hello.msg
   ```

   This file should be stored as */usr/lib/nls/american/hello.cat.*

You now have a localizable program. If your native language is English, you also have a localized message file. If your native language is something other than English, you still need to localize the message file. Let's say your native language is German, and rather than printing the message "hello, world" to the screen, you wish to print "Guten Tag Welt, wie geht es dir?".

Edit *hello.msg* or create a new file to read:

```
$set  1
1 Guten Tag Welt, wie geht es dir?\n
```

Execute *gencat* by typing in:

```
gencat hello.cat hello.msg
```

Store the new *hello.cat* message file in */usr/lib/nls/german/hello.cat* and change your LANG environment variable to *german.*

When you re-execute the program, it will automatically use the German message file rather than the American English message file. Execute *hello* to verify that it works. If the LANG variable is not defined, or the message catalog does not exist, the hard-coded message will appear.

Pre-localization Commands A

Series 500 HP-UX 5.0 has limited support for users whose native language is other than American English. To provide this support, several HP-UX commands have been enhanced to allow processing of files and keyboard entries which contain 8-bit (256 symbol) characters such as filenames and data. These commands are identified as **8-bit compatible** (pre-localized). They have also been enhanced to generate prompts, text output, and error messages in one of several native languages. The format of output can also be set according to local customs. These commands are identified as **localized**. The table below identifies the commands and the **NLS Level** to which they have been localized (**8-bit** or fully **localized**). (See *HP-UX Reference* Manual pages for further detail.)

Previous HP-UX systems only supported 7-bit (128 symbol) character sets, and fully supported only the ASCII 7-bit set. Commands not listed in the following table (such as *csh* and *vi*) are not supported. Processing 8-bit character data with a 7-bit-only commands yields unpredictable results. Typically the consequence is the 8th bit is stripped off, yielding an arbitrary (but predictable) 7-bit ASCII character code.

Table A.1 NLS-Compatible HP-UX Commands

Name(*)	NLS Level	Description
accept(1M)	localized	allow LP requests
at(1)	8-bit	time schedule a process
aterm(1)	8-bit	general purpose asynchronous terminal emulation
basename(1)	8-bit	extracts portions of path names
cancel(1)	localized	cancel spooler printer output
cat(1)	localized	concatenate and print file
cc(1)	8-bit	c compiler
cdb(1)	localized	c debugger
chmod(1)	8-bit	change file mode (permissions, etc.)
cmp(1)	localized	compare two files
comm(1)	localized	select or reject lines common to two sorted files
cp(1)	localized	copy a file
cpio(1)	8-bit	copy file archives in and out
cron(1)	8-bit	clock daemon
cu(1)	8-bit	call UNIX[1]; terminal emulator
date(1)	localized	print/set the date
diff(1)	localized	differential file comparator
disable(1)	localized	disable a spooled printer
echo(1)	8-bit	echo (print) arguments
ed(1)	localized	(line oriented) text editor
enable(1)	localized	enable a spooled printer
env(1)	8-bit	set environment for command execution
expr(1)	8-bit	evaluates arguments as an expression

[1] UNIX is a trademark of AT&T Bell Laboratories, Inc.
* Number denotes *HP-UX References* manual section.

Table A.1 NLS-Compatible HP-UX Commands (cont.)

Name(*)	NLS Level	Description
fc(1)	8-bit	FORTRAN 77 compiler
f77(1)	8-bit	FORTRAN 77 compiler
find(1)	8-bit	find files
getopt(1)	8-bit	parse command options
lp(1)	localized	line printer spooler
lpadmin(1M)	localized	configure LP spooling system
lpsched(1M)	localized	start LP spooling system
lpshut(1M)	localized	stop LP spooling system
ls(1)	8-bit	list contents of directories
mail(1)	8-bit	send and receive mail
mkdir(1)	8-bit	make a directory
more(1)	8-bit	file browser
mvdir(1)	8-bit	move a directory
newgrp(1)	8-bit	log into new group
passwd(1)	8-bit	change login passwd
pc(1)	8-bit	HP Series 200 Pascal compiler
pc(1)	8-bit	HP Series 500 Pascal compiler
pr(1)	localized	print file(s)
reject(8)	localized	deny LP spooler requests
rmdir(1)	8-bit	remove directories
sh(1)	8-bit	bourne shell command interpreter
sort(1)	localized	sort/merge text files
tar(1)	8-bit	tape file archiver
tee(1)	8-bit	pipe fitting
uniq(1)	localized	report repeated lines in a file
uucp(1)	8-bit	UNIX[1]-to-UNIX[1] copy
uulog(1)	8-bit	maintains summary log of *uucp*
uuname(1)	8-bit	lists the *uucp* names of known systems
wall(1)	8-bit	broadcast message to all users
wc(1)	localized	word/line/byte count
write(1)	localized	interactively write to another user

Native Language Support Library

B

The following library calls have been added to HP-UX to facilitate the development of fully localized programs. These are included in the standard C library */usr/lib/libc.a.*

Table B.1 NLS Library

Name(*)	Description
catread(3C)	adds MPE/RTE filetype support to getmsg
ctime(3C)	time conversion routines
evct(3C)	convert binary numbers to string numerics
nl_conv(3C)	character casefolding routines
nl_ctype(3C)	character classification
getmsg(3C)	get native language message from catalog
langinfo(3C)	get native language information
nl_string(3C)	string comparison routines
printmsg(3C)	print formatted numeric output
strtod(3C)	convert string numeric to binary number routines

Other HP-UX system and library calls are 8-bit compatible, with the following exceptions. Localized versions exist for many of these (see above) and should be used for new program development.

Table B.2 Non-NLS HP-UX System and Library Calls

Name(*)	Description
atof(3C)	convert ASCII string numerics to various binary forms
conv(3C)	ASCII character casefolding routines
ctime(3C)	date and time conversion routines
ctype(3C)	character classification routines
ecvt(3C)	convert binary number to ASCII string numeric
qsort(3C)	quick sort
regex(3C)	regular expression compile/execute
string(3C)	character string operations

Peripheral Configuration

C

European Character Sets

For European languages, many HP peripherals support the Hewlett-Packard ROMAN8 character set. ROMAN8 is a full superset of ASCII and offers 88 additional local language symbols. Older HP peripherals may use the HP **Roman Extension** set, which is a subset of ROMAN8. Roman Extension is missing ROMAN8 Characters À thru Ï, Ù, Û, Ç, ¥, ƒ, ¢, Á thru ±. See Table 1.2, ROMAN8 Character Set.

Japanese Character Sets

Many HP peripherals support an alternate 8-bit character set known as KANA8. The first 128 codes in the KANA8 set are JASCII (same as ASCII except substitutes "¥" for "\") and the last 128 codes are Katakana.

ISO 7-bit Substitution

ISO7 stands for International Standards Organization 7-bit character substitution. For each ISO7 language, certain ASCII character codes infrequently used in ordinary text (such as those for "|" and "{") are designated to generate different local-language symbol (such as "ø" or "æ" in Danish). Unfortunately, the designated ASCII codes represent special characters often used in HP-UX (and all other UNIX and UNIX-like systems). The use of ISO 7-bit substitution is neither recommended nor supported.

Character Set Support by Peripherals

ROMAN8 terminals can simultaneously display any characters in their set. Their keyboards have keycaps only for the specified local language, but you can enter any ROMAN8 character by use of the EXTEND key. You can also use most 8-bit terminals in ISO7 mode (see discussion above).

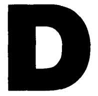
Plotter ROM (internal) fonts are normally used for **draft-quality** plots. **Final** plots are normally done with host-generated (software) vector fonts. DGL/9000 graphics presently generate only ASCII characters.

The following table summarizes the character set support of Series 200/500 peripherals. The **Ordering Information** column indicates what action you must take to obtain a peripheral which is not ASCII.

Table C.1 Peripheral Localization Summary

Peripheral Device	Character Set(s) Support	Ordering Information	Comments
9020A Computer	ASCII only	Keyboard option	
9020B Computer	ASCII only	Keyboard option	
9020C Computer	ASCII only	Keyboard option	
98700H Display Sta.	ASCII only	Product suffix	
HP 110 Terminal	ROMAN8 Std.	Product suffix	
HP 150 Terminal	ROMAN8 Std.	Product suffix	
2392A Terminal	ROMAN8 Std.	Keyboard option	Missing Á thru ±.
2622A Terminal	Roman Ext. Std.	Keyboard option	
2623A Terminal	Roman Ext. Std.	Keyboard option	
2624B Terminal	--------	-----------	Not recommended for NLS
2625A Terminal	ROMAN8 Std.	Keyboard option	
2626A Terminal	Roman Ext. Std.	Keyboard option	
2627A Terminal	Roman Ext. Std.	Keyboard option	
2628A Terminal	ROMAN8 Std.	Keyboard option	
2647F Terminal	ASCII only	NA	
2703A Terminal	Roman Ext. Std.	Keyboard option	
2225A *ThinkJet*(TM)	ROMAN8 Std.	NA	
2563A Printer	ROMAN8 Std.	KANA8 option	
2565A Printer	ROMAN8 Std.	KANA8 option	
2566A Printer	ROMAN8 Std.	KANA8 option	
2601A Printer	Substitution	Accessory	Series 500 only, change print wheel
2602A Printer	Substitution	Accessory	Series 500 only, change print wheel
2608S Printer	Roman Ext.	Option 002	Series 500 only
2631B/G Printer	Roman Ext. Std.	Formerly Option 009	
2671A/G Printer	Roman Ext. Std.	NA	Series 200 only
2673A Printer	Roman Ext. Std.	NA	Series 200 only
2680A Printer	Roman Ext. Std.	NA	Series 500 only
2686A *LaserJet*(TM)	ROMAN8 Std.	NA	
2688A Printer	ROMAN8 Std.	NA	Series 500 only, not all fonts ROMAN8
2932A Printer	ROMAN8/KANA8 Std.	NA	
2934A Printer	ROMAN8/KANA8 Std.	NA	
82906A Printer	ROMAN8 Std	KANA8 option	Series 200 only
97090A Printer	Roman Ext. Std.	NA	Series 500 only
9876A Printer	Roman Ext. Std.	NA	
7470A Plotter	ISO7 only	NA	
7475A Plotter	ISO7 only	NA	
7580A Plotter	ISO7 only	NA	
7585A Plotter	ISO7 only	NA	
7586A Plotter	ISO7 only	NA	

Character Sets



This section provides the table for the following character sets:

- ASCII Character Set
- Roman Character Sets
- Katakana Character Set

Table D.1 ASCII Character Set

ASCII Char.	EQUIVALENT FORMS		HP-IB
	Dec	Binary	
NUL	0	00000000	
SOH	1	00000001	GTL
STX	2	00000010	
ETX	3	00000011	
EOT	4	00000100	SDC
ENQ	5	00000101	PPC
ACK	6	00000110	
BEL	7	00000111	
BS	8	00001000	GET
HT	9	00001001	TCT
LF	10	00001010	
VT	11	00001011	
FF	12	00001100	
CR	13	00001101	
SO	14	00001110	
SI	15	00001111	
DLE	16	00010000	
DC1	17	00010001	LLO
DC2	18	00010010	
DC3	19	00010011	
DC4	20	00010100	DCL
NAK	21	00010101	PPU
SYNC	22	00010110	
ETB	23	00010111	
CAN	24	00011000	SPE
EM	25	00011001	SPD
SUB	26	00011010	
ESC	27	00011011	
FS	28	00011100	
GS	29	00011101	
RS	30	00011110	
US	31	00011111	

ASCII Char.	EQUIVALENT FORMS		HP-IB
	Dec	Binary	
space	32	00100000	LA0
!	33	00100001	LA1
"	34	00100010	LA2
#	35	00100011	LA3
$	36	00100100	LA4
%	37	00100101	LA5
&	38	00100110	LA6
'	39	00100111	LA7
(	40	00101000	LA8
)	41	00101001	LA9
*	42	00101010	LA10
+	43	00101011	LA11
,	44	00101100	LA12
–	45	00101101	LA13
.	46	00101110	LA14
/	47	00101111	LA15
0	48	00110000	LA16
1	49	00110001	LA17
2	50	00110010	LA18
3	51	00110011	LA19
4	52	00110100	LA20
5	53	00110101	LA21
6	54	00110110	LA22
7	55	00110111	LA23
8	56	00111000	LA24
9	57	00111001	LA25
:	58	00111010	LA26
;	59	00111011	LA27
<	60	00111100	LA28
=	61	00111101	LA29
>	62	00111110	LA30
?	63	00111111	UNL

Table D.1 ASCII Character Set (cont.)

ASCII Char.	EQUIVALENT FORMS		HP-IB
	Dec	Binary	
@	64	01000000	TA0
A	65	01000001	TA1
B	66	01000010	TA2
C	67	01000011	TA3
D	68	01000100	TA4
E	69	01000101	TA5
F	70	01000110	TA6
G	71	01000111	TA7
H	72	01001000	TA8
I	73	01001001	TA9
J	74	01001010	TA10
K	75	01001011	TA11
L	76	01001100	TA12
M	77	01001101	TA13
N	78	01001110	TA14
O	79	01001111	TA15
P	80	01010000	TA16
Q	81	01010001	TA17
R	82	01010010	TA18
S	83	01010011	TA19
T	84	01010100	TA20
U	85	01010101	TA21
V	86	01010110	TA22
W	87	01010111	TA23
X	88	01011000	TA24
Y	89	01011001	TA25
Z	90	01011010	TA26
[	91	01011011	TA27
\	92	01011100	TA28
]	93	01011101	TA29
^	94	01011110	TA30
_	95	01011111	UNT

<table>
<tr><th rowspan="2">ASCII Char.</th><th colspan="2">EQUIVALENT FORMS</th><th rowspan="2">HP-IB</th></tr>
<tr><th>Dec</th><th>Binary</th></tr>
<tr><td>`</td><td>96</td><td>01100000</td><td>SC0</td></tr>
<tr><td>a</td><td>97</td><td>01100001</td><td>SC1</td></tr>
<tr><td>b</td><td>98</td><td>01100010</td><td>SC2</td></tr>
<tr><td>c</td><td>99</td><td>01100011</td><td>SC3</td></tr>
<tr><td>d</td><td>100</td><td>01100100</td><td>SC4</td></tr>
<tr><td>e</td><td>101</td><td>01100101</td><td>SC5</td></tr>
<tr><td>f</td><td>102</td><td>01100110</td><td>SC6</td></tr>
<tr><td>g</td><td>103</td><td>01100111</td><td>SC7</td></tr>
<tr><td>h</td><td>104</td><td>01101000</td><td>SC8</td></tr>
<tr><td>i</td><td>105</td><td>01101001</td><td>SC9</td></tr>
<tr><td>j</td><td>106</td><td>01101010</td><td>SC10</td></tr>
<tr><td>k</td><td>107</td><td>01101011</td><td>SC11</td></tr>
<tr><td>l</td><td>108</td><td>01101100</td><td>SC12</td></tr>
<tr><td>m</td><td>109</td><td>01101101</td><td>SC13</td></tr>
<tr><td>n</td><td>110</td><td>01101110</td><td>SC14</td></tr>
<tr><td>o</td><td>111</td><td>01101111</td><td>SC15</td></tr>
<tr><td>p</td><td>112</td><td>01110000</td><td>SC16</td></tr>
<tr><td>q</td><td>113</td><td>01110001</td><td>SC17</td></tr>
<tr><td>r</td><td>114</td><td>01110010</td><td>SC18</td></tr>
<tr><td>s</td><td>115</td><td>01110011</td><td>SC19</td></tr>
<tr><td>t</td><td>116</td><td>01110100</td><td>SC20</td></tr>
<tr><td>u</td><td>117</td><td>01110101</td><td>SC21</td></tr>
<tr><td>v</td><td>118</td><td>01110110</td><td>SC22</td></tr>
<tr><td>w</td><td>119</td><td>01110111</td><td>SC23</td></tr>
<tr><td>x</td><td>120</td><td>01111000</td><td>SC24</td></tr>
<tr><td>y</td><td>121</td><td>01111001</td><td>SC25</td></tr>
<tr><td>z</td><td>122</td><td>01111010</td><td>SC26</td></tr>
<tr><td>{</td><td>123</td><td>01111011</td><td>SC27</td></tr>
<tr><td>|</td><td>124</td><td>01111100</td><td>SC28</td></tr>
<tr><td>}</td><td>125</td><td>01111101</td><td>SC29</td></tr>
<tr><td>~</td><td>126</td><td>01111110</td><td>SC30</td></tr>
<tr><td>DEL</td><td>127</td><td>01111111</td><td>SC31</td></tr>
</table>

Table D.2 Roman Character Set

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
NU	0	00000000
SH	1	00000001
SX	2	00000010
EX	3	00000011
ET	4	00000100
EQ	5	00000101
AK	6	00000110
🔔	7	00000111
BS	8	00001000
HT	9	00001001
LF	10	00001010
VT	11	00001011
FF	12	00001100
CR	13	00001101
SO	14	00001110
SI	15	00001111
DL	16	00010000
D1	17	00010001
D2	18	00010010
D3	19	00010011
D4	20	00010100
NK	21	00010101
SY	22	00010110
EB	23	00010111
CN	24	00011000
EM	25	00011001
SB	26	00011010
EC	27	00011011
FS	28	00011100
GS	29	00011101
RS	30	00011110
US	31	00011111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
	32	00100000
!	33	00100001
"	34	00100010
#	35	00100011
$	36	00100100
%	37	00100101
&	38	00100110
'	39	00100111
(	40	00101000
)	41	00101001
*	42	00101010
+	43	00101011
,	44	00101100
-	45	00101101
.	46	00101110
/	47	00101111
0	48	00110000
1	49	00110001
2	50	00110010
3	51	00110011
4	52	00110100
5	53	00110101
6	54	00110110
7	55	00110111
8	56	00111000
9	57	00111001
:	58	00111010
;	59	00111011
<	60	00111100
=	61	00111101
>	62	00111110
?	63	00111111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
@	64	01000000
A	65	01000001
B	66	01000010
C	67	01000011
D	68	01000100
E	69	01000101
F	70	01000110
G	71	01000111
H	72	01001000
I	73	01001001
J	74	01001010
K	75	01001011
L	76	01001100
M	77	01001101
N	78	01001110
O	79	01001111
P	80	01010000
Q	81	01010001
R	82	01010010
S	83	01010011
T	84	01010100
U	85	01010101
V	86	01010110
W	87	01010111
X	88	01011000
Y	89	01011001
Z	90	01011010
[	91	01011011
\	92	01011100
]	93	01011101
^	94	01011110
_	95	01011111

ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary
`	96	01100000
a	97	01100001
b	98	01100010
c	99	01100011
d	100	01100100
e	101	01100101
f	102	01100110
g	103	01100111
h	104	01101000
i	105	01101001
j	106	01101010
k	107	01101011
l	108	01101100
m	109	01101101
n	110	01101110
o	111	01101111
p	112	01110000
q	113	01110001
r	114	01110010
s	115	01110011
t	116	01110100
u	117	01110101
v	118	01110110
w	119	01110111
x	120	01111000
y	121	01111001
z	122	01111010
{	123	01111011
|	124	01111100
}	125	01111101
~	126	01111110
	127	01111111

Table D.2 Roman Character Set (cont.)

ASCII Char.	EQUIVALENT FORMS		ASCII Char.	EQUIVALENT FORMS		ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary		Dec	Binary		Dec	Binary
	160	10100000	â	192	11000000	Á	224	11100000
À	161	10100001	ê	193	11000001	Ã	225	11100001
Â	162	10100010	ô	194	11000010	ã	226	11100010
È	163	10100011	û	195	11000011	Đ	227	11100011
Ê	164	10100100	á	196	11000100	đ	228	11100100
Ë	165	10100101	é	197	11000101	Í	229	11100101
Î	166	10100110	ó	198	11000110	Ì	230	11100110
Ï	167	10100111	ú	199	11000111	Ó	231	11100111
´	168	10101000	à	200	11001000	Ò	232	11101000
`	169	10101001	è	201	11001001	Õ	233	11101001
ˆ	170	10101010	ò	202	11001010	õ	234	11101010
¨	171	10101011	ù	203	11001011	Š	235	11101011
˜	172	10101100	ä	204	11001100	š	236	11101100
Ù	173	10101101	ë	205	11001101	Ú	237	11101101
Û	174	10101110	ö	206	11001110	Ÿ	238	11101110
₤	175	10101111	ü	207	11001111	ÿ	239	11101111
—	176	10110000	Å	208	11010000	Þ	240	11110000
B1	177	10110001	î	209	11010001	þ	241	11110001
B2	178	10110010	Ø	210	11010010	F2	242	11110010
·	179	10110011	Æ	211	11010011	F3	243	11110011
Ç	180	10110100	å	212	11010100	F4	244	11110100
ç	181	10110101	í	213	11010101	F5	245	11110101
Ñ	182	10110110	ø	214	11010110	—	246	11110110
ñ	183	10110111	æ	215	11010111	¼	247	11110111
¡	184	10111000	Ä	216	11011000	½	248	11111000
¿	185	10111001	ì	217	11011001	ª	249	11111001
¤	186	10111010	Ö	218	11011010	º	250	11111010
£	187	10111011	Ü	219	11011011	«	251	11111011
¥	188	10111100	É	220	11011100	■	252	11111100
§	189	10111101	ï	221	11011101	»	253	11111101
ƒ	190	10111110	ß	222	11011110	±	254	11111110
¢	191	10111111	Ô	223	11011111		255	11111111

Table D.3 Katakana Character Set

ASCII Char.	EQUIVALENT FORMS		ASCII Char.	EQUIVALENT FORMS		ASCII Char.	EQUIVALENT FORMS		ASCII Char.	EQUIVALENT FORMS	
	Dec	Binary		Dec	Binary		Dec	Binary		Dec	Binary
NU	0	00000000		32	00100000	@	64	01000000	`	96	01100000
SH	1	00000001	!	33	00100001	A	65	01000001	a	97	01100001
SX	2	00000010	"	34	00100010	B	66	01000010	b	98	01100010
EX	3	00000011	#	35	00100011	C	67	01000011	c	99	01100011
ET	4	00000100	$	36	00100100	D	68	01000100	d	100	01100100
EQ	5	00000101	%	37	00100101	E	69	01000101	e	101	01100101
AK	6	00000110	&	38	00100110	F	70	01000110	f	102	01100110
🔔	7	00000111	'	39	00100111	G	71	01000111	g	103	01100111
BS	8	00001000	(	40	00101000	H	72	01001000	h	104	01101000
HT	9	00001001	)	41	00101001	I	73	01001001	i	105	01101001
LF	10	00001010	*	42	00101010	J	74	01001010	j	106	01101010
VT	11	00001011	+	43	00101011	K	75	01001011	k	107	01101011
FF	12	00001100	,	44	00101100	L	76	01001100	l	108	01101100
CR	13	00001101	-	45	00101101	M	77	01001101	m	109	01101101
SO	14	00001110	.	46	00101110	N	78	01001110	n	110	01101110
SI	15	00001111	/	47	00101111	O	79	01001111	o	111	01101111
DL	16	00010000	0	48	00110000	P	80	01010000	p	112	01110000
D1	17	00010001	1	49	00110001	Q	81	01010001	q	113	01110001
D2	18	00010010	2	50	00110010	R	82	01010010	r	114	01110010
D3	19	00010011	3	51	00110011	S	83	01010011	s	115	01110011
D4	20	00010100	4	52	00110100	T	84	01010100	t	116	01110100
NK	21	00010101	5	53	00110101	U	85	01010101	u	117	01110101
SY	22	00010110	6	54	00110110	V	86	01010110	v	118	01110110
EB	23	00010111	7	55	00110111	W	87	01010111	w	119	01110111
CN	24	00011000	8	56	00111000	X	88	01011000	x	120	01111000
EM	25	00011001	9	57	00111001	Y	89	01011001	y	121	01111001
SB	26	00011010	:	58	00111010	Z	90	01011010	z	122	01111010
EC	27	00011011	;	59	00111011	[	91	01011011	{	123	01111011
FS	28	00011100	<	60	00111100	¥	92	01011100	|	124	01111100
GS	29	00011101	=	61	00111101	]	93	01011101	}	125	01111101
RS	30	00011110	>	62	00111110	^	94	01011110	~	126	01111110
US	31	00011111	?	63	00111111	_	95	01011111		127	01111111

Table D.3 Katakana Character Set (cont.)

ASCII Char.	EQUIVALENT FORMS Dec	Binary
NOTE	128	10000000
NOTE	129	10000001
NOTE	130	10000010
NOTE	131	10000011
NOTE	132	10000100
NOTE	133	10000101
NOTE	134	10000110
NOTE	135	10000111
NOTE	136	10001000
NOTE	137	10001001
NOTE	138	10001010
NOTE	139	10001011
NOTE	140	10001100
NOTE	141	10001101
NOTE	142	10001110
NOTE	143	10001111
[illegible]	144	10010000
[illegible]	145	10010001
[illegible]	146	10010010
[illegible]	147	10010011
[illegible]	148	10010100
[illegible]	149	10010101
[illegible]	150	10010110
[illegible]	151	10010111
[illegible]	152	10011000
[illegible]	153	10011001
[illegible]	154	10011010
[illegible]	155	10011011
[illegible]	156	10011100
[illegible]	157	10011101
[illegible]	158	10011110
[illegible]	159	10011111
	160	10100000
｡	161	10100001
｢	162	10100010
｣	163	10100011
､	164	10100100
･	165	10100101
ｦ	166	10100110
ｧ	167	10100111
ｨ	168	10101000
ｩ	169	10101001
ｪ	170	10101010
ｫ	171	10101011
ｬ	172	10101100
ｭ	173	10101101
ｮ	174	10101110
ｯ	175	10101111
ｰ	176	10110000
ｱ	177	10110001
ｲ	178	10110010
ｳ	179	10110011
ｴ	180	10110100
ｵ	181	10110101
ｶ	182	10110110
ｷ	183	10110111
ｸ	184	10111000
ｹ	185	10111001
ｺ	186	10111010
ｻ	187	10111011
ｼ	188	10111100
ｽ	189	10111101
ｾ	190	10111110
ｿ	191	10111111
ﾀ	192	11000000
ﾁ	193	11000001
ﾂ	194	11000010
ﾃ	195	11000011
ﾄ	196	11000100
ﾅ	197	11000101
ﾆ	198	11000110
ﾇ	199	11000111
ﾈ	200	11001000
ﾉ	201	11001001
ﾊ	202	11001010
ﾋ	203	11001011
ﾌ	204	11001100
ﾍ	205	11001101
ﾎ	206	11001110
ﾏ	207	11001111
ﾐ	208	11010000
ﾑ	209	11010001
ﾒ	210	11010010
ﾓ	211	11010011
ﾔ	212	11010100
ﾕ	213	11010101
ﾖ	214	11010110
ﾗ	215	11010111
ﾘ	216	11011000
ﾙ	217	11011001
ﾚ	218	11011010
ﾛ	219	11011011
ﾜ	220	11011100
ﾝ	221	11011101
ﾞ	222	11011110
ﾟ	223	11011111
[illegible]	224	11100000
[illegible]	225	11100001
[illegible]	226	11100010
[illegible]	227	11100011
[illegible]	228	11100100
[illegible]	229	11100101
[illegible]	230	11100110
[illegible]	231	11100111
[illegible]	232	11101000
[illegible]	233	11101001
[illegible]	234	11101010
[illegible]	235	11101011
[illegible]	236	11101100
[illegible]	237	11101101
[illegible]	238	11101110
[illegible]	239	11101111
[illegible]	240	11110000
[illegible]	241	11110001
[illegible]	242	11110010
[illegible]	243	11110011
[illegible]	244	11110100
[illegible]	245	11110101
[illegible]	246	11110110
[illegible]	247	11110111
[illegible]	248	11111000
[illegible]	249	11111001
[illegible]	250	11111010
[illegible]	251	11111011
[illegible]	252	11111100
[illegible]	253	11111101
[illegible]	254	11111110
[illegible]	255	11111111

Glossary

16-bit character set	formed from pairs of ROMAN8 printable 8-bit characters. This allows representation of up to 35 344 characters, as would be needed to support Chinese, Japanese, and Korean languages.
8-bit character set	an extended ASCII (American Standard Code for Information Interchange) set. The characters include letters, numbers, punctuation, control characters, and foreign character sets.
applications program	a program that typically has a better user interface than the operating system and performs a specific application.
applications programmer	a person who writes programs for an end-user.
ASCII	American Standard Code for Information Interchange. A 128-character set represented by 7-bit binary values. (ASCII does not define the value of the eighth bit.)
bit	a contraction of BInary digiT. A bit can have a value of 0 or 1.
byte	a unit of data storage consisting of 8 bits. A byte can represent one ASCII, KANA8, or ROMAN8 character.
character	a language unit, usually consisting of 7 (ASCII) or 8 (KANA8, ROMAN8) bits.
character set	a set of characters used in a programming language or computer. They can differ in size, character type and collating sequence.
collating sequence	the ordering sequence assigned to characters or a group of characters when they are sorted and ordered by a computer.
command	a program which is executed by the shell command interpreter. Arguments following the command name are passed on to the command program. You can write your own command programs, either as compiled programs or as shell scripts (written in the shell command language).
command interpreter	a program that reads lines typed at the keyboard or from a file, and interprets them as requests to execute other programs. The command interpreter for HP-UX is called the shell.
comment	an expression used to document a program or routine that has no effect on the execution of the program.
compiler	a program that translates a high-level language into machine-dependent form.
control character	a member of a character set that produces action in a device other than a printed or displayed character. In ASCII, control characters are those in the code range 0 thru 31, and 127. Control characters are generated by simultaneously pressing a displayable character key and CNTL.

default search path — the sequence of directory prefixes that *sh, time,* and other HP-UX commands apply when searching for a file known by an incomplete path name. It is defined by PATH in *environ*. *login* sets PATH = *:bin:/usr/bin*, which means that your working directory is the first directory searched, followed by */bin*, followed by */usr/bin*.

device — a piece of peripheral equipment, usually used to input or output data.

directory — a file used to catalog other files on a mass storage medium. Each directory contains entries for its own unique files. The directory information includes name, type, length, location, and protection.

downshifting — a peripheral's provision for producing lowercase letters by using the `shift` key (on most keyboards).

editor — a program that allows you to create and modify text files based on text and commands entered from a terminal.

end-user — a person who uses existing programs and applications.

environment — the set of conditions (such as your working directory, home directory, and type of terminal you are using) that exist when you log in.

file name — a sequence of 14 or fewer characters which uniquely identifies a file in a directory. Any character except "/" can be used.

hp-8 — Hewlett-Packard's implementation of the ISO's (International Standard Organization) 8-bit character code set.

hp-16 — Hewlett-Packard's implementation of the ISO's (International Standard Organization) 16-bit character code set.

ideogram — the use of graphic symbols to represent ideas.

ideographic — representing an idea by use of a character or symbol rather than a word; the use of ideograms.

ISO7 — International Standards Organization 7-bit character substitution. The character graphics associated with some less-used ASCII codes are changed to other characters needed for a particular language.

KANA8 — the Hewlett-Packard supported 8-bit character set for support of phonetic Japanese (Katakana).

Kanji — the Japanese ideographic character set based on Chinese characters. The set consists of roughly 50,000 characters.

Katakana — The Japanese phonetic character set typically used in traditional data processing, telegrams, or to express foreign things and names. The set consists of 64 characters including punctuation.

LANG — the Unix environment variable (LANGuage) that should be set to the American English name of the native language desired.

library — a set of subroutines contained in a file that can be accessed by a user program.

library routine — one of a collection of programs within the HP-UX operating system. Each routine performs a unique task.

local customs — refers to a region's local conventions such as date, time, and currency formats.

localization — the adaptation of software for use in different countries or local environments.

message catalog — the external file containing prompts, responses to prompts, error messages, and mnemonic command names in the user's native language.

message catalog system — a set of tools developed by Hewlett-Packard to extract print statements from C programs and place them in the message catalog.

native language — a person's or user's first language (learned as a child) such as Japanese, Finnish, or American English.

natural language — the spoken or written language as opposed to a computer implementation of a language.

NLS — Native Language Support. The Hewlett-Packard model that provides capabilities for reducing or eliminating the barriers that would make HP-UX difficult to use in a native language.

operating system — a program which manages the computer's resources. It provides the programmer with utilities, including I/O routines, peripheral-handling routines, and high-level languages.

parameter — in a program, a quantity that may be given different values. It is usually used to pass conditions or selected information to a subroutine that is used by different main routines or by different parts of one main routine. Its value frequently remains unchanged throughout any one such use.

path name — a sequence of directory names separated by slashes (/), and ending in a file name (any type).

peripheral — a device connected to the computer's processor that is used to accept information from or provide information to an external environment.

pre-localization — modification to application programs before compilation to make use of language-dependent library routines and to ensure that 8-bit data can be handled properly.

program — a sequence of instructions to the computer, either in the form of a compiled high-level language or a sequence of shell command language instructions in a text file.

prompt a character displayed by the system on a terminal indicating that the previous command has been completed and the system is ready for another command. It is usually a "$" or "%", but can be redefined to any character string.

pseudo-teletype a pair of interconnected character devices; a master device and a slave device. Anything written on the master is given to the slave as input and anything written on the slave is presented as input to the master.

pty abbreviation for pseudo-teletype.

radix character the actual or implied character that separates the integer portion of a number from the fractional portion.

ROMAN8 the Hewlett-Packard supported 8-bit character set for Europe. It includes all of ASCII plus those characters necessary to support the major western European languages.

root directory the highest level directory of the hierarchical file system, in which other directories are contained. In HP-UX, the "/" refers to the root directory.

shell the shell is both a command language and a programming language that provides the user-interface to the HP-UX operating system.

shell script a sequence of shell commands and shell programming language constructs, usually stored in a text file, for invocation as a user command (program) by the shell.

space a blank character. In ASCII a space is represented by character code 32 (decimal).

standard input the source of input data for a program. The default standard input is the terminal keyboard, but the shell may redirect the standard input to be from a file or pipe.

standard output the destination of output data from a program. The default standard output is the terminal CRT, but the shell may redirect the standard output to be a file or pipe.

string a connected sequence of characters, words, or other elements.

supported language the computer-implemented version of a written or spoken language.

syntax the rules governing sentence structure in a spoken language, or statement structure in a computer language such as that of a compiler program.

teletype a trademark for a form of teletypewriter.

teletypewriter a peripheral for telegraphic data communication with a computer.

upshifting — a peripheral's provision for producing uppercase letters by using the `shift` key (on most keyboards).

USASCII — A less common name for ASCII. See ASCII.

variable — a storage location for data.

working directory — the directory in which you currently reside. Also, the default directory in which path name searches begin, when a given path name does not begin with "/".

Index

a

Accessing NLS Features 17
Active Set 12
Alternate Set 12
ASCII, 7-bit 6,7,9,11,39,53
Aspects of NLS:
  Character Set Support 6-7
  Introduction of 6-8
  Local Customs 7
  Messages 7
atof 23

b

Backslash 34
Backspace 34
Base Set 12
Bit Pattern 34

c

C:
  Accessing Message Catalogs from 35
  Example Program 1 24
  Example Program 2 25
  Key Words 6
  Library /usr/lib/libc.a 41
  Library Routines 15,17
  Message Catalog Commands 27,28
Carriage Return 34
Case 6
Character Code:
  16-bit 6
  8-bit 6
Character Set Support Model:
  Active Set 12
  Alternate Set 12
  Base Set 12
  ID Field 12
Character Set Support:
  16-bit 6
  7-bit 6
  8-bit 6
  By Peripherals 43

Collating Sequences ... 7
Directionality of Text ... 7
Introduction of ... 6-7
Language Num and Name ... 13
Character Set:
7-bit ... 9
8-bit ... 9,12
8-bit Compatible ... 9
8-bit Structure ... 9
8-bit Support Model ... 12
European ... 43
Extended ... 9
ID Numbers ... 12
Japanese ... 6,7,43
KANA8 ... 6,10,11,43,54
Names ... 12
ROMAN8 ... 6,7,10,43,56
Support By Peripherals ... 43
Supported ... 13
Characters:
Control ... 9
Escape Sequences ... 12,34
Extended Control Characters ... 9
Extended Printable ... 9
Printable ... 9
Shift In ... 12
Shift Out ... 12
Collating Sequences ... 7,15,53
Commands:
dumpmsg ... 17,28
findmsg ... 17,28
findstr ... 17,28,30,31,36
Fully Localized ... 8,39
gencat ... 17,28,30,33,37
HP-UX NLS ... 17,39-40
Incorporating NLS into ... 29
insertmsg ... 17,28,30,31,36
Message Catalog ... 28
NLS Compatible HP-UX ... 39-40
Pre-localization ... 39
See Routine ... 68
Compatible,8-bit ... 9,39
Configuration File, /usr/lib/nls/config ... 15
Control Codes ... 9
conv(3C) ... 20
ctime(3C) ... 19,24
ctype(3C) ... 21
currlangid ... 21,22,24

d

date(1) 19
Directionality of Text 7
Directory:
  /bin 15
  /usr/bin 15
  /usr/include 19
  /usr/lib 15
  /usr/lib/nls 20
Downshifting 13,54
dumpmsg 17,28

e

Escape Sequence:
  Backslash 34
  Backspace 34
  Bit Pattern 34
  Carriage Return 34
  Character 12
  Form Feed 34
  Horizontal Tab 34
  Newline 34
European Character Set 43
evct(3C) 20
Extended:
  Character Sets 9
  Control Characters 9
  Printable Characters 9

f

File:
  Format of Source Message File 34
  .login 16
  .profile 16
  /etc/profile 16
  /usr/lib/nls/config 15
  langinfo.h 19
  Language Configuration 15
  Localized Message 27
  msgbuf.h 19
  nl_ctype.h 19
  System Organization of 35
findmsg 17,28
findstr 17,28,30,31,36
Form Feed 34

FORTRAN . . . 17,27,35
fprintf . . . 25
fprintmsg . . . 22,25,28,35
Fully Localized Commands . . . 8,39

g

gencat . . . 17,28,30,33,37
getmsg(3C) . . . 21,28
GMT (Greenwhich Mean Time) . . . 7,16

h

Header Files . . . 19
Horizontal Tab . . . 34
HP-UX:
  NLS Commands . . . 17,39-40
  Non-NLS Library Calls . . . 41
  Portability Guide . . . 3,17,35
  Reference Manual . . . 3,19,35,39
  System Administrator Manual . . . 3,16

i

ID:
  Field . . . 12
  Numbers . . . 12
Ideograms . . . 7,54
Ideographic . . . 6,7,54
idtolang . . . 21,22
insertmsg . . . 17,28,30,31,36
ISO7 (7-bit Substitution) . . . 43
ISV's (Independent Software Vendors) . . . 1

j

Japanese Character Set . . . 43
JASCII . . . 43

k

KANA8 6,10,11,43,54
Kanji 6,54
Katakana 6,43,54

l

LANG (LANGuage) 15,16,24,35,37,38,54
langid(7) 20,21,22
langinfo(3C) 19,21
langinfo.h 19
langtoid 21
Language:
  Configuration File 15
  Configuring 16
  Definition of 16
  Family 13
  Name 13
  Native 5,55
  Natural 13,55
  Number 13
  Supported 13,56
Library:
  C Routines 15,17
  NLS Routines 19-23,41
  Non-NLS 41
  Support for NLS 17
Local Customs:
  Currency Units 7
  Day and Week Names 7
  Definition of 5,55
  Introduction to 7
  Representation of Numbers 7
  Time 7
Localization:
  6 Steps 36-37
  Definition of 5,55
  Steps for Simplifying the Process 28
Localized Message File:
  Definition of 8
  Example of 36-37
  Introduction to 27-28
Localized:
  Fully 8,39
  Message File 8,37
  Output 36-37

m

m (Message File Line) 34
Manual Page:
  conv(3C) 20
  ctime(3C) 19,24
  ctype(3C) 21
  date(1) 19
  evct(3C) 20
  getmsg(3C) 21,28
  langid(7) 20,21,22
  langinfo(3C) 19,21
  nl_conv(3C) 20,24
  nl_ctype(3C) 20,24
  nl_string(3C) 23,24
  printf(3S) 7,22,25,35
  printmsg(3C) 22,25,28
  strod(3C) 23
Message Catalog Command:
  findstr 17,28,30,31,36
  gencat 17,28,30,33,37
  insertmsg 17,28,30,31,36
Message Catalog System:
  Definition of 29,55
  Introduction to 27-36
Message Catalog:
  Accessing 35
  Commands 28
  Creating 29
  Definition of 8,55
  Flow of 30
  Naming Conventions 35
Message File Line:
  $ comment 34
  $delset n comment 33,34
  $set 34
  m message text 34
Message File:
  /usr/lib/nls/$LANG/hello.cat 37
  /usr/lib/nls/american/hello.cat 37
  /usr/lib/nls/german/hello.cat 38
  Format of 34
Messages 8
msgbuf.h 19

n

Native Language:
- Configuring 66
- Definition of 5,55
- Supported 13

native-computer 13,15,16,22

Natural Language 13,55

Newline 34

NLS:
- Accessing Features of 17
- Aspects of 6-7
- Compatible HP-UX Commands 39-40
- Configuring the Native Language 16
- Definition of 5,55
- File Hierarchy 15
- Header Files 19
- HP-UX Commmands 17
- Introduction to 5-13
- Library Routines 19-23,41
- Library Support for 17,41
- Non-NLS HP-UX Library Calls 41
- Programming With 19-25
- Scope of 6-8
- What It Is? 5
- Who Will Use It? 1

nl_asctime 19

nl_atof 23,24

nl_catopen 35,37

nl_conv(3C) 20,24

nl_ctime 19,24

nl_ctype(3C) 20,24

nl_ctype.h 19

nl_gvt 20

nl_isalnum 20,21

nl_isalpha 20,21

nl_isgraph 20,21

nl_islower 20,21

nl_isprint 20,21

nl_ispunct 20,21

nl_isupper 20,21

nl_string(3C) 23,24

nl_strtod 23,24

nl_tolower 20

nl_toupper 20

o

OEM's (Original Equipment Manufacturers) 1
opinstall 16

p

Pascal 17,27,35
Pathname:
/usr/lib/nls/$LANG 15
/usr/lib/nls/$LANG/collate8 15
/usr/lib/nls/$LANG/ctype 15
/usr/lib/nls/$LANG/info.cat 15
/usr/lib/nls/$LANG/shift 15
/usr/lib/nls/german/prog.cat 35
/usr/lib/nls/language/filename.cat 35
/usr/lib/nls/n-computer/prog.cat 35
/usr/lib/nls/spanish/prog.cat 35
Peripheral Localization Summary 43
Pre-localization:
Commands 39-40
Definition of 8,55
Pre-localized:
8-bit Compatible 39
Commands 8,15,39
Printable Characters 9
printf(3S) 7,22,25,35
printmsg 22,25,28,35
Program:
C Example 1 24
C Example 2 25
Example of Localization 36-37
Localizable 27

r

Roman Extension 43
ROMAN8 6,7,10,43,56
Routine:
C 15,17
ctime 19,24
currlangid 21,22,24
fprintf 25
fprintmsg 22,25,28,35
getmsg 21,28
idtolang 21,22
langid 20,21
langinfo 15,19,21
langtoid 21

Message Catalog Specific 28
nl_asctime 19
nl_atof 23,24
nl_catopen 35,37
nl_ctime 19,24
nl_ctype 20,24
nl_gvt 20
nl_isalnum 20,21
nl_isalpha 20,21
nl_isgraph 20,21
nl_islower 20,21
nl_isprint 20,21
nl_ispunct 20,21
nl_isupper 20,21
nl_strtod 23,24
nl_tolower 20
nl_toupper 20
opinstall 16
printf 22,25
printmsg 22,25,28,35
sprintf 25
sprintmsg 22,25,28,35
strcmp16 23
strcmp8 23
strncmp16 23
strncmp8 23
tolower 20
toupper 20

S

Scope of NLS 6-8
Shift In 12
Shift Out 12
sprintf 25
sprintmsg 22,25,28,35
strcmp16 23
strcmp8 23
strncmp16 23
strncmp8 23
strod(3C) 23

t

tolower 20
toupper 20
TZ (Time Zone) 16

u

Upshifting 5,13,57

Table of Contents

Using Curses and Terminfo

Introduction . . . 1
Display Data Handling . . . 2
 Output Data Structure . . . 2
 Applications Program Structure . . . 3
 Applications Program Operation . . . 5
Keyboard Input . . . 6
 Keypad Character Handling . . . 7
 Keyboard Input Program Example . . . 9
Display Highlighting . . . 10
Multiple Windows . . . 13
 Pads . . . 13
 Creating Windows . . . 14
 Using Multiple Windows . . . 14
 Subwindows . . . 16
Multiple Terminals . . . 17
Low-Level Terminfo Usage . . . 19
A Larger Example . . . 21
 Use of Escape in Program Control . . . 22
Program Routines . . . 23
 Program Structure Considerations . . . 24
 Terminal Initialization Routines . . . 25
 Option Setting Routines . . . 25
 Terminal Configuration Routines . . . 27
 Window Manipulation Routines . . . 28
 Terminal Data Output Routines . . . 29
 Window Writing Routines . . . 29
 Window Data Input Routines . . . 31
 Terminal Data Input Routines . . . 31
 Video Highlighting Attribute Routines . . . 32
 Miscellaneous Functions . . . 33
curses Routines . . . 34
Description of Routines . . . 36
Terminfo Routines . . . 55
Termcap Compatibility Routines . . . 57
Program Operation . . . 58
 Insert/Delete Line . . . 58
 Additional Terminals . . . 58
 Multiple Terminals . . . 59

Video Highlighting 60
Special Keys 62
Scrolling Regions 63
Mini-Curses 63
TTY Mode Functions 64
Example Programs 66
SCATTER 66
SHOW 67
HIGHLIGHT 68
WINDOW 70
TWO 71
TERMHL 73
EDITOR 75
Subject Index

Using Curses and Terminfo

Introduction

This tutorial describes the operation of *curses(3x)* and *terminfo(5)*. It is intended for use by programmers who are interested in writing screen-oriented software using the *curses* package. *curses* uses *terminfo* when interacting with a given terminal in the system and when formatting display data for subsequent output to the terminal display.

curses is a versatile cursor and screen control package that has many capabilities. It is designed to efficiently utilize terminal screen control and display capabilities, thus limiting its demand for computer CPU resources. It can create and move windows and subwindows, use display highlighting features, and support other terminal capabilities that enhance visual interaction with display terminal users. All interaction with a given terminal is tailored to the terminal type which is obtained from the environment variable TERM).

curses also interacts with the terminal keyboard, and can handle user inputs. Its ability to handle keys that produce multi-character sequences (such as arrow keys) as ordinary keys can be used to add versatility to application programs.

Display Data Handling

Output Data Structure

curses uses data structures called windows to collect display text, then transfers the data structures to the terminal display screen during execution of *refresh* routines. Each window contains a two-dimensional data array for storing text and character highlighting attributes. Other data structures associated with the window contain the current cursor position and various pointers, and fill other *curses* needs.

Two windows are always present when *curses* is active. **Current screen** is named *curscr* for programming purposes, and represents the current screen. It is used as a reference when optimizing output operations to the CRT screen. The **standard screen** window, named *stdscr*, is the default destination for all text output operations that are not directed to a window specified in the function. Both *curscr* and *stdscr* have the same row and column dimensions as the physical display screen.

Additional program-definable windows can be created and dimensioned as programming needs dictate. Such windows can be any size, provided they do not exceed the row and/or column capacity of the physical display screen.

When a program requires a window that is larger than the available display screen, pads are used. Pads have the same structure and characteristics as a window, but they can be any size within the limits of reasonable memory usage (each pad requires two bytes per character position plus data structure overhead).

Text and Highlighting Data Format

Every window data structure contains, among other things, a two-dimensional array of 16-bit data words, each word corresponding to a displayable character in the window. Seven bits in each 16-bit word contain the 7-bit character code of the character associated with the corresponding screen display position. The remaining nine bits specify which highlighting attributes, if any, are to be used when the character is displayed. The window data structure also contains a set of current attributes that are used to form the attribute bits as each word is placed in the array by *addch* or its equivalent. If text highlighting is to be changed for a given character or set of characters, an update to the current attribute set must be performed by *attrset* (or its equivalent) before *addch* is performed. The beginning default attribute set disables all highlighting.

Applications Program Structure

Consider the following example of an application program structure that uses *curses*:

```
#include <curses.h>
 . . .
     initscr(); /* Initialization */

     cbreak();/* Various optional mode settings */
     nonl();
     noecho();
 . . .
     while (!done) {/* Main body of program */
         . . .
         /* Sample calls to draw on screen */
         move(row,col);
         addch(ch);
         printw("Formatted print with value %d\n", value);
         . . .
         /* Flush output */
         refresh();
         . . .
     }

     endwin();/* Clean up */
     exit(0);
```

One of *curses*' major advantages is its ability to optimize the process of updating terminal screen contents, thus reducing the demand for CPU and I/O resources by reducing the amount of data handling required for requested changes in displayed text. This is accomplished by comparing the current screen contents with the window being transferred, then transmitting only those text and control characters that are needed to most efficiently update the screen. Other screen contents remain undisturbed.

NOTE

Most terminals are equipped with hardware scrolling whose operating characteristics make it impossible to write characters in the extreme lower right-hand character position.

In order to optimize screen updates, *curses* must have access to a data base that reflects current screen contents. When an application program starts execution, the current screen is unknown. To provide a starting current screen reference, a screen clearing operation must be set up early in the program by a call to *initscr()* which identifies the terminal, initializes data structures, and enables the *clearok* option in *curses* so that the screen is cleared during the first *refresh* operation in the program. Upon completion of the first refresh operation, the terminal screen is an exact replica of the text stored in the current screen data base. Use of *initscr()* in a typical program is shown in the preceding sample program structure example.

When initialization is complete, other operating modes and options can be selected as dictated by program needs. Available operating modes include *cbreak()* and *idlok(stdscr, TRUE)* which are explained in detail later. During program execution, screen output is handled through routines such as *addch(ch)* and *printw(fmt,args)*. They are equivalent to *putchar* and *printf*, respectively, but use *curses* in addition to the usual other system facilities. Cursor and character positioning are performed by *move* and other similar calls.

All of the routines mentioned send their output to program-specified window data structures; not directly to the display screen. The window data structure represents all or part of a CRT display screen, and contains the following items:

- An array of characters to be displayed on the screen area defined by the window boundaries,
- Present cursor location,
- Current set of video attributes, and
- Various operating modes and options.

There is little need to be concerned with windows (unless you use several windows during program operation), except to recognize that the data structure corresponding to a given window acts as a buffer/data accumulator for display output requests.

Accumulated contents of a window data structure are sent to the display screen by use of *refresh()* or an equivalent function for windows and pads (functionally similar to a *flush*). *curses* considers many different ways of handling the output operation, taking into account the various available terminal characteristics, similarities between the current screen display and the desired pattern, and other factors. Refresh operations are usually handled using as few characters as possible, but not always.

When the application program is finished, certain clean-up operations should be performed before termination. While the amount of clean-up needed varies, depending on program structure and capabilities, termination should always include a call to *endwin()*. *endwin()* restores all terminal settings to their original state prior to program execution, places the cursor at the bottom left corner of the screen, and dismantles data structures that are no longer needed.

Among the example programs at the end of this tutorial is a program named *scatter* that reads a file and displays the file contents in random order on the CRT display screen. While some application programs assume that terminals have twenty-four 80-character lines of available display space, many terminals do not. To accommodate display terminals having various screen sizes, the variables `LINES` and `COLS` are defined by *initscr* to specify the current screen size. Application programs should always use screen-size variables rather than assuming a 24×80 display screen.

Applications Program Operation

During program operation, no data is output to the display terminal until *refresh* is called. Instead, program routines such as *move* and *addch* place data in a window data structure called *stdscr* (standard screen) that is maintained by *curses*. *curses* also maintains a replica of what is on the current physical screen in *curscr* for updating purposes.

When *refresh* or an equivalent function is called, *curses* compares the *curscr* window with what is presently contained in *stdscr* (or other specified window or pad). The results of the comparison are combined with terminal hardware capabilities to construct character streams that most efficiently update the physical display to the desired contents. Available terminal capabilities are considered while comparing *stdscr* and *curscr* so that the most efficient means of updating the screen can be determined. This sequence is referred to as cursor optimization, and is the basis for naming the *curses* package. During the update operation, *curscr* is also changed to reflect the contents of the updated screen.

Keyboard Input

curses capabilities include more than screen writing functions. Several keyboard input functions are also supported, including special handling of certain keys that normally generate a sequence of two or more characters (usually an escape code followed by a single character, but not always). Such keys can then be treated as ordinary single-character keys for improved programming versatility.

The most commonly used keyboard input function is *getch( )* which waits for the terminal user to type a character on the terminal keyboard, then returns the character to the calling program. *getch* is similar to *getchar*, except that it uses *curses* instead of other HP-UX facilities. *getch* is particularly useful in programs that use *cbreak( )* or *noecho( )* options because *getch* supports several terminal- and system-dependent options that are not accessible through *getchar*. Available *getch* options include:

- *keypad* enables programmers to use non-typing keys such as arrow keys, function keys, and other special keys that transmit escape sequences or other multi-character sequences as ordinary single-character keys. Keypad character code length requires 16-bit integer variables for storage.
- *nodelay* enabled option causes *getch* to return immediately with the value −1 if no input character is waiting. This avoids program delays that would otherwise result when no response from the terminal is available.
- *getstr* can be used to input an entire string of characters up to a newline instead of a single character. It also handles echo, erase, and kill character functions associated with the input operation.

Example programs at the end of this tutorial show how these options are used.

Keypad Character Handling

When *keypad* is enabled, keypad character sequence conversion tables in the *terminfo* data base are used to map keypad character sequences into corresponding single, 16-bit character form. Each supported keypad key must produce a unique character or character sequence when pressed. All convertible sequences must be included in the *terminfo* data base. If any sequence is absent from the table, it cannot be converted, so it is handled in unaltered form. The following special keys are assigned the values and names indicated. Some of the keys listed may not be supported on given terminals, depending on the terminal model and its internal operating characteristics, and whether the conversion sequence is in *terminfo*.

NOTE

Keypad character codes do not fit in a normal 8-bit data element. Therefore a *char* variable cannot be used. Use a larger (16-bit) variable for storing and handling keypad character codes.

Keypad Character Code Values

Character Name	Octal Value	Key name
KEY_BREAK	0401	Break key (unreliable)
KEY_DOWN	0402	Down Arrow key
KEY_UP	0403	Up Arrow key
KEY_LEFT	0404	Left Arrow key
KEY_RIGHT	0405	Right Arrow key
KEY_HOME	0406	Home Up (to upper left corner) key
KEY_BACKSPACE	0407	Backspace key (unreliable)
KEY_F0	0410	Function Key 0
. . .	. . .	. . .
KEY_F(n)	0410+(n)	Function Key (n)
KEY_DL	0510	Delete Line key
KEY_IL	0511	Insert Line key
KEY_DC	0512	Delete Character key
KEY_IC	0513	Insert Character or Enter Insert Mode key
KEY_EIC	0514	Exit Insert-character Mode Key
KEY_CLEAR	0515	Clear Screen key
KEY_EOS	0516	Clear to End-of-Screen key
KEY_EOL	0517	Clear to End-of-line key
KEY_SF	0520	Scroll Forward 1 Line
KEY_SR	0521	Scroll Reverse (backwards) 1 line
KEY_NPAGE	0522	Next Page key
KEY_PPAGE	0523	Previous Page key
KEY_STAB	0524	Set Tab key
KEY_CTAB	0525	Clear Tab key
KEY_CATAB	0526	Clear All Tabs key
KEY_ENTER	0527	Enter or Send key (unreliable)
KEY_SRESET	0530	Soft (partial) Reset key (unreliable)
KEY_RESET	0531	Reset or Hard Reset key (unreliable)
KEY_PRINT	0532	Print or Copy key
KEY_LL	0533	Home Down (to lower left) key

Keyboard Input Program Example

The example program **show** at the end of this tutorial contains an example use of *getch*. **Show** displays a file, one screen at a time; advancing to the next page each time the space bar is pressed. Nearly any exercise for *curses* can be created by constructing an input file that contains a series of 24-line pages, each page varying slightly from the previous page.

In the **show** program:

- *cbreak* is used so that only the space bar need be pressed (use of **RETURN** is unnecessary).
- *Noecho* is used to prevent the character transmitted by the space bar from being echoed during *refresh* calls so that refresh operations are not adversely affected.
- *nonl* is called to enable additional screen optimization.
- *idlok* allows insert and delete line. This capability helps streamline updates in some instances, but produces undesirable effects in other cases. Therefore an option to allow or disallow the capability has been provided.
- *clrtoeol* clears from cursor to end of current line.
- *clrtobot* clears from cursor to end of current line, then clears all subsequent lines to the bottom of the screen.

Display Highlighting

curses supports nine highlighting attributes, each of which has a corresponding 16-bit integer constant named in the include file <`curses.h`>. The value of each constant is selected such that one bit (corresponding to the attribute) in the 16-bit integer is set while all other bits are cleared. Here is a list of the nine attributes with their corresponding enable-bit positions. The name and octal value of each constant is also shown (note that only six digits are needed to represent the 16-bit value; the leading zero identifies the constant as an octal value).

- Standout (bit 7):
  `A_STANDOUT` = 0000200
- Underlining (bit 8):
  `A_UNDERLINE` = 0000400
- Inverse Video (bit 9):
  `A_REVERSE` = 0001000)
- Blinking (bit 10):
  `A_BLINK` = 0002000
- Dim (bit 11):
  `A_DIM` = 0004000
- Bold (bit 12):
  `A_BOLD` = 0010000
- Invisible (bit 13):
  `A_INVIS` = 0020000
- No print or display (bit 14):
  `A_PROTECT` = 0040000
- Alternate Character Set (bit 15):
  `A_ALTCHARSET` = 0100000

addch and *waddchr* store window characters as 16-bit data words where the lower seven bits (0-6) of each word contain the character code and the upper nine bits (7-15), when set, enable the corresponding display highlighting attributes when that character is displayed on a terminal. Each attribute bit corresponds to one of the highlighting functions listed above. Obviously, any selected highlighting feature that is not available on a given terminal cannot be used even though the capability is standard fare for *curses*. However, when a requested attribute is not available on a given terminal, *curses* attempts to identify and use a suitable substitute. If none is possible, the attribute is ignored.

Three other constants in <`curses.h`> are also useful:

- `A_NORMAL` (value = 0000000) can be used as an argument for *attrset* to disable all attributes. *attrset(A_NORMAL)* is equivalent to *attrset(0)*, but more descriptive.
- `A_ATTRIBUTES` has an octal value of 0177600. It can be used in a bit-level logical AND to remove character bits, isolating the attributes attached to a given character.
- `A_CHARTEXT` has an octal value of 0000177. It is useful in a bit-level logical AND to discard all except the lower seven bits of the data word; in effect, separating the character from its highlighting attributes.

curses maintains a set of **current attributes** for each window. Whenever text is being placed in a given window by the program, the current attribute bits for the selected window are added to each character of text data, forming a 16-bit word for each character handled. To select a specific combination of attributes, a program call to *attrset* (or *attron*) with new attribute values must precede text output to the window. This can be used to enable one or more attributes when all were previously disabled, disable all currently enabled attributes (*attrset(0)*), or change the current set to any other new current set.

To enable one or more attributes in the current set without altering other active or inactive attributes, call *attron*. A call to *attroff* performs the opposite function, disabling the selected attributes without disturbing any other attributes in the current set.

curses always uses current attribute values, so a call to *attrset*, *attron*, or *attroff* (or their related window functions) must be used whenever you begin, end, or change any selected highlighting option. Here is an example program segment that illustrates how to set a word in boldface then restore normal display attributes for remaining text:

```
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
. . .
refresh();
```

In this example, the space characters before and after the word **boldface** are included in text blocks outside (before and after) the *attrset* calls. This technique prevents *curses* from applying display highlights to the spaces, thus avoiding possible undesirable effects; especially in situations where *curses* attempts to substitute an alternative for unavailable highlighting features.

The attribute `A_STANDOUT` offers unique program flexibility. In many interactive programs, displayed text needs to be enhanced to attract attention. However, it is not critical that the text be displayed with specific attributes. Many multi-terminal systems contain various terminal models that do not support identical highlighting features. For versatility, `A_STANDOUT` uses the terminal characteristics stored in the *terminfo* data base to determine the most pleasing highlighting feature available on the terminal being addressed (usually bold or inverse video), then uses that feature when sending corresponding text to the selected window on the terminal display screen. Two functions, *standout()* and *standend()* are provided so you can conveniently enable and disable `A_STANDOUT` highlighting.

attrset can be used to select only one (such as **A_BOLD**, shown in the earlier example in this section) or multiple attributes (such as **A_REVERSE** and **A_BLINK** for blinking inverse video). To change only one attribute or a certain combination of attributes while leaving the others undisturbed, use *attron()* and *attroff()*.

The example program **highlight** at the end of this tutorial demonstrates typical use of attributes. The program uses a text file as input, and embedded escape sequences in the file to control attributes. In the example program, \U enables underlining, \B selects bold, and \N restores normal text. An initial call to *scrollok* allows the terminal to scroll if the text file exceeds the capacity of a single display screen. When *scrollok* is active, if any text extends beyond the lower screen boundary, *curses* automatically scrolls the internally stored window up one line, then calls *refresh* to update the terminal display screen each time a line of input text exceeds the lower screen boundary. The scrolling process continues until end-of-file is reached on the input file.

The **highlight** program comes about as close to being a filter as is possible with *curses*. It is not a true filter because *curses* interacts directly with the terminal screen. *curses*' ability to optimize interaction between HP-UX programs and terminals is inherently linked to its direct monitoring of the current CRT screen and the windows where display text is being held for output through *refresh* operations. This capability requires that *curses* clear the screen as part of the first *refresh* operation so that it has a known beginning reference condition, then maintain a continually up-to-date data structure that reflects current screen contents and cursor location.

Multiple Windows

A window is a data structure that represents all or part of the CRT display screen. It contains a two-dimensional array of 16-bit character data words, a cursor, a set of current attributes, and several flags. Each 16-bit character data word contains:

- A 7-bit character code in the lower seven bits, and
- A 9-bit video highlighting code in the upper nine bits. Each bit enables one of nine attributes when set, each attribute represented by one of the respective bits.

curses provides a full-screen window called `stdscr` and a set of functions that use `stdscr`. Another window called `curscr` that represents the current physical display screen is also provided.

It is important that you clearly understand that a window is only a data structure. Use of more than one window does not imply the presence of more than one terminal, nor does it involve more than one process. A window is nothing more than a data object that can be copied to all or part of the terminal screen. *curses*, as presently implemented, cannot handle windows that are larger than the available display screen (use pads for such applications).

Pads

Pads are data structures that are essentially identical to windows, except that they can be larger than the available terminal screen size, and, as a result, must be handled differently. For example, a special refresh function is required that knows how to transfer only a specified part of the total pad area to the current screen instead of the entire pad. Other window operations do not depend on the size of the structure, so they can treat windows and pads identically. In such instances, a single function supports pads and windows (such as *addch*, *delwin*, and similar functions).

Creating Windows

Additional windows can be created so that the applications program can maintain several different screen images. Images can then be alternated under program control as needs dictate. Windows can be useful in editors, games, and other applications such as when handling interactive processes involving multiple users on multiple terminals.

Overlapping windows can also be constructed so that changes to one window are easily copied onto the overlapping area of the second. Several *curses* routines have been provided specifically to handle such cases. *overlay* and *overwrite* copy one window onto the second, each handling the copy operation differently. *wrefresh* can be used to refresh the terminal screen, but in some cases it is more efficient and pleasing to perform a series of internal window operations that are equivalent to refresh, but which do not update the screen. This is done by using a series of calls to *wnoutrefresh* (or its equivalent for pads), followed by a single *doupdate* that copies the series of refreshes onto the physical screen in a single operation. This is readily provided because *refresh* is really a call to *wnoutrefresh* followed by a call to *doupdate.*

To create a new window, use the function:

newwin(`lines`, `cols`, `begin_row`, `begin_col`*)*

The newwin function call returns a pointer to the newly created window whose dimensions are `lines` by `cols`, and whose upper left-hand corner is positioned at screen location `begin_row` and `begin_col`.

Using Multiple Windows

All operations that affect *stdscr* have a corresponding function for use with other named windows. These functions' names are formed by adding the letter *w* in front of the *stdscr* function name. For example, the window function that corresponds to *addch* is named:

waddch(`mywin`, `c`*)*

To update the contents of the currently displayed screen to match the contents of a window, use:

wrefresh(`mywin`*)*

Whenever the boundaries of two or more windows overlap and thus conflict, the most recently refreshed window becomes the currently displayed screen in that area of the display area that is defined by the window size and location.

Any call to the non-w version of any window function (`stdscr` function calls) is converted to its w-prefixed counterpart. Thus, a call to *addch(*`c`*)* produces a call to *waddch(*`stdscr`, `c`*)*, automatically adding the `stdscr` argument in the process.

The example program **window** at the end of this tutorial shows how windowing can be handled. The main display is kept in *stdscr.* When the user wants to put something else on the screen, a new window is created that covers part of the screen. A call to *wrefresh* on that window causes the window to be written over *stdscr* on the display screen. A subsequent call to *refresh* on *stdscr* causes the original window to be fully restored to the screen, eliminating the temporarily displayed window.

Examine the *touchwin* calls in **window** that precede refresh calls on overlapping windows. *touchwin* calls prevent optimization by *curses*, thus forcing *wrefresh* to completely overwrite the entire window area on the physical screen (previously displayed data is thus erased in the window area only). In some situations, if the *touchwin* call is omitted, only part of the window is written and existing information from a previous window may remain in the newly written window area.

For improved screen addressability, a set of move functions are available in conjunction with most common window functions. They produce a call to *move* before the other function is called, so that the cursor can be relocated before the window function is executed. Here are some examples:

- *mvaddch(row,col,ch)* is equivalent to *move(row,col); addch(ch)*
- *mvwaddch(row,col,win,ch)* is equivalent to *wmove(win,row,col); waddch(win,ch).*

Refer to the *curses* routines section of this tutorial for more detailed descriptions of the window routines and their related move functions.

Subwindows

Subwindows can be created within any existing window or pad. Subwindows are identical to normal windows except that the subwindow's character data structure occupies the same memory locations as the corresponding character positions in the main window. This means that whenever a character is placed in a subwindow, the main window automatically contains the same character in the same location with the same highlighting attributes. In fact, as a result of shared character storage, any character stored in the character array automatically receives the current attributes for the window or subwindow through which it was stored, regardless of how many subwindows overlap the storage location. This feature greatly simplifies combining windows in a single display for some types of applications.

Each subwindow has its own cursor location, can be configured with a soft scrolling region, and generally has the same capabilities as any normal window, but, except for shared character storage, is completely independent of the original window it is associated with. Because of shared character data structures, *curses* does not allow deletion of any window (*delwin(win)* or pad that has one or more undeleted subwindows.

If subwindows are created within a pad, care must be exercised in the choice of correct refresh functions and other program characteristics to ensure correct data handling.

Multiple Terminals

curses can produce simultaneous output on multiple terminals. This capability is useful in single-process programs that access a common data base such as multi-player games. Output to multiple terminals is a complex issue, and *curses* does not solve all of the related programming problems. For example, it is the program's responsibility to determine the special file name for each terminal line and what type of terminal is connected to that line. The normal method, checking the environment variable `$TERM`, does not work because each process can only examine its own environment. Another issue that must be addressed is the case of multiple programs reading data from a single terminal line, a situation that produces race conditions which must be avoided because a program that wants to take over a terminal cannot arbitrarily stop whatever program is currently running on that terminal (particularly where security considerations make this action inappropriate, though it is appropriate for some applications such as inter-terminal communication programs).

Race conditions may or may not be a problem, depending on the overall relationships of running programs and processes. For example, if a *curses* program is looking for input from a terminal, there **must** be no other program looking for input from the same terminal (such as a shell). On the other hand, if two programs are sending output to the same terminal at the same time, the result is usually no worse than an unusable screen display. In any event, for interaction with the terminal to flow smoothly, conflicts in terminal access must be prevented.

A typical solution requires the user logged onto each terminal line to run a program that notifies the master program that the user is interested in joining the master program. The master program is given the notification program's process id, the name of the tty link, and the type of terminal being used. The notification program then goes to sleep until the master program finishes. During termination, the master program wakes up the notification program and all programs exit.

curses handles multiple terminals by always having a **current terminal**. All function calls always pertain to the current terminal. The master program should set up each terminal, saving a reference (pointer) to the terminal in its own variables. When it is ready to interact with a given terminal, the master program should set the current terminal (use *set_term*) according to program needs, then use ordinary *curses* routines.

Terminal references have type `struct screen *`. To initialize a new terminal, call *newterm(type,fd)*. *newterm* returns a screen reference to the terminal being set up. `type` is a character string that names the kind of terminal being used. `fd` is a stdio file descriptor to be used for input and output to the terminal (if only output is needed, the file can be opened for output only). The *newterm* call replaces the normal call to *initscr*.

To select a new current terminal, call *set_term(sp)* where **sp** is the screen reference returned by *newterm* for the terminal being selected. *set_term* returns a screen reference to the previous terminal.

A full set of windows and options must be maintained for each terminal according to program needs. Each terminal must be initialized separately with its own *newterm* call. Options such as *cbreak* and *noecho*, and functions such as *endwin* and *refresh* must be set (or called) separately for each terminal. Here is a typical scenario for sending a message to each terminal:

```
for (i=0; i <nterm; i++) {
   set_term(terms[i]);
   mvaddstr(0,0,"Important message");
   refresh();
}
```

The sample program **two** at the end of this tutorial contains a full example of how this technique is implemented. The program pages through a file, showing one page to the first terminal; the next page to the second. It then waits for a space character to be typed on either terminal, then sends the next page to the terminal that sent the space character. Each terminal has to be put into nodelay mode separately. No standard multiplexer is available in current HP-UX versions, so it is necessary to busy wait or call *sleep(1);* between each check for keyboard input. **two** waits one second between checks for available terminal keyboard characters.

two is only a simple example of two-terminal *curses*. It does not handle notification as described above; instead, it requires the name and type of the second terminal on the program procedure line. As written, **two** requires that the command *sleep 100000* be typed on the second terminal to put it to sleep while the program runs, and the the first-terminal user must have read and write permission on the second terminal.

Low-Level Terminfo Usage

Some programs need access to lower-level primitives than those offered by *curses.* For such programs, the ***terminfo*-level** interface is provided. This interface does not manage the CRT screen, but gives programs access to strings and capabilities that can be used to manipulate the terminal.

Use of *terminfo*-level routines is discouraged. Whenever possible, higher-level *curses* routines should be used instead, in order to maintain portability to other systems and handle a wider variety of terminal types. *curses* takes care of all of the anomolies, glitches, and personality defects present in physical terminals, but at the *terminfo* level they must be dealt with in the program. Also, there is no guarantee that the *terminfo* interface will not change with new releases of HP-UX or be upward compatible with previous releases.

There are two circumstances where use of *terminfo* routines is appropriate. On instance is where a special-purpose program 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). The second is when writing a filter. A typical filter performs one transformation on the input stream without clearing the screen or addressing the cursor. If this transformation is terminal-dependent and clearing the screen is inappropriate, *terminfo* routines are preferred.

A program written at the *terminfo* level uses the framework shown here:

```
#include <curses.h>
#include <term.h>
. . .
    Setupterm(0,1,0);
    . . .
    putp(clear_screen);
    . . .
    reset_shell_mode();
    exit(0);
```

The call to *setupterm* handles initialization (*setupterm(0,1,0)* invokes reasonable defaults). If *setupterm* cannot determine the terminal type, it prints an error message and exits. The calling program should call *reset_shell_mode* before exiting.

Global variables with such names as *clear_screen* and *cursor_address* are defined during the call to *setupterm.* When outputting these variables, use calls to *putp* or *tputs* for better programmer control during output. Global variable strings should not be output to the terminal through *printf* because they contain padding information that must be processed. A program (*such as printf*) that transmits unprocessed strings will fail on terminals that require padding or use Xon/Xoff flow-control protocol.

Higher-level routines described previously are not available at the *terminfo* level. The programmer must determine output needs and structure programs accordingly. For a list of *terminfo* capabilities and their descriptions, see *terminfo(5)* in the *HP-UX Reference.*

The example program **termhl** at the end of this tutorial shows simple use of *terminfo.* It is similar to **highlight**, but uses *terminfo* instead of *curses.* This version can be used as a filter. The strings used to enter bold and underline mode, and to disable all highlighting attributes are demonstrated.

The program was made more complex than necessary in order to illustrate several *terminfo* properties. For example, *vidattr* could have been used instead of directly outputting *enter_bold_mode*, *enter_underline_mode*, and *exit_attribute_mode.* In fact, the program could easily be made more robust by using *vidattr* because there are several ways to change video attributes. However, this program was structured only to illustrate typical use of *terminfo* routines.

The function *tputs(cap,affcnt,outc)* adds padding information to the capability `cap`. Some capabilities contain strings such as $<20>, which means to pad for 20 milliseconds. *tputs* adds enough pad characters to produce the desired delay. `cap` is the string capability to be output; `affcnt` is the number of lines affected by the output (for example, *insert_line* may have to copy all lines below the current line, and may require time proportional to the number of lines being copied). By convention, `affcnt` is 1 if no lines are affected rather than 0 because `affcnt` is multiplied by the amount of time required per item, and a zero time may be undesirable. `outc` is the name of a routine that is to be called with each character being sent.

In many simple programs, `affcnt` is set to 1, and `outc` just calls *putchar.* For such programs, the *terminfo* routine *putp(cap)* is a convenient abbreviation. The example program **termhl** could be simplified by using *putp.*

Note the special check for the *underline_char* capability. Some terminals, rather than having a code to start underlining and a code to stop underlining, use a code to underline the current character. **termhl** keeps track of the current mode, and outputs *underline_char*, if necessary, whenever the current character is to be underlined. Low-level details such as this are a major reason why *curses* routines are preferred over *terminfo* routines. *curses* takes care of all the different terminal keyboard and display functions and highlighting sequences instead of forcing such details onto the application program.

A Larger Example

The example program **editor** is a very simple screen editor that has been patterned after the *vi* editor and illustrates how *curses* can be used for such applications. **editor** uses *stdscr* as a buffer for simplicity, whereas a more useful editor would maintain a separate data structure for editing operations, then display the pertinent contents of that separate structure on the screen. **Editor**, as written, requires a file size equal to screen size. It also cannot handle lines longer than the screen, and has no provision for control characters in the file.

Several program characteristics are of interest. The routine that writes the file back to the file system shows how *mvinch* is used to retrieve characters from given window positions. The data structure used does not provide for keeping track of the number of characters in a line nor the number of lines in the file, so trailing blanks are eliminated when the file is written out.

editor uses built-in *curses* functions *insch*, *delch*, *insertln*, and *deleteln*. These functions behave much like equivalent functions on intelligent terminals when inserting and deleting characters and lines.

The command interpreter accepts not only ASCII characters, but also special (non-typing) keys. This is important — a good program accepts both. Defining the keyboard so that every special key has its function defined on a normal typing key as well provides a desirable increase in flexibility. The benefit for new users, for example, is that they can use arrow keys without having to remember that the same functions are available on h, j, k, and l keys in the normal typing area. On the other hand, an experienced user may prefer to keep his fingers on the home typing row where he can work faster, so the typing key equivalent of special keys is appreciated. Handling both classes of keys also widens the variety of terminals the program can interact with because some terminals may not be equipped with arrow or other special keys on the keyboard. Providing an ASCII character synonym for each special keypad key provides better overall program and system flexibility, and makes the program more salable and easier to learn.

Note the call to *mvaddstr* in the input routine. *addstr* is roughly equivalent to the *fputs* function in C. Like *fputs*, *addstr* does not add a trailing newline. It is equivalent to a series of calls to *addch*, using the characters in the string. *mvaddstr* moves the current cursor position to the specified location in the window before writing the string into the data structure.

The control-L command demonstrates a feature that most programs using *curses* should include. Frequently, an independent program operating beyond the control of *curses* may write something to the terminal screen, or some other event such as line noise causes the physical screen to be altered without *curses* being notified. In such a case, CTRL-L can be used to clear and redraw the current screen at the user's request. This is accomplished by a call to *clearok(curscr)* which sets a flag that causes the next *refresh* to clear the screen. A call to *refresh* follows immediately so that the screen is immediately redrawn using the data in *curscr* so that there is no wait for other program activities or completion of a pending keyboard input. There is also no loss of current screen data.

Note also the call to *flash()* which flashes the screen (unless the terminal has no flashing capability, in which case it rings the bell instead). Replacing the bell with the flashing capability is useful in environments where the sound of the bell is objectionable or distracting. Still, there may be instances where an audible signal is still needed for certain purposes, even in quiet environments. In such cases, the *beep()* routine can still be called instead whenever a real beep is preferred. If *beep* is called and the terminal is not equipped to process the call, *curses* substitutes the *flash* in its place if possible, and vice versa. Thus, a terminal with no beep capability receives a flash sequence when beep is called; a terminal that cannot flash receives a beep sequence when flash is called. If the terminal has neither capability, ... well, ... some situations do present certain limitations — do without or get a different terminal because both are ignored in such a case.

Use of Escape in Program Control

Another important programming practice is terminating the input command with control-D; not escape. It is very tempting to use escape as a command because the escape key is one of the few special keys that is available on nearly every terminal keyboard (return and break are the only others). However, using escape as a separate key introduces an ambiguity which is handled by *curses* as follows:

Most terminals use sequences of characters beginning with an escape character (called escape sequences) to control the terminal. They also use similar escape sequences to transmit special keys to the computer. If the computer sees an escape character from the terminal, it cannot immediately determine whether the user pressed the escape key, or whether a special key was pressed instead. *curses* handles the ambiguity by waiting for up to one second. If another character is received within the one-second time limit, the escape and second character are compared with possible escape sequences. If the character pair represents a valid possibility, the wait is extended for up to one more second, or until the next character is received. The cycle continues until a valid special key sequence is completed or a character is received that could not be part of a valid sequence (or the time limit expires).

While this technique works well most of the time, it is not foolproof. For example, a user could press the escape key then press one or more other keys that represent a valid sequence before the time limits expired (less than one second between successive key strokes). *curses* would then think that a special key had been pressed. Another disadvantage is the inevitable delay from the time a key is pressed until it can be processed by the program when an escape key is pressed, possibly even accidentally.

Many existing programs use escape as a fundamental command which often cannot be changed without incurring the wrath of a large group of users. Such programs cannot make use of special keys without dealing with the aforementioned ambiguity, and must, at best, resort to a timeout solution. The pathway is clear. When designing new programs and updating older ones, avoid using the escape key for program control whenever possible.

Program Routines

This and the following sections describe *curses* routines that are available to programmers. In this section, the routines are discussed in groups by function in the context of program operation. The next sections list *curses*, *terminfo*, and *termcap* compatibility routines alphabetically for easy reference, and each is discussed in greater detail. Both are helpful as tutorial and reference information, expanding on the information contained in the *curses(3X)* and *terminfo(5)* entries in the *HP-UX Reference*.

The *curses* routines discussed in this section operate on pads, windows, and subwindows. In general, windows and subwindows are treated identically by most routines. Subwindows share character data structures with the original window, but have their own cursor location and other non-character data structures. Unless indicated otherwise, all references to windows during discussion of window routines apply equally to windows and subwindows.

Program Structure Considerations

All programs using *curses* should include the file <*curses.h*> which defines several *curses* functions as macros and establishes needed global variables as well as the datatype `WINDOW` (window references are always of type `WINDOW *`). *curses* also defines the `WINDOW *` constants *stdscr* (the standard screen that is used as a default for all routines that interact with windows) and *curscr* (the current screen, used as a reference for low-level operations when updating the current display or clearing and redrawing a scrambled display. The integer constants `LINES` and `COLS` are defined, and contain values equal to the number of available lines and columns in the physical display. The constants `TRUE` and `FALSE` are also defined with the values 1 and 0, respectively. Two additional constants are defined; the values returned by most *curses* routines. `OK` is returned when the routine was able to successfully complete its assigned task. `ERR` indicates that an error occurred (such as an attempt to place the cursor outside a defined window boundary or create a window larger than the physical screen); thus, the task was not successfully completed.

The include file `<curses.h>` that must be specified at the beginning of the program automatically includes `<stdio.h>` and an appropriate tty driver interface file, presently `<termio.h>`. Including `<stdio.h>` again in a subsequent program statement is harmless though wasteful, but including a tty driver interface file could cause a fatal error if the file is not the same as the one selected by *curses.*

Any program that uses curses should include the loader option

```
-lcurses
```

in its makefile, whether the program operates at the *curses* or *terminfo* level. If the program only needs *curses*' screen output and optimization capabilities, and no non-default windows are involved, you can improve output speed and processing efficiency by restricting the program to the mini-curses package. Mini-curses is selected by using the compilation flag

```
-DMINICURSES
```

Routines supported by mini-curses are marked by asterisks in the complete list of *curses* routines at the beginning of the *curses* Routines section of this tutorial. They are also similarly marked in the *HP-UX Reference* under *curses(3X).*

Terminal Initialization Routines

Program entry and exit states must be handled correctly to maintain system integrity and proper terminal operation. If the program interacts with only one user/terminal, *initscr* should be the first function call in the program. It sets up the necessary data structures and makes sure that terminal handling and screen clearing are properly initialized. The program should call *endwin* before terminating, ensuring that the terminal is restored to its original operating state and the cursor is placed in the lower left corner of the screen. *endwin* also dismantles data structures and other program entities that were created by *curses* and are no longer needed.

If the program must interact with multiple terminals during operation, *newterm* should be used for each terminal instead of the single call to *initscr*. *newterm* returns a variable of type `SCREEN *` which should be saved and used each time that terminal is referenced. Two file descriptors must be present, one for input, and one for output. Use *endwin* for each terminal prior to program termination to restore previous terminal states and dismantle data structures that were created by *curses* and are no longer needed. During program operation with multiple terminals, *set_term* is used to switch between terminals.

Another initialization function is *longname* which returns a pointer to a static area containing a verbose description of the current terminal upon completion of a call to *initscr*, *newterm*, or *setupterm*.

Option Setting Routines

These routines set up options within *curses*. Arguments specify the window to which the option applies, and the boolean flag which must be **TRUE** or **FALSE** (not 1 or 0) specifies whether the option is enabled or disabled. Default for all functions in this group is FALSE (disabled).

- *clearok(win, boolean_flag)*, when set, clears and redraws the entire screen on the next call to *refresh* or *wrefresh*.
- *idlok(win, boolean_flag)*, when set, allows *curses* to use the insert/delete line features of the terminal if they are available. This feature tends to be visually annoying if used in applications where it is not really needed. Insert/delete character capabilities are always considered by *curses*, and are not related to insert/delete line considerations.
- *keypad(win, boolean_flag)*, when set, enables handling of special keys from the terminal keyboard as single values instead of character sequences.

- *leaveok(win,boolean_flag)*, when set, allows *curses* to ignore cursor position and relocation at the end of an operation. This feature helps simplify program operation when the cursor is not used or cursor position is not important.
- *meta(win,boolean_flag)*, when set, handles characters from the *(getch)* function as 8-bit entities instead of the usual seven. However, this feature has no value if other programs and networks interacting with the data can only pass 7-bit characters.

  This feature is useful for applications where an extended non-text character set is needed and the terminal has a meta shift key available. Curses takes whatever measures are needed to handle the 8-bit input, including the use of raw mode, if necessary. In most cases, the character size is set to 8, parity checking disabled, and 8th-bit stripping is disabled. For the data to continue unaltered, all programs using it must also be capable of handling 8-bit character codes.
- *nodelay(win,boolean_flag)*, when set, makes *getch* a non-blocking call. When enabled, *getch* returns immediately with the value −1 if no input is ready. If not enabled, the program hangs until a terminal key is pressed.
- *intrflush(win,boolean_flag)*, when set, flushes all output in the tty driver queue if an interrupt key (interrupt, quit, or suspend, if available on the system) is pressed on the terminal keyboard. While this capability provides faster interrupt response, the flush destroys the representative relationship between *curscr* and the current physical display contents.
- *typeahead(file_descriptor)*, when set, enables typeahead for the specified file where `file_descriptor` is the terminal input file. A file descriptor value of zero selects *stdin*; −1 disables typeahead checking.
- *scrollok(win,boolean_flag)*, when set, enables scrolling on the specified window whenever the cursor position exceeds the lower boundary of the window (or scrolling region, if set). Boundary crossing results when a newline occurs on the bottom line or a character is placed in the last character position of the bottom line. If *scrollok* is enabled, the window or scrolling region is scrolled up one line, and a *refresh* operation is performed to update the terminal screen. *idlok* must be enabled on the terminal to get a physical scrolling effect on the visible display. If *scrollok* is disabled, the cursor is left on the bottom line, and no advances are allowed beyond the last character position.
- *setscrreg(top,bottom)* and *wsetscrreg(win,top,bottom)* are used to set software scrolling regions within a given window. If this option and *scrollok* are both active, the scrolling region is scrolled up one line and *refresh* is called to update the screen whenever the cursor position is moved beyond the lower limit of the scrolling region in the window. To get a scrolling effect on the terminal screen, *idlok* must also be enabled.

Terminal Configuration Routines

These routines are used to set or disable various operating modes that are supported by the terminal being used.

- *cbreak()* and *nocbreak()* enable and disable single-character mode. When *cbreak* is enabled, characters are received and processed from the terminal keyboard as they are typed. When *nobreak* is active, characters are held by the tty driver until a newline key is received before making the line available to the program. Interrupt and flow control characters are not affected by either option. *cbreak* enabled is the preferred operating mode for most interactive programs. Default is *nobreak* active.

- *echo()* and *noecho()* select direct echoing of characters back to the terminal display as they are received by the tty driver, or transfer the characters to the program without returning them to the terminal display. *noecho* can be used to process incoming text under program control then echo selected characters to a controlled area of the screen or not echo at all.

- *nl()* and *nonl()* select or disable conversion of newline characters into a carriage-return line-feed sequence on output and conversion of incoming return character(s) into newlines. By disabling newline conversions, *curses* can use line-feed capability more effectively, resulting in better cursor motion.

- *raw()* and *noraw()* select or disable raw mode. Raw mode is similar to *cbreak* in that characters are passed to the program as they are typed, but interrupt, quit, and suspend characters are not interpreted, so they do not generate a signal. Raw mode also handles characters as 8-bit entities. BREAK handling is not affected.

- *resetty()* and *savetty()* restore and save tty modes. *savetty* saves the current state in a buffer. *resetty* restores the terminal to the state that was obtained by the last previous call to *savetty*.

Window Manipulation Routines

Window manipulation routines are used to create, move, and delete windows, subwindows, and pads, and perform certain other operations. *newwin*, *newpad*, and *subwin* create new structures. *delwin* deletes window, pad, and subwindow structures, and *mvwin* relocates a window to a different area within the physical screen boundary. *touchwin*, *overlay*, and *overwrite* affect optimization and character replacement during refresh and window copying operations as follows:

- *touchwin* forces the entire window to be rewritten to the screen during refresh.
- *overlay* copies non-blank characters from one window onto the overlapping area of another.
- *overwrite* overwrites all characters from one window onto the overlapping area of another.

Pad functions are related to window functions, with some differences. Pads are essentially the same as windows but usually larger than the available screen size so that only part of the pad can be displayed at any given time. Pads cannot be directly transferred to the terminal screen by use of window *refresh* functions. Pad refresh functions must be used instead, so that the appropriate area of the pad can be specified for display.

When a new window, subwindow, or pad is created, the function returns a pointer that should be stored in a variable for later use when accessing the window or pad. The returned variable then becomes the `win` argument for writing to the window (or pad), deleting the window (or pad), and for other text and cursor operations that include `win` as an argument. Except for *prefresh*, *pnoutrefresh*, and *newpad*, all pad operations use the appropriate window function for all text and cursor manipulations and other pad/window activities.

Terminal Data Output Routines

All data transfers from a pad or window to the terminal display are handled by pad and window refresh/update functions:

- *refresh()* and *wrefresh(win)* transfer the contents of the default or specified window to the current screen window and to the terminal display.
- *doupdate()* and *wnoutrefresh (win)* are used to accumulate several window copy operations to the standard screen window by using multiple calls to *wnoutrefresh (win)*, then transferring the current screen window to the terminal screen by calling *doupdate()*.
- *prefresh(...)* and *pnoutrefresh(...)* are equivalent to *wrefresh* and *wnoutrefresh*, except that the pad and area within the pad are specified. *pnoutrefresh* is followed by the *doupdate* function that is normally used with window updates.

Window Writing Routines

Placing Text in the Window

These routines are used to write data in windows, subwindows, and pads. Only the root function is listed here. Other related functions are listed with the root function in the alphabetical *curses* Routines section later in this tutorial.

Routines that use the `win` argument operate on the *stdscr* window if `win` is not specified. The cursor can be relocated before a function is executed by adding *mv* onto the beginning of the function name. This produces a *move(y,x)* or *wmove(win,y,x)* call on the default or specified window associated with the function, followed by a call to the remaining window writing routine. Row (y) and column (x) coordinates begin with (0,0) in the upper left-hand corner of the window or screen (not (1,1). Use of the *mv* prefix was also discussed earlier. See the section, Using Multiple Windows.

- *move(y,x)* and *wmove(win,y,x)* move the cursor in the given window or pad. *move(y,x)* is equivalent to *wmove(stdscr,y,x)*.
- *addch(ch)* and related functions (see *curses* routines section for related functions) write a single character in the given window or pad. *mv* prefixed to the base function name causes the current cursor/character position to be changed to the specified `y,x` location before the character is placed. Cursor position after the placement is determined by the type of character written.
- *addstr(str)* and related functions place the specified string in the selected window. *mv* prefixed to the base function name causes the current cursor/character position to be changed to the specified `y,x` location before the string is placed. Cursor position after the placement is determined by the characters contained in the written string.

- *erase()* and *werase(win)* place blanks in the entire window or pad, destroying all previous window contents.
- *clear()* and *wclear(win)* are similar to *erase()*. They erase the window by filling it with blanks, but they also call *clearok()* which clears the terminal screen on the next *refresh()* for that window.
- *clrtoeol()* and *clrtobot()* and their related window/pad functions erase the specified window/pad from the present cursor position to the end of the cursor line or to the end of the window or pad, respectively.

Inserting and Deleting Text in the Window

The following routines are used to insert and delete lines and characters in the window. These operations are performed on the window only, and have no effect on the terminal at the time of execution.

- *delch* and related window and move routines delete a single character from the current or specified new cursor position.
- *deleteln()* and *wdeleteln(win)* remove the current cursor line from the default or specified window.
- *insch(c)* and related routines insert the specified character in front of the current cursor position and move succeeding text appropriately to accommodate the new character.
- *insertln()* and *winsertln(win)* insert a blank line at the present cursor line position and move the existing cursor line (and subsequent lines) down one position. The bottom line in the window is lost. The inserted line becomes the new cursor line.

Formatted Output to the Window

printw is functionally similar to *printf* except the output is handled by *addch* which places the formatted data in the window.

Miscellaneous Window Operations

scrollw(win) is used to scroll a given window up one line each time the function is called. *box(win,vert,hor)* uses the specified characters to draw a box around the specified window. When the window is boxed, the top and bottom rows and left and right columns in the window are no longer available for normal text use.

Window Data Input Routines

Two functions are available that are used to obtain data from a given window. *getyx(y,x)* is used to obtain the present cursor position for use by the program. *inch()* and related functions can be used to retrieve any character in a given window. The returned character includes video highlighting attribute bits, each of which is set or cleared according to the original highlighting attributes that were stored with the character when it was written to the window.

Terminal Data Input Routines

getch and its related window and move routines are the basic building block for all program input from the terminal. *getch* handles individual characters, one at a time, returning a character as a 16-bit integer value each time it returns from a call.

If echo is enabled, *getch* also places each character at the current cursor position in the window associated with the function and updates the terminal screen with a *refresh* on the window as the character is received and processed (the cursor is advanced as each character is written to the window). If *noecho* is active instead, input character(s) are not placed in the window.

getstr and its related functions generate a series of calls to *getch* to read an entire line, one character at a time, up to the terminating newline character. The line is stored in the specified string before *getstr* returns to the calling program.

scanw and its related functions perform formatted processing on the input line after it has been placed in a special buffer used by *getstr*. (If echo is enabled, the string is also placed in the associated window, but only the characters stored in the buffer are used by *scanw*. When scanning is complete, the processed results string results are placed in the specified `args` variables.

Video Highlighting Attribute Routines

Each character written into a window is stored as a 16-bit word. Seven bits contain the character code; the remaining nine bits control video highlighting. As each word is stored, the 7-bit character code is combined (through a bit-level logical OR operation) with the current set of nine video highlighting attributes to obtain the 16-bit result. Video attribute routines are used to construct the current attribute set that is used during character storage.

Highlighting attributes can be specified as a complete set by using *attrset* or *wattrset.* Using 0 (or `A_NORMAL`) as an argument for *attrset* disables all highlighting.

Highlighting can be altered from the present state by turning individual attributes on or off without altering the state of other attributes in the set. This is done with *attron*, *attroff*, *wattron*, and *wattroff*.

As characters are stored in a given window, the current attributes are attached to each character. To change highlighting, attributes must be changed before the next character is written to the window. When deciding where to change highlighting attributes, remember that highlighting applies to non-printing space and tab characters as well as visible characters.

standout and *standend* provide easy access to the `A_STANDOUT` attribute. *standout* is equivalent to a call to *attron(A_STANDOUT)*, and adds `A_STANDOUT` to the currently active set of attributes (if any are active). However, *standend* is not the opposite. *standend* is equivalent to *attrset(0)*, not *attroff(A_STANDOUT)*. Thus, a call to *standout* with underlining on would maintain underlining until another highlighting call. *standend*, on the other hand, would not only terminate the previous *standout* call, but would terminate underlining as well.

Attribute functions and arguments must be logically conceived. For example, *attron(A_NORMAL)* and *attroff(A_NORMAL)*, though executable, do nothing because all bits in *A_NORMAL* are cleared (value is zero). The bit-level logical OR of *attron* has no effect (all bits zero), and *attroff* is ineffectual because *A_NORMAL* is inverted (all bits set to 1) before a bit-level logical AND is used to clear the selected highlighting attribute.

Miscellaneous Functions

beep/flash

beep() and *flash()* are used to signal the terminal operator. If the terminal does not support the called function, the other is substituted where possible. Thus a call to *beep* flashes the screen if the terminal has no beep capability; a call to *flash* produces a beep if no flashing video capability is available.

Portability Functions

Several functions have been included to aid portability of *curses* between various systems:

- *baudrate()* returns the terminal datacomm line speed as an integer baud rate value. The returned value can then be used for program and system configuration purposes.
- *erasechar()* returns the terminal erase character that has been chosen by the user. This character is used to cancel the last previous character. Interactive programs should include cancellation capabilities so users can correct typographical errors during keyboard inputs.
- *killchar()* is similar to the erase character, but cancels the entire line where the character appears.
- *flushinp()* discards any typeahead characters when an interrupt character is detected. This enables users to interrupt a series of commands or other activities that have accumulated in the typeahead buffer and terminate the current process without waiting for the typeahead queue to empty. Normally used for aborts, this function and the related program structure must be handled carefully to ensure proper termination of program processes before the program exits.

Delay Functions

Delay functions are not highly portable, but are frequently needed by programs that use *curses*, especially real-time interactive response programs. Use of these functions should be avoided where possible:

- *draino(ms)* is used to reduce the amount of data being held in the output queue. The main purpose of this function is to keep the program (and keyboard) from getting ahead of the screen. With careful program design, use of this function should be unnecessary in most cases.
- *napms(ms)* suspends program operation for a specified time. It is similar to *sleep*, but offers higher resolution (resolution varies, depending on system resources). *napms* uses a call to *select* for its time base reference.

curses Routines

curses supports the following functions. Those marked with an asterisk are also supported by *Mini-curses* (some unmarked routines might work, but are not officially supported by *Mini-curses*. Proceed at your own risk if you try them).

*addch(ch)**
*addstr(str)**
*attroff(attrs)**
*attron(attrs)**
*attrset(attrs)**
*baudrate()**
*beep()**
box(win,vert,hor)
*cbreak()**
*clear()**
clearok(win,boolean_flag)
clrtobot()
clrtoeol()
*delay_output(ms)**
delch()
deleteln()
delwin(win)
doupdate()
draino (ms)
*echo()**
*endwin()**
*erase()**
*erasechar()**
fixterm()
*flash()**
*flushinp()**
getch()
getstr(str)
gettmode()
getyx(win,y,x)
*has_ic()**
*has_il()**
*idlok(win,boolean_flag)**
*inch()**
*initscr()**
insch(c)
insertln()
intrflush(win,boolean_flag)
keypad(win,boolean_flag)
*killchar()**
leaveok(win,boolean_flag)
longname()
*meta(win,boolean_flag)**
*move(y,x)**
*mvaddch(y,x,ch)**
*mvaddstr(y,x,str)**
mvcur(oldrow,oldcol,
newrow,newcol)
mvdelch(y,x)
mvgetch(y,x)
mvgetstr(y,x,str)
mvinch(y,x)
mvinsch(y,x,c)
mvprintw(y,x,fmt,args)
mvscanw(y,x,fmt,args)
mvwaddch(win,y,x,ch)
mvwaddstr(win,y,x,str)
mvwdelch(win,y,x)
mvwgetch(win,y,x)
mvwgetstr(win,y,x,str)
mvwin(win,beg_y,beg_x)
mvwinch(win,y,x)
mvwinsch(win,y,x,c)
mvwprintw(win,y,x,fmt,args)
mvwscanw(win,y,x,fmt,args)
napms(ms)
newpad(num_lines,num_cols)
*newterm(type,fdout,fdin)**
newwin(num_lines,num_cols,
beg_y,beg_x)
*nl()**
*nocbreak()**

nodelay(win,boolean_flag)
*noecho()**
*nonl()**
*noraw()**
overlay(win1,win2)
overwrite(win1,win2)
pnoutrefresh(pad,pminrow,
pmincol,sminrow,
smincol,smaxrow,
smaxcol)
prefresh(pad,pminrow,
pmincol,sminrow,
smincol,smaxrow,
smaxcol)
printw(fmt,args)
*raw()**
*refresh()**
*resetterm()**
*resetty()**
*saveterm()**
*savetty()**
scanw(fmt,args)
scroll(win)
scrollok(win,boolean_flag)
setscrreg(t,b)
setterm(type)
setupterm(term,filenum,errret)
*set_term(new)**
*standend()**
*standout()**
subwin(orig_win,n_lines,
n_cols,beg_y,beg_x)
touchwin(win)
traceoff()
traceon()
typeahead(fd)
unctrl(ch)
waddch(win,ch)
waddstr(win,str)
wattroff(win,attrs)
wattron(win,attrs)
wattrset(win,attrs)
wclear(win)
wclrtobot(win)
wclrtoeol(win)
wdelch(win,c)
wdeleteln(win)
werase(win)
wgetch(win)
wgetstr(win,str)
winch(win)
winsch(win,c)
winsertln(win)
wmove(win,y,x)
wnoutrefresh(win)
wprintw(win,fmt,args)
wrefresh(win)
wscanw(win,fmt,args)
wsetscrreg(win,t,b)
wstandend(win)
wstandout(win)

Description of Routines

The *curses* package includes the following functions. Function names that are associated with operations on user-specified windows contain a *w* or *mvw* prefix, and the window must be included as a parameter in the function call. If no *w* or *mvw* prefix is present, or if the window is not specified in the parameter set, the operation is performed on the default window *stdscr*. Programs that use the *curses* package are subject to the normal rules of C compiler statement syntax.

Routines are listed alphabetically by function keyword which is printed in slanted bold type. When two or more functions are related to a common keyword, the root keyword is listed in bold, followed by a list of related function names in normal italics. The individual related functions are also included elsewhere in the list with references back to the root keyword where a detailed explanation of all keywords related to the root keyword is located.

addch*(ch)*
  waddch(win,ch)
  mvaddch(y,x,ch)
  mvwaddch(win,y,x,ch)

Places the character `ch` in the window at the current cursor position for that window, then advances the cursor to the next position. If `ch` is a tab, newline, backspace, the cursor is moved appropriately, but no text is altered. If `ch` is a control character other than tab, newline, or backspace, the character is drawn using ^x notation (where `x` is a printable character preceded by ^ to indicate a control character – see *unctrl(ch)*). If the character is placed at the right margin, an automatic newline is performed. At the bottom of the scrolling region, the region is scrolled up one line if *scrollok* is enabled.

The *ch* parameter is an integer; not a character. *addch* performs a bit-level logical OR between the 16-bit character and the current attributes if any are active. Highlighting of individual characters can also be handled by the program if the current attributes are all zero (disabled) by performing an equivalent bit-level logical OR operation between the 7-bit character code in bit positions 0 through 6 and selected video attribute bits in bit positions 7 through 15 to create a single 16-bit integer representing the character and its associated highlighting attributes. If no highlighting attributes for the window are currently active, any attributes added to the character by the program or already present from the source are preserved. If any are active, they are added to the character and any attached attributes without altering other attributes. Thus, you can copy text (including attributes) from one place to another with *inch* and *addch*.

addch is used with *stdscr* window; *waddch* with window *win*; *mvaddch* moves the cursor to row Y, column X, then places the character at that location; *mvwaddch* is identical to *mvaddch*, but operates on a specified window win. If win is not specified, default is to *stdscr*. All row and column references are relative to the upper left corner whose corner character position is represented by row 0, column 0.

addstr*(str)*
waddstr(win,str)
mvaddstr(y,x,str)
mvwaddstr(win,y,x,str)
Places the character string specified by **str** at the current cursor position (*addstr* and *waddstr*) or at the specified location in the window (*mvaddstr* and *mvwaddstr*). String placement consists of a series of character placements using the *addch* routine. **str** must be terminated by a null character.

attroff*(attrs)*
wattroff(win,attrs)
Disables the specified video highlighting attributes without affecting other attributes. Any or all of the following attributes can be specified (multiple attributes must be separated by the C logical OR operator, | which performs a bit-level logical OR on all attributes specified in the function call): **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS** (invisible), **A_PROTECT**, and **A_ALTCHARSET**.

attron*(attrs)*
wattron(win,attrs)
Enables the specified video highlighting attributes without affecting other attributes. Any or all of the following attributes can be specified (multiple attributes must be separated by the C logical OR operator, | which performs a bit-level logical OR on all attributes specified in the function call): **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS** (invisible), **A_PROTECT**, and **A_ALTCHARSET**.

attrset*(attrs)*
wattrset(win,attrs)
Enables the specified video highlighting attributes, and disables all others. Any or all of the following attributes can be specified (multiple attributes must be separated by the C logical OR operator, | which performs a bit-level logical OR on all attributes specified in the function call): **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS** (invisible), **A_PROTECT**, and **A_ALTCHARSET**. *attrset(0)*, *attrset(A_NORMAL)*, and *standend()* (or *stan dend(win)*) are equivalent functions that disable all attributes (normal display). See *standend()*.

baudrate()
Returns the terminal serial I/O datacomm speed. The value returned is the integer baud rate (such as 9600) rather than a table index value (such as B9600). If the baud rate is External A or External B, the value −1 is returned instead.

beep()
Used to signal the terminal user with an audible signal. If no audible signal is available on the terminal, the screen is flashed instead (see *flash()*). If neither capability is available, no output is sent to the terminal.

box*(win,vert,hor)*
Draws a box around the specified window. `vert` specifies the character to be used for left and right columns; `hor` specifies the character for top and bottom rows. Usable window space is reduced by two lines and columns when a box is present.

cbreak()
nocbreak()
These functions place the terminal in and out of `CBREAK` mode, respectively. When *cbreak* (character-mode operation) is active, each typed character is immediately available to the program. If disabled (*nocbreak*), the tty driver holds characters until a newline character is received, then releases the entire line to the program (line-mode operation). Interrupt and flow control characters are not affected by *cbreak*; default is *nocbreak*, but most interactive programs that use *curses* run with *cbreak* enabled.

clear()
wclear(win)
Similar to *erase* and *werase*, but *clearok* is also called so that the terminal screen is cleared by the next call to *refresh* for that window. *clearok* sets a flag to clear the screen, blanks are placed in the window, and the next call to *refresh* outputs a screen clearing operation or blanks or both to the terminal, depending on terminal capabilities.

clearok*(win,boolean_flag)*
If set, the next *wrefresh* call for the specified window clears and redraws the entire screen (instead of just the area represented by the specified window). If `win` specifies *curscr*, the next call to *wrefresh* for **any** window clears and redraws the entire screen. This is useful when current screen contents are uncertain, or in some cases for a more pleasing visual effect.

cleartobot()
wcleartobot(win)
Clears all character positions from the current cursor position to the right margin, and all lines below the current cursor line to the end of the window.

***cleartoeol**()*
wcleartoeol(win)
Clears all character positions from the current cursor position to the right margin. The rest of the window remains undisturbed.

***delay_output**(ms)*
See *terminfo* routines in the next section of this tutorial.

***delch**()*
wdelch(win)
mvdelch(y,x)
mvwdelch(win,y,x)
The character at the present cursor position is deleted. All remaining characters on the line to the right of the deleted character are moved left one position. Other lines are not disturbed. The operation is performed only on the window, and does not use the terminal hardware delete-character feature because no terminal operation has been performed.

***deleteln**()*
wdeleteln(win)
The present cursor line is deleted. All remaining lines in the window below the cursor line are moved up one position, leaving a blank line at the bottom of the window. This window operation does not interact directly with the terminal when performed, so no terminal hardware delete-line feature is used.

***delwin**(win)*
Deletes the specified window and releases all memory associated with it. If the window contains subwindows, all subwindows must be deleted first.

doupdate()
wnoutrefresh(win)
pnoutrefresh(pad,...)
wnoutrefresh (or *pnoutrefresh*) and *doupdate* essentially divide *wrefresh* into two independent functions that can be called separately for more efficient handling of multiple output operations to windows and pads. In normal operation, *wrefresh(win)* calls *wnoutrefresh(win)* to copy the named window to the virtual screen, then uses *doupdate* to update the physical screen to match the virtual screen. When outputting multiple windows, *wnoutrefresh(win)* can be used successively, once for each window; followed by a single *doupdate()* to transfer the new screen to the terminal, probably with fewer characters transmitted. *pnoutrefresh* is used similarly when writing to pads.

***draino**(ms)*

Suspends program operation until the output queue has been reduced sufficiently ("drained") so that the remaining characters can be transmitted in not more than `ms` milliseconds. For example, *draino(50)* at 1200 baud would suspend program execution until no more than 6 characters remain to be sent (6 characters @ 1200 baud require about 50 ms transmit time). This routine is used to keep the program (and thus the keyboard) from getting ahead of the screen. If the operating system does not support the I/O controls (ioctls) that are needed to implement *draino*, the value ERR is returned; otherwise OK is returned.

***echo**()*

noecho()

Enables or disables echoing of characters by *getch* through the specified window and back to the terminal as each character is typed on the keyboard and subsequently processed by *getch*. Default is *echo* (enabled). In some interactive programs, it is preferable to suppress echoing by *getch* (*noecho*), then let the program place incoming characters in a controlled area of the screen or not return them at all, as needs dictate.

***endwin**()*

endwin should always be called before exiting from a *curses*-based program. Restores tty modes, places the cursor in the lower left corner of the terminal screen, resets the terminal into the proper n on-visual mode, and removes data structures that are no longer needed by the exiting program.

***erase**()*

werase(win)

Copies blanks to every character position in the specified or default window. As each blank is stored in the window, the highlighting attribute bits are set to zero (disabled).

***erasechar**()*

Returns the user's chosen erase character from the terminfo data base. The returned character should be interpreted by the program as an "erase previous character" command whenever it is received from the terminal.

***fixterm**()*

Restores the current terminal to the state it was in prior to the most recent call to *resetterm()*. State information stored by the most recent previous call to *saveterm()* provides the needed restoration information. See *resetterm()*.

flash()
Used to signal the terminal user by flashing the screen. If the terminal has no screen flashing feature, the audible signal is sounded instead (see *beep()*). If neither capability is available, no output is sent to the terminal.

flushinp()
Discards any typeahead characters in the typeahead buffer (characters that have been typed on the terminal but are still waiting to be handled by the program.

getch()
wgetch(win)
mvgetch()
mvwgetch(win)
Takes a character from the terminal keyboard input buffer as a 16-bit integer, processes it, and returns it to the program as a 16-bit integer. Character processing and return conditions vary as follows:

If *mv* is placed in front of *getch* or *wgetch*, the cursor position for the selected window is moved to the specified location which becomes the new current cursor position. This operation is completed before any character processing begins.

If *echo* is active and the character is a normal typing character (keypad and meta characters are discussed later), the character is placed in the current cursor position by a call to *waddch* from *getch*. During character placement in the window, a bit-level logical OR in *waddch* attaches current highlighting attributes to the character. *waddch* is followed immediately by a call to *wrefresh* which updates the terminal screen with the echo character.

If an escape character is received, special timeouts are set up to determine whether the character is part of a multiple-character keypad sequence. See Use of Escape in Program Control topic earlier in this tutorial for a detailed discussion of how escape is handled.

If **meta** is enabled and the character is not a keypad sequence, the 16-bit input character is logical ORed with octal 0377 to mask the upper bits to zero and return an 8-bit text character value. The eighth bit interferes with the *A_STANDOUT* highlighting attribute bit in the same position, so *noecho* is usually chosen for programs that operate with meta active.

If **meta** is not enabled, text characters are logical ORed with octal 0177 to mask the upper bits to zero and return a 7-bit character value. Echoing is handled in the normal manner if enabled.

If **keypad** is **not enabled**, function key sequences are treated as individual characters and handled as normal text.

If **keypad** is **enabled**, each function key sequence (usually an escape sequence) is handled as a single-character keycode which is assigned a 16-bit integer value in a range beginning at 0401 (octal) and a name that starts with *KEY_* (a complete list of keypad character value and name definitions is included in the keypad discussion near the beginning of this tutorial). The character value is **not** placed in the window for echoing, even if echo is enabled.

If **nodelay** is active: if no input is available in the keyboard input buffer when *getch* is called, *getch* returns with the value −1 and no other action is taken. If *nodelay* is not active, the program hangs until text is available in the buffer. Depending on the current ***cbreak*** setting, text is made available to the program as each character is received (*cbreak*), or incoming characters are held by the tty driver until a newline is received then they are made available to the program (*nocbreak*).

getstr*(str)*
wgetstr(win,str)
mvgetstr(y,x,str)
mvwgetstr(win,y,x,str)
This routine is used to input an entire line from the terminal. It is equivalent to *getch*, except that it handles an entire string instead of single characters. Handling of each character is identical to *getch* except that text and meta characters are packed into the string variable *str* instead of being returned to the program as individual 16-bit integers. Keypad characters (except for kill, erase, key_left (left arrow), and backspace) are not recognized and cannot be handled through *getstr*.

During execution, *getstr* generates a series of calls to *getch* until a newline is received, at which time it returns. The 16-bit integers returned by successive calls to *getch* are stripped of their unneeded upper bits (except recognized keypad keys) before packing into a string variable beginning at the location identified by the character pointer *str*.

If **echo** is enabled, incoming string characters are also placed in the associated window (by *getch*) as they are received and processed, and echoed to the terminal (by *refresh*). If *noecho* is active, characters are not placed in the window; they are only placed in *str*.

gettmode*()*
(Get tty mode). Dummy entry point. Performs no useful function.

getyx*(win,y,x)*
Places the current cursor position of the specified window in the specified two integer variables `y` and `x`. This is a macro, so no `&` is necessary.

has_ic*()*
Returns a value indicating whether or not the terminal has insert/delete character capability. Zero value indicates the capability is not present; non-zero: capability present.

has_il*()*
Returns a value indicating whether or not the terminal has insert/delete line capability. Zero value indicates the capability is not present; non-zero: capability present.

idlok*(win,boolean_flag)*
Insert and Delete Line OK. If enabled, *curses* can use hardware insert/delete line capabilities when the terminal is so equipped. If disabled, *curses* does not use the capability. Use only when the program requires it (such as a screen editor). *idlok* is disabled by default because it tends to be annoying when used in applications where it is not really needed. If insert/delete line cannot be used, *curses* redraws changed portions of all lines that do not match the desired result.

inch*()*
winch(win)
mvinch(y,x)
mvwinch(win,y,x)
Returns the character located at the current or specified position in the specified window as a 16-bit integer. If any attributes are set for that position, their values are included in the value returned. To extract only the character or the attributes, perform a bit-level logical AND on the returned value, using the predefined constant `A_CHARTEXT` (octal 0177) or `A_ATTRIBUTES` (octal 0177600).

initscr*()*
The first function called in *curses*-based programs. Determines terminal type, and initializes *curses* data structures as appropriate. Also sets indicators so that the first call to *refresh* clears the terminal screen and updates *curscr* to reflect the cleared screen.

insch*(c)*
winsch(win,c)
mvinsch(y,x,c)
mvwinsch(win,y,x,c)
Inserts the character (byte, usually a 7-bit code) specified by `c` at the current cursor position position or at the specified location in the standard or specified window (current attributes are attached during the placement operation). All characters beginning at the

insertion location are moved right one position for the remainder of the line. If the line is full, the rightmost character is discarded. This does not interact with the terminal so no hardware insert-character feature is used.

insertln*()*
winsertln(win)
Inserts a blank line between the current cursor line and the line above it. The current line and subsequent lines of text in the window are moved down one position, and the blank line becomes the new current cursor line. The bottom line of text is discarded if it cannot fit inside the window. This is a window operation that does not interact with the terminal, so no hardware insert-line feature is used.

intrflush*(win,boolean_flag)*
Causes tty driver queue to be flushed on interrupt. When enabled, an interrupt, quit, or suspend keypress from the terminal flushes all output from the tty driver queue, providing a faster response to the interrupt. However, *curses* loses its record of what is currently displayed on the screen when the interrupt occurs. Disabling the option prevents the flush. Default is flush enabled. Requires proper support from the underlying driver.

keypad*(win,boolean_flag)*
Enables keypad character handling for the user terminal associated with `win`. When true, the terminal operator can press any key that generates multiple-character sequences (such as a function key), and *getch* returns a single 16-bit integer value representing the function key (the returned character must be handled as a 16-bit value). If *keypad* is disabled (default), *curses* handles keypad sequences as normal text. *keypad* also enables and disables keypad keys on the terminal if the terminal hardware is equipped to support such command sequences from the external computer.

killchar*()*
Returns the line-kill character chosen by the terminal user. This character, when typed by the user, is a command to the program to cancel the entire line being typed.

leaveok*(win,boolean_flag)*
Upon completion of normal *refresh* operations (*leaveok* disabled) the terminal hardware cursor is placed at the current cursor location for the window being refreshed. A call to *leaveok(win,* `TRUE`*)* prior to *refresh* allows refresh operations to leave the terminal hardware cursor in any convenient position instead of updating it to the current window cursor position when refresh is finished. This is useful for applications where the cursor is not used because it reduces the need for cursor movements. When possible, the cursor is made invisible when *leaveok* is specified for the window. Once *leaveok* is set `TRUE` for a given window, it remains active for the duration of the program or until another call sets it `FALSE`.

longname*()*
Returns a pointer to a static area containing a verbose description of the current terminal. This static area is defined only after a call to *initscr*, *newterm*, or *setupterm*.

meta*(win,boolean_flag)*
When enabled, text characters are returned by *getch* as 8-bit character codes (masked by octal 0377) instead of 7-bit (masked by octal 0177) characters. Returns the value `OK` if the request succeeds; `ERR` if the terminal or system cannot handle 8-bit character codes.

meta is useful for extending the non-text command set in applications where the terminal has a meta shift key. *curses* takes whatever measures are necessary to arrange for 8-bit input. When *meta* is true, HP-UX sets datacomm configuration to 8-bit character length, no parity checking, and disables 8th-bit stripping. Remember that if any program or facility handling the data can only pass 7-bit codes or strips the 8th bit, 8-bit handling is not possible.

move*(y,x)*
wmove(win,y,x)
Places the cursor associated with the specified or default window at the specified row (`y`) and column (`x`) where the upper left corner of the window is row 0, column 0. The cursor is not moved on the display screen until a *refresh* or equivalent function is executed.

mvaddch*(y,x,ch)*
Same as *move(y,x); addch(ch)*. See *addch(ch)*.

mvaddstr*(y,x,str)*
Same as *move(y,x); addstr(str)*. See *addstr(str)*.

mvcur*(oldrow,oldcol,newrow,newcol)*
Optimally moves the cursor from (`oldrow`, `oldcol`) to (`newrow`, `newcol`). The user program is expected to keep track of the current cursor position. Unless a full-screen image is kept, *curses* must make pessimistic assumptions that sometimes result in less than optimal cursor motion. For example, if the cursor needs to be moved a few spaces to the right, the task could be accomplished by retransmitting the characters between the present and the desired position; but if *curses* cannot access the screen image, it cannot determine what those characters are.

mvdelch*(y,x)*
Same as *move(y,x); delch()*. See *delch()*.

mvgetch*(y,x)*
Same as *move(y,x); getch()*. See *getch()*.

mvgetstr*(y,x,str)*
Same as *move(y,x); getstr(str)*. See *getstr(str)*.

mvinch*(y,x)*
Same as *move(y,x); inch()*. See *inch()*.

mvinsch*(y,x,c)*
Same as *move(y,x); insch(c)*. See *insch(c)*.

mvinsch*(y,x,c)*
Same as *move(y,x); insch(c)*. See *insch(c)*.

mvprintw*(y,x,fmt,args)*
Same as *move(y,x); printw(fmt,args)*. See *printw (fmt,args)*.

mvscanw*(y,x,fmt,args)*
Same as *move(y,x); scanw(fmt,args)*. See *scanw(fmt,args)*.

mvwaddch*(win,y,x,ch)*
Same as *wmove(win,y,x); waddch(win,ch)*. See *addch(ch)*.

mvwaddstr*(win,y,x,str)*
Same as *wmove(win,y,x); waddstr(win,str)*. See *addstr (str)*.

mvwdelch*(win,y,x)*
Same as *wmove(win,y,x); addch(ch)*. See *delch()*.

mvwgetch*(win,y,x)*
Same as *wmove(win,y,x); wgetch(win)*. See *getch()*.

mvwgetstr*(win,y,x,str)*
Same as *wmove(win,y,x); wgetstr(win,str)*. See *getstr(str)*.

mvwin*(win,beg_y,beg_x)*
Moves the specified window so that the upper left-hand corner is located at character position (`beg_y`, `beg_x`). If the move causes any part of the relocated window to lie outside the physical screen boundary, the command is considered to be in error, and the window remains in its original location.

mvwinch*(win,y,x)*
Same as *wmove(win,y,x); winch(win)*. See *inch()*.

mvwinsch*(win,y,x,c)*
Same as *wmove(win,y,x); winsch(win,c)*. See *insch(c)*.

mvwprintw*(win,y,x,fmt,args)*
Same as *wmove(win,y,x); wprintw(win,fmt,args)*. See *printw(fmt,args)*.

mvwscanw*(win,y,x,fmt,args)*
Same as *wmove(win,y,x); wscanw(win,fmt,args)*. See *scanw(fmt,args)*.

napms*(ms)*
Suspends program operation for `ms` milliseconds. *napms* is similar to *sleep*, but has higher resolution. The resolution actually provided depends on the resolution of available operating system facilities. If a resolution of at least 0.100 sec is not available, the routine rounds to the next higher second, calls sleep, and returns ERR. Other wise the value OK is returned.

newpad*(num_lines, num_cols)*
Creates a new **pad** data structure. A pad is similar to a window, but it is not restricted by physical screen size nor is it associated with a particular part of the screen. Pads are useful when a large window is needed and only part of the window will be displayed at any given time. Automatic refreshes from pads (such as scrolling or input echo) do not occur. Refresh cannot be used with a pad as an argument. Instead, the routines *prefresh* and *pnoutrefresh* are used. Pad refresh routines require additional parameters to specify what part of the pad to display, and where to display it on the screen.

newterm*(type,fpout,fpin)*
Used instead of *initscr* in programs that output to more than one terminal. *newterm* should be called once for each terminal. It returns a variable of type `struct screen *` which should be saved for use as a reference to that terminal. Arguments are: a string defining the terminal type, a file pointer for the output file, and another for the input file if needed (interactive terminal).

newwin*(num_lines, num_cols,beg_y,beg_x)*
Create a new window with the specified number of lines and columns whose upper left-hand corner is located at the specified row and column of the physical screen, and return a window pointer (the upper left-hand corner of the physical screen is row 0, column 0). If the number of lines and/or columns is specified as zero, the default value *LINES* minus `beg_y` and *COLS* minus `beg_x` is used instead. A screen buffer for the window is also created. To create a new full-screen window, use *newwin(0,0,0,0)*.

nl()
nonl()
Defines handling of newline characters. When enabled (*nl*), newline is translated into a carriage-return and line-feed on output, and carriage-return is translated into a newline character on input. *curses* initially enables newline, but if it is disabled by *nonl*, *curses* can make better use of line feed capability, resulting in faster cursor motion.

nocbreak()
See *cbreak()*.

nodelay(*win,boolean_flag*)
Makes *getch* a non-blocking call. When enabled, if no input is ready, a call to *getch* returns −1. If disabled, *getch* hangs until a key is pressed.

noecho()
See *echo()*.

nonl()
See *nl()*.

noraw()
See *raw()*.

overlay(*win1,win2*)
overwrite(win1,win2)
Copies `win1` onto `win2` for all screen area where the two windows overlap. *overlay* copies only visible (non-blank) text, and does not disturb those `win2` character positions where `win1` is blank. *overwrite* copies all of overlapping `win1` onto `win2`, including blanks, thus destroying all original data in the overlapping area of `win2`.

overwrite(*win1,win2*)
See *overlay*.

pnoutrefresh(*pad, pminrow,pmincol, sminrow,smincol, smaxrow,smaxcol*)
See *prefresh*.

prefresh(*pad, pminrow,pmincol, sminrow,smincol, smaxrow,smaxcol*)
pnoutrefresh (pad, pminrow,pmincol, sminrow,smincol, smaxrow, smaxcol)
Analogous to *wrefresh* and *wnoutrefresh*, except that pads are involved instead of windows. Additional parameters specify what part of the pad and screen are to be used. `pminrow` and `pmincol` identify the upper left corner of the pad area to be displayed. `sminrow`, `smincol`, `smaxrow`, and `smaxcol` de fine the display boundaries on the physical screen.

The lower right-hand corner of the pad area being displayed is calculated from the screen boundary parameters because both rectangles must be the same size. Both rectangles must lie completely within their respective structures.

printw*(fmt, args)*
wprintw(win,fmt, args)
mvprintw(y,x,fmt, args)
mvwprintw(win,y,x, fmt, args)
These commands are functionally equivalent to *printf.* Characters that would normally be output by *printf* are instead output by *waddch* on the associated window.

raw*()*
noraw()
Places the terminal in or out of raw mode. Raw mode is similar to *cbreak* mode in that characters are immediately passed to the user program as they are typed on the terminal keyboard, except that interrupt and quit characters are passed as normal text instead of generating a special interrupt signal. Raw mode handles all terminal I/O as 8-bit characters instead of 7. BREAK key behavior may vary, depending on the terminal.

refresh*()*
wrefresh(win)
These functions output window data to the terminal (other routines only manipulate data structures). *wrefresh* copies the named window to the physical screen on the terminal by using *wnoutrefresh(win)* followed by *doupdate()*, taking into account what is already on the screen in order to optimize the transfer. *refresh()* is similar, except it uses *stdscr* as the default screen. Unless *leaveok* is enabled, the cursor is placed at the location of the window cursor when the operation is complete.

resetterm*()*
saveterm()
fixterm()
resetterm restores the current terminal to the operating condition it was in when *curses* was started. The "current *curses* state" is saved by *saveterm()* for possible future use by *fixterm(). resetterm* and *fixterm* should be used in all shell escapes. Equivalent routines are also available at the *terminfo* level.

resetty*()*
savetty()
Restores (resets) the tty modes to those stored in the buffer by the last previous *savetty()* command. This means that only one set of states can be stored at any given time. See *savetty()*.

***saveterm**()*
Preserves the current terminal *curses* state for possible future use by *fixterm.* See *resetterm()* and *fixterm().*

***savetty**()*
Saves the current state of the tty modes in a buffer for possible later use by *resetty().* See *resetty().*

***scanw**(fmt,args)*
wscanw(win,fmt,args)
mvscanw(y,x,fmt,args)
mvwscanw(win,y,x,fmt,args)
Corresponds to *scanf(3S). Calls wgetstr* which inputs characters from the terminal and places them in a buffer until newline is received. When newline is received, the string in the buffer serves as input for the scan which processes the buffered string and places the result in the appropriate `args`. Uses *getch* for character input and echo handling.

***scroll**(win)*
Scrolls the window up one line by moving the lines in the window data structure. As an optimization, if the window being scrolled is *stdscr*, and the scrolling region is the entire window, the physical screen is scrolled at the same time.

***scrollok**(win,boolean_flag)*
Controls window handling when the cursor advances beyond the bottom boundary of the window or scrolling region due to a newline in the bottom line or a character placed in the last character position of the bottom line. If scrolling is disabled, the cursor is left on the bottom line (characters are accepted until the bottom line is full, but newlines are ignored). If the cursor crosses the bottom boundary while *scrollok* is enabled, a *wrefresh* is performed on the window, then the window and terminal are scrolled up one line. *idlok* must also be called before a physical scrolling effect can be produced on the terminal screen.

***setscrreg**(t,b)*
wsetscrreg(win,t,b)
Sets up a software scrolling area in window `win` or *stdscr.* `t` and `b` are the top and bottom lines of the scrolling region (line 0 is the top line of the window). If this option and *scrollok* are both enabled, an attempt to move off the bottom margin causes all lines in the scrolling region to scroll up one line. Note that this process has nothing to do with the physical scrolling region capability that exists in some terminals (only the text in the window is scrolled). If the terminal has scrolling region or insert/delete line capabilities, they will probably be used by the output routines during refresh. *idlok* must be enabled before a scrolling effect can be produced on the terminal screen (see *scrollok*).

setterm*(type)*
Low-level interface used by old *curses* and included here for compatibility with earlier software.

setupterm*(term,filenum,errret)*
terminfo routine. See *terminfo* routines in the next section of this tutorial for description.

set_term*(new)*
Switches to a different terminal. The screen reference `new` becomes the new current terminal, and the function returns the previous terminal. All other calls affect only the current terminal. This function is used to handle multiple terminals interacting with a single program.

standend*()*
wstandend(win)
Equivalent to *attrset(0)* and *attrset(A_NORMAL)*. Turns off all video highlighting attributes for the default (*stan-
dend*) or specified (*wstandend*) window.

standout*()*
wstandout(win)
Equivalent to *attron(A_STANDOUT)*. Turns on the video highlighting attributes used for standout highlighting for the terminal being used. Does not alter other attributes in effect at the time. *standout* applies to the default window *stdscr*. *wstandout* affects the specified window.

subwin*(orig_win,num_lines,num_cols, beg_y,beg_x)*
Creates a new window containing the specified number of lines and columns within existing window `orig_win`. `beg_y` and `beg_x` specify the starting row and column position of the window on the physical screen (not relative to window `orig_win`). The subwindow uses that part of the main window character data storage structure that corresponds to its own area (each window maintains its own pointers, cursor location, and other items pertaining to window operation; only character storage is shared). Thus, the subwindow always contains character data (including highlighting attributes) that is identical to the data contained in the corresponding area of the original window, regardless of which window is the target of a write operation (highlighting bits are determined by the current attributes in effect for the window through which each character was stored). When using subwindows, it is often necessary to call *touchwin* before *refresh* in order to maintain correct display contents.

touchwin*(win)*
Discards optimization information on the specified window so that the entire window must be completely rewritten during refresh. This is sometimes necessary when using overlapping windows because changes to one window do not update the overlapping window structure in such a manner that a subsequent refresh operation can be handled correctly.

traceoff*()*
Dummy entry point. Performs no useful function.

traceon*()*
Dummy entry point. Performs no useful function.

typeahead*(fd)*
Sets the file descriptor for typeahead check. `fd` is an integer obtained from *open* or *fileno*. Setting typeahead to −1 disables typeahead check. Default file descriptor is 0 (standard input). Typeahead is checked independently for each screen; for multiple interactive terminals, it should be set to the appropriate input for each screen. A call to *typeahead* always affects only the current screen.

unctrl*(ch)*
Converts the character code represented by `ch` into a printable form if it is an unprintable control character. The converted character is printed as an alpha-numeric character preceded by ˆ where (ˆ) represents the control key, and the alpha-numeric character corresponds to the key that is pressed in conjunction with the control key to produce the control character.

waddch*(win,ch)*
See *addch(ch)*.

waddstr*(win,str)*
See *addstr(str)*.

wattroff*(win,attrs)*
See *attroff(attrs)*.

wattron*(win,attrs)*
See *attron(attrs)*.

wattrset*(win,attrs)*
See *attrset(attrs).*

wclear*(win)*
See *clear().*

wcleartobot*(win)*
See *cleartobot().*

wcleartoeol*(win)*
See *cleartoeol().*

wdelch*(win)*
See *delch().*

wdeleteln*(win)*
See *deleteln().*

werase*(win)*
See *erase().*

wgetch*(win)*
See *getch()*

wgetstr*(win,str)*
See *getstr(str)*

winch*(win)*
See *inch()*

winsch*(win,c)*
See *insch(c).*

winsertln*(win)*
See *insertln().*

wmove*(win,y,x)*
See *move(y,x).*

wnoutrefresh*(win)*
See *doupdate().*

wprintw*(win,fmt,args)*
See *printw(fmt,args).*

wrefresh*(win)*
See *refresh().* See also *doupdate().*

wscanw*(win,fmt,args)*
See *scanw(fmt,args).*

wsetscrreg*(win,t,b)*
See *setscrreg(t,b).*

wstandend*(win)*
See *standend().*

wstandout*(win)*
See *standout().*

Terminfo Routines

delay_output*(ms)*
Inserts a delay into the output stream for the specified number of milliseconds by inserting sufficient pad characters to effect the delay. This should not be used in place of a high-resolution *sleep*, but rather to slow down or hold off the output. Due to system buffering, it is unlikely that a delay can result in a process actually sleeping. `ms` should not exceed about 500 because of the large number of pad characters used to produce such delays.

putp*(str)*
Outputs a string capability without use of an `affcnt` (see *tputs*). The string is sent to *putchar* with an `affcnt` of 1. It is used in simple applications that do not require the output processing capability of *tputs*.

setupterm*(term,filenum,errret)*
Initializes the specified terminal. `term` is the character string representing the name or model of the terminal; `filenum` is the HP-UX file descriptor of the terminal being used for output; `errret` is a pointer to the integer in which a success/failure indication is returned. The values returned can be: 1 (initialize complete); −1 (*terminfo* data base not found); or 0 (no such terminal).

If 0 is given as the value of `term`, the default value of TERM is obtained from the enviroment. `errret` can be specified as 0 if no error code is wanted. If `errret` is default (0), and something goes wrong, `setupterm` prints an appropriate error message and exits rather than returning. Thus, a simple program can call *setupterm(0,1,0)* and not provide for initialization errors.

If the environment variable TERMINFO is set to a path name, *setupterm* checks for a compiled *terminfo* description of the terminal under that path before checking */etc/term*. Otherwise, only */etc/term* is checked.

setupterm uses `filenum` to check the tty driver mode bits, and changes any that might prevent correct operation of low-level *curses* routines. Tabs are not expanded into spaces because various terminals exhibit inconsistent uses of the tab character. If the HP-UX system is expanding tabs, *setupterm* removes the definition of the `tab` and `backtab` functions because they may not be set correctly in the terminal. Other system-dependent changes such as disabling a virtual terminal driver may also be made here, if deemed appropriate by *setupterm*.

setupterm also initializes the global variable `ttytype` (an array of characters) to the value of the list of names for the terminal in question. The list is obtained from the begining of the *terminfo* description.

Upon completion of *setupterm*, the global variable `cur_term` points to the current structure of terminal capabilities. A program can use two or more terminals at once by calling *setupterm* for each terminal, and saving and restoring `cur_term`.

nl() is enabled, so newlines are converted to carriage return-line feed sequences on output. Programs that use *cursor_down* or *scroll_forward* should avoid these two capabilities or disable the mode with *nonl()*. *setupterm* calls *reset_prog_mode* after any changes are made.

tparm*(instring,p1,p2,p3,p4,p5,p6,p7,p8,p9)*
Instantiates a parameterized string. Up to nine parameters can passed (in addition to the input string) that define what operations are to be performed on `instring` by *tparm*. The resultant string is suitable for output processing by *tput*.

tputs*(cp,affcnt,outc)*
Processes *terminfo(5)* capability strings for terminal devices. The padding specification, if present, is replaced by enough padding characters to produce the specified time delay. The resulting string is passed, one character at a time, to the routine *outc* which expects a single character parameter each time it is called. Often, *outc* simply calls *putchar* to complete its task. `cp` is the capability string, and `affcnt` is the number of units affected (such as lines or characters). For example, the `affcnt` for *insert_line* is the number of lines on the screen below the inserted line; that is, the number of lines that will have to be moved on the terminal. In certain cases, `affcnt` is used to determine the number of padding characters that must be created in the output string to produce the required delay(s), based on known terminal characteristics (obtained from the terminal identification data base).

vidattr*(attrs)*
Transmits the appropriate string to *stdout* to activate the specified video attributes which can include any or all of the following: `A_STANDOUT`, `A_UNDERLINE`, `A_REVERSE`, `A_BLINK`, `A_DIM`, `A_BOLD`, `A_BLANK` (invisible), `A_PROTECT`, and `A_ALTCHARSET` (multiple attributes must be separated by the C logical OR operator `|`).

***vidputs**(attrs,putc)*
Transmits the appropriate string to the terminal, activating the specified video highlighting attributes. `attrs` can include any or all of the following (multiple attributes must be separated by the C logical OR operator `|`): `A_STANDOUT`, `A_UNDERLINE`, `A_REVERSE`, `A_BLINK`, `A_DIM`, `A_BOLD`, `A_BLANK` (invisible), `A_PROTECT`, and `A_ALTCHARSET`. `putc` is a *putchar-like* function. Previous highlighting attributes are preserved by this routine and restored upon return.

Termcap Compatibility Routines

Several routines have been included in *curses* that support programs written with calls to *termcap* routines. Calling parameters are the same as for equivalent *termcap* calls, but the routines are emulated using the *terminfo* data base. These routines may be removed in future releases of HP-UX.

tgetent(bp,name)	Obtains and returns with *termcap* entry for `name`
tgetflag(id)	Returns the boolean entry for `id`.
tgetnum(id)	Returns the numeric entry for `id`.
tgetstr(id,area)	Returns the string entry for `id` and places the result in `area`.
tgoto(cap,col,row)	Attaches `col` and `row` parameters to the capability `cap`.
tputs(cap,affcnt,fn)	Equivalent to the *terminfo* routine *tputs*. Parameters are identical for both cases.

Program Operation

This section describes how *curses* routines behave and how they are used in a typical programming environment.

Insert/Delete Line

The output optimization routines associated with *curses* use terminal hardware insert/delete line capabilities provided the routine

```
idlok(stdscr,TRUE);
```

has been called to enable the capability. By default, insert/delete line during refresh is disabled (`FALSE`); not for performance reasons (there is no speed penalty involved), but because experience has shown that not only is insert/delete line frequently not needed (especially in simple programs); it can sometimes be visually annoying when used by *curses.* Insert/delete character is **always** available to *curses* if it is supported by the terminal.

Additional Terminals

Curses can be used, even when absolute cursor addressing is not provided on the terminal, as long as the cursor can be moved from any location to any other location. *curses* considers available cursor control options such as local motions, parameterized motions, home, and carriage return.

curses is intended for use with full-duplex, alphamumeric, video display terminals. No attempt is made to handle half-duplex, synchronous, hard copy, or bitmapped terminals. Bitmapped terminals can be handled by programming the bitmapped terminal to emulate an ordinary alphanumeric terminal. This prevents *curses* from using the bitmap capabilities, but *curses* was not designed for bitmapping.

curses can also deal with terminals that have the "magic cookie" glitch in their display highlighting behavior. The term "magic cookie" means that changes in highlighting are controlled by storing a "magic cookie" character in a location on the screen. While this "cookie" takes up a space, preventing an exact implementation of what the programmer wanted, *curses* takes the extra character space into account, and moves part of the line to the right when necessary. In some cases, this unavoidably results in losing text along the right-hand edge of the screen, but *curses* compensates where possible by omitting extra spaces.

Multiple Terminals

Some applications require that text be displayed on more than one terminal at the same time from the same process. This is easily accomplished, even when the terminals are different types.

curses maintains all information about the current terminal in a global variable called

```
struct screen *SP;
```

Although the screen structure is hidden from the user, the C compiler accepts declarations of variables that are pointers. The user program should declare one screen pointer variable for each terminal that is to be handled. The routine:

```
struct screen *
newterm(type,fdout,fdin)
```

sets up a new terminal of the specified `type` and output is handled through file descriptor `fdout`. This is comparable to the usual program call to *initscr* which is essentially equivalent to

```
newterm(getenv(''TERM''),stdout)
```

A program that uses multiple terminals should call *newterm* for each terminal, and save the value returned as a reference to that terminal for other calls.

To change to a different terminal, call

```
set_term(term)
```

which returns the old value of variable `SP`. Do not assign to `SP` because certain other global variables must also be changed.

All *curses* routines always interact with the current terminal. *set_term* is used to change from one terminal to the next in a multi-terminal environment. When the program is ready to terminate, each terminal should be selected in turn by a call to *set_term*, then cleaned up with screen clearing and cursor locating routines, followed by a call to *endwin()* for that terminal. Repeat the sequence for each additional terminal used by the program. The example program **TWO** demonstrates the technique.

Video Highlighting

Video highlighting attributes can be displayed in any combination on terminals that support the various attribute capabilities. Each character position in screen data structures is allotted 16 bits: seven for the character code; the remaining nine for highlighting attributes, one bit per attribute. Each respective bit is associated with one of the following attributes: standout, underline, inverse video, blink, dim, bold, invisble, protect, and alternate character set. Standout selects the visually most pleasing highlighting method, and should be used by all programs that do not need a specific highlighting combination. Underlining, inverse video, blinking, dim, and bold are standard features on most popular terminals, though they are not usually all present on a single terminal (for example, no current terminal implements both bold and dim). Invisible means that visible characters are displayed as blanks for security reasons (such as when echoing passwords). Protected and Alternate Character Set are subject to the characteristics of the terminal being used. Invisible, protected, and alternate character set attributes are subject to change or substitution by *curses*, and should be avoided unless necessary.

When characters are stored, each character is combined with the **current attributes** variable associated with the window. The variable is formed by using one of the following routines:

attrset(attrs)	*wattrset(win,attrs)*
attron(attrs)	*wattron(win,attrs)*
attroff(attrs)	*wattroff(win,attrs)*
standout()	*wstandout(win)*
standend()	*wstandend(win)*

The following attributes can be specified in the `attrs` argument for corresponding attribute set/on/off routines.

`A_STANDOUT`	`A_BLINK`	`A_INVIS`
`A_UNDERLINE`	`A_DIM`	`A_PROTECT`
`A_REVERSE`	`A_BOLD`	`A_ALTCHARSET`

When specifying multiple attributes, they should be separated by the C logical OR operator (|). Thus, to specify blinking underline and disable all other attributes on the *stdscr* window, use `attrset(A_BLINK|A_UNDERLINE)`.

curses forms the current attributes word as follows:

- Each attribute (such as `A_UNDERLINE`) is stored as a 16-bit word where all bits are zero except the bit that represents the corresponding attribute in a stored character word (for example, 0000010000000000 controls blinking).

- All attributes forming the **attrs** argument are combined using the logical OR operator to create a single 16-bit word containing all attributes in the argument. For example, the three attribute words

  0000010000000000,
  0001000000000000, and
  0000001000000000 are combined to form
  0001011000000000 which identifies the new attributes.

- Three things can be done with the new attributes word. It can be used as the new current attributes (*attrset* or *wattrset*); or the new attributes can be added to any currently active attributes (*attron* or *wattron*), or deleted from the currently active attributes (*attroff* or *wattroff*).

- If *attrset* (or *wattrset*) was called, the routine stores the new attributes in the current attributes variable and returns. The previous set of current attributes is destroyed.

- If *attron* (or *wattron*) was called, the routine performs a logical OR of the current attributes with the new attributes, then places the result in the current attributes variable and returns. The revised current attributes variable contains all previously active attributes plus the new attributes.

- If *attroff* (or *wattroff*) was called, the routine inverts the new attributes, performs a logical AND on the inverted new attributes and the current attributes, then places the result in the current attributes variable and returns. The altered current attributes variable contains all previously active attributes except those specified in the call, which are now disabled.

- *standout* and *wstandout* obtain their highlighting attributes from the *terminfo* data base, then perform the same operation as *attron* prior to returning.

- *standend* and *wstandend* disable all attributes then return. They are equivalent to *attrset(0)* and *attrset(A_NORMAL)*.

- *attrset(0)* and *wattrset(win,0)* set the 16-bit current attributes variable value to zero which disables all attributes. **A_NORMAL** can be substituted for zero as an argument.

The preceding scenarios assume that the specified attributes are available on the current terminal. In every case, the *terminfo* data base is used to determine whether the selected attribute is present. If it is not, *curses* attempts to find a suitable substitute before forming the new attribute set. If the terminal has no highlighting capabilities, all highlighting commands are ignored.

Three other constants (defined in `<curses.h>`), in addition to the previously listed attributes are also available for program use if needed:

- `A_NORMAL` has the octal value 0000000, and can be used as an attribute argument for *attrset* to restore normal text display. *attrset(0)* is easier to type, but less descriptive. Both are equivalent.
- `A_ATTRIBUTES` has the octal value 0177600. It can be logically ANDed with a character data word to isolate the attribute bits and discard the character.
- `A_CHARTEXT` has the octal value 0000177. It can be logically ANDed with a character data word to isolate the character code and discard the attributes.

Special Keys

Most terminals have special keys, such as arrow keys, screen/line clearing keys, insert and delete line or character keys, and keys for user functions. The character sequences that such keys generate and send to the host computer vary from terminal to terminal. *curses* provides a convenient means for handling such keys through the use of *keypad* routines. Keypad capabilities are enabled by the call:

```
keypad(stdscr,TRUE)
```

during program initialization, or

```
keypad(win,TRUE)
```

when setting up and initializing other windows, as appropriate. When keypad is enabled, keypad character sequences are passed to the program by *getch*, but they are converted to special character values starting at 0401 octal (keypad character codes are listed in the keypad discussion early in this tutorial). Keypad codes are 16-bit values, and must not be stored in a *char* type variable because the upper bits must be preserved.

When keypad keys are used in a program, avoid using the escape key for program control because most keypad sequences begin with escape. If escape is used for program control, an ambiguity results that is not easily dealt with, and, at best, results in sluggish program response to all escape sequences as well as significant potential for incorrect program operation.

Scrolling Regions

Each window has a programmer-accessible scrolling region that is normally set to include the entire window. *curses* contains a routine that can be used to change the scrolling region to any location in the window by specifying the top and bottom margin lines. The routines are called by

```
setscrreg(top,bottom)
```

for the *stdscr* window, or

```
wsetscrreg(win,top,bottom)
```

for other windows. When the cursor advances beyond the bottom line in the region, all lines in the region are moved up one line (destroying the top line in the process) and a new line at the bottom of the region becomes the new cursor line. If scrolling has been enabled by a call to *scrollok* for that window, scrolling takes place, but only within the window boundary (if *scrollok* is not enabled, the cursor stays on the bottom line and no scrolling can occur). The scrolling region is a software feature only, and only causes a given window data structure to scroll. It may or may not translate to use of the hardware scrolling region that is featured on some terminals or hardware insert/delete line capabilities on the terminal.

Mini-Curses

All calls to *refresh* copy the current window to an internal screen image (*stdscr*). For simpler applications where window capabilities are not important and all operations can be handled by the standard screen, the screen output optimization capabilities of *curses* can be obtained through the low-level *curses* interface routines supported by *mini-curses*. Mini-*curses* is a subset of full *curses*, so any program that runs on the subset can also run on full *curses* without modification.

A complete list of commands is shown at the beginning of the *curses* commands section in this tutorial. Commands that are supported by *mini-curses* are marked with an asterisk (some that are not marked may also be accessible – if a program calls routines that are not, an error message showing undefined calls is produced by the compiler at compile time).

mini-curses routines are limited to commands that deal with the *stdscr* window. Certain other high-level functions that are convenient but not essential (such as *scanw*, *printw*, and *getch*) are not available, as well as all commands that begin with `w`. Low-level routines such as hardware insert/delete line and video attributes are supported, as are mode-setting routines such as *noecho*.

To access *mini-curses*, add `-DMINICURSES` to the CFLAGS in the makefile. If any routines are requested that are not available in *mini-curses*, an error diagnostic such as

```
Undefined:
m_getch
m_waddch
```

is listed to indicate that the program contains calls (in this case to *getch* and *waddch*) that cannot be linked because they are not available.

Remember that the preprocessor is involved in the implementation of *mini-curses*, so any programs that are compiled for use with *mini-curses* must be recompiled if they are to be used with full *curses*.

TTY Mode Functions

In addition to the save/restore functions *savetty()* and *resetty()*, other standard routines are provided by *curses* for entering and exiting normal tty mode.

- *resetterm()* restores the terminal to its state prior to *curses*' start-up.
- *fixterm* performs the equivalent of an *undo* on the previous *fixterm* on that terminal; it restores the "current curses mode" using the results of the most recent call to *saveterm()*.
- *endwin* automatically calls *resetterm*.
- Routines that handle control-Z (on systems that have process control) also use *resetterm()* and *fixterm()*.

Programs that use *curses* should use these routines before and after shell escapes, and also if the program has its own routines for dealing with control-Z. These routines are also available at the *terminfo* level.

Typeahead Check

When a user types something during a screen update, the update stops, pending a future update. This is useful when several keys are pressed in sequence, each of which produces a large amount of output. For example in a screen editor, the "forward screen" (or "next page") key draws the next screenful of text. If the key is pressed several times in rapid succession, rather than drawing several screens of text, *curses* cuts the updates short and only displays the last requested full screen. This feature is automatic, and cannot be disabled. It requires support by certain routines in the HP-UX operating system.

getstr

No matter whether echo is enabled or disabled, strings typed and input by *getstr* are echoed at the current cursor location. Erase and kill characters assigned by the user for his (or her) terminal are considered when handling input strings. Thus it is unnecessary for interactive programs to deal directly with erase, echo, and kill when processing a line of text from the terminal keyboard.

longname

The longname function does not require any arguments. It returns a pointer to a static storage area that contains the actual long (verbose) terminal name.

Nodelay Mode

The program call

```
nodelay(stdscr,TRUE)
```

puts the terminal in "no delay" mode. When nodelay is active, any call to getch returns the value -1 if there is nothing available for immediate input. This feature is helpful for real-time situations where a user is watching terminal screen outputs and presses a key when he wants to respond. For example, a program can be producing a text pattern on the screen while maintaining an open opportunity for the user to press certain keys to alter the output pattern, change cursor direction, or produce some other effect.

Example Programs

SCATTER

This program takes the first 23 lines from the standard input, then displays them in random order on the display terminal screen.

```
#include <curses.h>
#define    MAXLINES     120
#define    MAXCOLS      160
char    s[MAXLINES][MAXCOLS];    /* Screen Array */

main()
{
        register int  row = 0,
                      col = 0;
        register char c;
        int           char_count = 0; /* count non-blank characters */
        long          t;
        char          buf[BUFSIZ];

        initscr();
        for (row = 0; row < MAXLINES; row++)     /* initialize screen array */
                for (col = 0; col < MAXCOLS;  col++)
                        s[row][col] = ' ';

        row = 0;
        col = 0;
        /* Read screen in */
        while ( (c = getchar()) != EOF && row < LINES) {
                if (c != '\n' && col < COLS) {
                        /* 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((int)(t&0177777L));

        while (char_count) {
                row = rand() % LINES;
                col = (rand() >> 2) % COLS;
                if (s[row][col] != ' ' && s[row][col] != EOF) {
                        move(row,col);
                        addch(s[row][col]);
                        s[row][col] = EOF;
                        char_count--;
                        refresh();
                }
        }

        endwin();
        exit(0);
}
```

SHOW

This example program displays a file taken from the standard input, one screen at a time. Press the terminal space bar to advance to the next screen.

```
#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();                             /* enable more screen optimization */
        idlok(stdscr,TRUE);         /* allow insert/delete line */

        while (1) {
                move(0,0);
                for (line = 0; line < LINES; line++) {
                        if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
                                clrtobot();
                                done();
                        }
                        move(line,0);
                        printw("%s", linebuf);
                }

                refresh();
                if (getch() == 'q')
                        done();
        }
}
        void
done()
{
        move(LINES-1, 0);
        clrtoeol();
        refresh();
        endwin();
        exit(0);
}
```

HIGHLIGHT

This example program displays text taken from the standard input. Highlighting is determined by embedded character sequences in the file. \U starts underlining, \B starts bold highlighting, and \N restores normal display characteristics.

```
#include <curses.h>

main(argc,argv)
        char    **argv;
{
        FILE    *fd;
        int     c,c2;

        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);

	for (;;) {
		c = getc(fd);
		if (c == EOF)
			break;

		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);
}
```

WINDOW

This program demonstrates the use of multiple windows.

```
#include <curses.h>

WINDOW    *cmdwin;

main()
{
        int     i,c;
        char    buf[120];

        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);         /* clear window */
                        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.
                         */
                        erase();
                        for (i=0; i < LINES; i++)
                                mvprintw(i,0,"%s",buf);
                        refresh();
                        break;
                case 'q':
                        endwin();
```

```
                        exit(0);
                }
        }
}
```

TWO

This program shows how to handle two terminals from a single program.

```
#include <curses.h>
#include <signal.h>

struct screen *me, *you;
struct screen *set_term();

FILE    *fd, *fdyou;
char    linebuf[512];

main(argc,argv)
        char    **argv;
{
        int     done();
        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 his/her 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);
        noecho();
        cbreak();
        nonl();
        nodelay(stdscr,TRUE);

        /* Dump first screen full on my terminal */
        dump_page(me);
```

```
        /* Dump second screen full on his/her 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)
        struct 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();
                }
                mvprintw(line,0,"%s",linebuf);
        }

        standout();
        mvprintw(LINES-1,0,"--More--");
        standend();
        refresh();    /* sync screen */
}

/*
 * Clean up and exit.
 */
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 clean up */

        /* 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 clean up */

        exit(0);
}
```

TERMHL

This program is equivalent to the earlier example program **HIGHLIGHT**, but uses *terminfo* routines instead.

```
#include <curses.h>
#include <term.h>

int     ulmode = 0;     /* Currently underlining */

main(argc, argv)
        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(0,1,0);
        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);
}
```

EDITOR

This program is a very simple screen-oriented editor that is similar to a small subset of *vi*. For simplicity, the *stdscr* window is also used as the editing buffer.

```
#include <curses.h>
#define CTRL(c) ('c'&037)
main(argc,argv)
        char    **argv;
{
        int     i,n,l;
        int     c;
        FILE    *fd;

        if (argc != 2) {
                fprintf(stderr,"Usage: edit file\n");
                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)
                addch(c);
        fclose(fd);

        move(0,0);
        refresh();
        edit();

        /* Write out the file */
        fd = fopen(argv[1],"w");
        for (l=0; l < LINES; l++) {
                n = len(l);
                for (i=0; i<n; i++)
                        putc(mvinch(l,i),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();
                switch(c) {     /* Editor commands */

                /* hjkl and arrow keys: move cursor */
                /* in direction indicated */
                case    'h':
                case    KEY_LEFT:
                        if (col > 0)
                                col--;
                        break;

                case    'j':
                case    KEY_DOWN:
                        if (row < LINES-1)
                                row++;
                        break;

                case    'k':
                case    KEY_UP:
                        if (row > 0)
                                row--;
                        break;

                case    'l':
                case    KEY_RIGHT:
                        if (col < COLS-1)
```

```
                        col++;
                    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):
                    clearok(curscr);
                    refresh();
                    break;

                /* w: write and quit */
                case    'w':
                    return;

                /* q: quit without writing */
                case    'q':
                    endwin();
                    exit(1);
                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();
}
```

Index

a

addch 2,4,10,29,36
addstr 29,37
alternate character set 10
arrow keys 1,62
attributes 10,11
attroff 11,32,37,61
attron 11,37,61
attrset 2,10,11,37,61

b

baudrate 33,38
beep 22,33,38
blinking highlight 10
bold highlight 10
box 30,38

c

cbreak 4,9,27,38
clear 30,38
clearok 4,25,38
cleartobot 38
cleartoeol 39
clrtobot 9,30
clrtoeol 9,30
COLS 5
configuration routines 27
current attributes 11
current screen 2
current terminal 17
curscr 2

curses ... 1
curses routines ... 34,35
curses.h ... 10

d

data output routines ... 29
delay functions ... 33
delay_output ... 39,55
delch ... 30,39
deleteing text ... 30
deleteln ... 30,39
delwin ... 39
dim highlight ... 10
doupdate ... 29,39
draino ... 33,40

e

echo ... 27,40
endwin ... 5,25,40,64
erase ... 30,40
erasechar ... 33,40
ERR ... 24
escape sequences ... 22
example programs:
- editor ... 21,75
- highlight ... 12,68
- scatter ... 66
- show ... 12,67
- termhl ... 20,73
- two ... 18,59,71
- window ... 15,70

f

fixterm 40,64
flash 22,33,41
flush 4
flushinp 33,41

g

getch 6,31,41
getstr 31,42,65
gettmode 42
getyx 31,43

h

half-bright highlight 10
has_ic 43
has_il 43
highlight escape sequences 12
highlighting 2
highlights 10,32,60

i

idiok 25
idlok 4,9,43
inch 31,43
include files 24
initialization routines 25
initscr 4,25,43
input routines 31
insch 30,43
inserting text 30
insertln 30,44
intrflush 26,44
inverse video 10
invisible highlight 10

k

keyboard input 6
keypad 6,7,25,44,62
keypad codes 8
killchar 33,44

l

leaveok 26,44
LINES 5
loader options 24
longname 25,45,65
low-level access 19

m

magic cookie 58
manipulation routines 28
meta 26,45
mini-curses 24,63,64
move 29,45
multiple terminals 17,59
mvaddch 45
mvaddstr 45
mvcur 45
mvdelch 45
mvgetch 46
mvgetstr 46
mvinch 46
mvinsch 46
mvprintw 46
mvscanw 46
mvwaddch 46
mvwaddstr 46
mvwdelch 46
mvwgetch 46
mvwgetstr 46
mvwin 46

mvwinch 47
mvwinsch 47
mvwprintw 47
mvwscanw 47

n

napms 33,47
newpad 47
newterm 17,25,47,59
newwin 14,47
nl 27,48
nocbreak 48
nodelay 26,48
nodelay mode 65
noecho 9,27,48
non print highlight 10
nonl 9,27,48
noraw 27,48

o

OK 24
options 25
overlay 14,28,48
overwrite 14,28,48

p

padding 2
pads 13
pnoutrefresh 29,48
portability functions 33
prefresh 29,48
printw 4,30,49
putp 55

r

race conditions 17
raw 27,49
refresh 4,12,22,29,49
resetterm 49,64
resetty 27,49

s

saveterm 50
savetty 27,50
scanw 31,50
screen size 5
scroll 50
scrollok 26,50
scrollw 30
setscrreg 26,50,63
setterm 51
setupterm 19,25,51,55
set_term 17,18,51
standard screen 2
standend 32,51
standout 32,51,61
standout highlight 10
stdscr 2
struct screen 59
sttron 32
sttrset 32
subwin 51
Subwindows 16

t

TERM 1
termcap routines 57
terminfo 1
terminfo-level access 19
touchwin 15,28,52
tparm 56
tputs 20,56
traceoff 52
traceon 52
tty mode 64
typeahead 26,52,64

u

unctrl 52
underlining highlight 10

v

vidattr 20,56
vidputs 57

w

waddch 14,52
waddchr 10
wattroff 32,61
wattron 32,61
wattrset 61
wclear 30
wdeleteln 30
window 2
windows 13

Windows:
Creating 14
Multiple 13
Subwindows 16
wmove 29
wnoutrefresh 29
wrefresh 14,29
writing routines 29
wsetscrreg 63
wstandout 61

Manual Comment Sheet Instruction

If you have any comments or questions regarding this manual, write them on the enclosed comment sheets and place them in the mail. Include page numbers with your comments wherever possible.

If there is a revision number, (found on the Printing History page), include it on the comment sheet. Also include a return address so that we can respond as soon as possible.

The sheets are designed to be folded into thirds along the dotted lines and taped closed. Do not use staples.

Thank you for your time and interest.

Reorder Number
97089-90030
Printed in U.S.A. 4/85

97089-90601
Mfg. No. Only