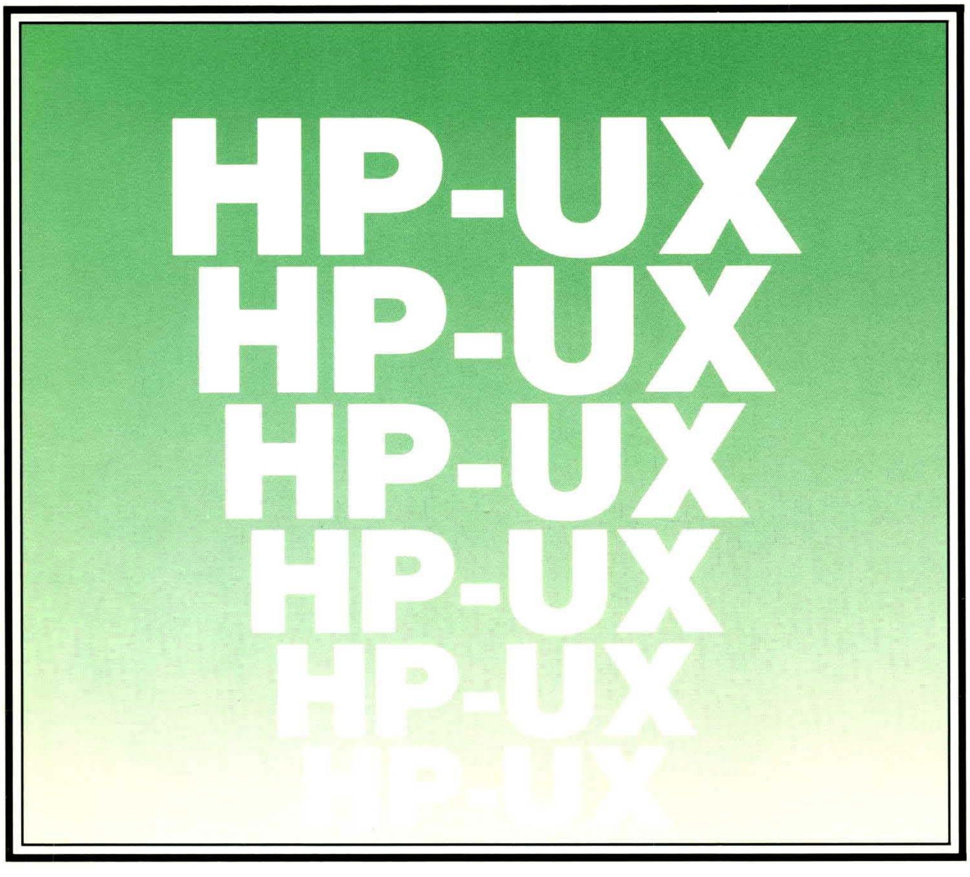
HEWLETT PACKARD

HP-UX Reference
Vol. 3: Sections 3, 4, 5, 7, and 9



HP-UX Reference
Vol. 3: Sections 4, 5, 7, and 9

for

HP Part Number 09000-90008

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 1985...Edition 1. This manual replaces HP-UX Reference Manual 09000-90007 and documents HP-UX Release 5.0 for Series 200, 300 and 500.

November 1985...Edition 2. Updated from Edition 1 to reflect Series 200/300 HP-UX Release 5.1 changes. Several omitted pages in Edition 1 were also added.

June 1986...Edition 3. Update 1 incorporated.

September 1986...Edition 3 Update 1. This update reflects additions and changes incorporated in Series 500 HP-UX Release 5.1. Added command *autobackup*(1M) and core files support (*core*(5)), changed blocksize limitations for SDF file formats, and fixed various bugs.

TABLE OF CONTENTS

Table of Contents

Table of Contents

NAME

intro - introduction to subroutines and libraries

SYNOPSIS

#include <stdio.h>

#include <math.h>

HP-UX COMPATIBILITY

Level:	The level given is the level for which the library is available, not the level at which the linkable object code appears. The supporting host will contain appropriate libraries for HP-UX/RUN ONLY and HP-UX/NUCLEUS systems.
Origin:	System III, System V, UCB

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke HP-UX system primitives, which are described in Section 2 of this volume. Certain major collections are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). The link editor *ld*(1) searches this library under the **-lc** option. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

(3M) These functions constitute the Math Library, *libm.* They are automatically loaded as needed by the FORTRAN compiler *f77*(1). They are not automatically loaded by the C compiler, *cc*(1); however, the link editor searches this library under the **-lm** option. Declarations for these functions may be obtained from the **#include** file **<math.h>**. Several generally useful mathematical constants are also defined there (see *math*(5)).

(3N) These functions constitute the networking library, *libn.* The link editor searches this library under the **–ln** option. Declarations for these functions can be obtained from the **#include** file **<stdio.h>**.

(3S) These functions constitute the "standard I/O package" (see *stdio*(3S)). These functions are in the library *libc*, already mentioned. Declarations for these functions may be obtained from the **#include** file **<stdio.h>**.

(3X) Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

DEFINITIONS

A *character* is any bit pattern able to fit into a byte on the machine. The *null character* is a character with value 0, represented in the C language as '\0'. A *character array* is a sequence of characters. A *null-terminated character array* is a sequence of characters, the last of which is the *null character*. A *string* is a designation for a *null-terminated character array*. The *null string* is a character array containing only the null character. A **NULL** pointer is the value that is obtained by casting **0** into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. **NULL** is defined as **0** in **<stdio.h>**; the user can include an appropriate definition if not using **<stdio.h>**.

FILES

/lib/libc.a
/lib/libm.a
/lib/libn.a

SEE ALSO

intro(2), stdio(3S), math(5).
ar(1), cc(1), f77(1), ld(1), lint(1), nm(1), ranlib(1), intro(2), stdio(3S).

DIAGNOSTICS

Functions in the C and Math Libraries (3C and 3M) may return the conventional values **0** or

±**HUGE** (the largest–magnitude single–precision floating–point numbers; **HUGE** is defined in the <*math.h*> header file) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* (see *errno*(2)) is set to the value EDOM or ERANGE.

WARNING

Many of the functions in the libraries call and/or refer to other functions and external variables described in this section and in section 2 (*System Calls*). If a program inadvertantly defines a function or external variable with the same name, the presumed library version of the function or external variable may not be loaded. The *lint*(1) program checker reports name conflicts of this kind as "multiple declarations" of the names in question. Definitions for sections 2, 3C, and 3S are checked automatically. Other definitions can be included by using the **-l** option (for example, **-l***m* includes definitions for the Math Library, section 3M). Use of *lint* is highly recommended.

NAME

a64l, l64a - convert between long integer and base–64 ASCII string

SYNOPSIS

long a64l (s)
char *s;

char *l64a (l)
long l;

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

DESCRIPTION

These functions are used to maintain numbers stored in *base–64* ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix–64 notation.

The characters used to represent "digits" are **.** for 0, **/** for 1, **0** through **9** for 2-11, **A** through **Z** for 12-37, and **a** through **z** for 38-63.

The leftmost character is the least significant digit. For example,

$$\text{a0} = (38 \text{ x } 64^0) + (2 \text{ x } 64^1) = 166$$

A64l takes a pointer to a null–terminated base–64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l* will use the first six.

L64a takes a **long** argument and returns a pointer to the corresponding base–64 representation. If the argument is 0, *l64a* returns a pointer to a null string.

BUGS

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

NAME

abort - generate an IOT fault

SYNOPSIS

int abort ( )

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Abort first closes all open files if possible, then causes the SIGIOT signal to be sent to the process. This usually results in termination with a core dump.

It is possible for *abort* to return control if **SIGIOT** is caught or ignored, in which case the value returned is that of the *kill*(2) system call.

SEE ALSO

adb(1), exit(2), kill(2), signal(2).

DIAGNOSTICS

If **SIGIOT** is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message "abort - core dumped" is written by the shell.

NAME

abs - return integer absolute value

SYNOPSIS

int abs (i)
int i;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Abs returns the absolute value of its integer operand.

HARDWARE DEPENDENCIES

Series 200/500 and Integral Personal Computer:

The largest negative integer recognized by the system returns itself.

SEE ALSO

floor(3M).

NAME

assert - verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
int expression;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), *assert* prints

"Assertion failed: *expression*, file *xyz*, line *nnn*"

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the *assert* statement.

Compiling with the preprocessor option **-DNDEBUG** (see *cpp*(1)), or with the preprocessor control statement "**#define** NDEBUG" ahead of the "**#include** <assert.h>" statement, will stop assertions from being compiled into the program.

SEE ALSO

cpp(1), abort(3C).

NAME

atof, atoi, atol – convert ASCII to numbers

SYNOPSIS

double atof (nptr)
char *nptr;

int atoi (nptr)
char *nptr;

long atol (nptr)
char *nptr;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System III

DESCRIPTION

These functions convert a string pointed to by *nptr* to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

Atof recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optionally signed integer.

Atoi and *atol* recognize an optional string of tabs and spaces, then an optional sign, then a string of digits.

HARDWARE DEPENDENCIES

Series 200/500:
Atoi and *atol* are identical.

SEE ALSO

scanf(3S).

BUGS

There are no provisions for overflow.

NAME

j0, j1, jn, y0, y1, yn - Bessel functions

SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

J0 and *j1* return Bessel functions of x of the first kind of orders 0 and 1 respectively. *Jn* returns the Bessel function of x of the first kind of order n.

Y0 and *y1* return the Bessel functions of x of the second kind of orders 0 and 1 respectively. *Yn* returns the Bessel function of x of the second kind of order n. The value of x must be positive.

DIAGNOSTICS

Non-positive arguments cause *y0*, *y1* and *yn* to return the value **-HUGE** and to set *errno* to **EDOM**. They also cause a message indicating DOMAIN error to be printed on the standard error output; the process will continue.

Arguments too large in magnitude cause *j0*, *j1*, *y0* and *y1* to return zero and to set *errno* to **ERANGE**. In addition, a message indicating TLOSS error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

bsearch - binary search a sorted table

SYNOPSIS

char *bsearch ((char *) key, (char *) base, nel, sizeof (*key), compar)
unsigned nel;
int (*compar)( );

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: System V

DESCRIPTION

Bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *Key* points to a datum instance to be sought in the table. *Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero as accordingly the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

```
#include <stdio.h>

#define TABSIZE      1000

struct node {                   /* these are stored in the table */
        char *string;
        int length;
};
struct node table[TABSIZE];     /* table to be searched */
        .
        .
        .
{
        struct node *node_ptr, node;
        int node_compare( );  /* routine to compare 2 nodes */
        char str_space[20];   /* space to read string into */
        .
        .
        .
        node.string = str_space;
        while (scanf("%s", node.string) != EOF) {
                node_ptr = (struct node *)bsearch((char *)(&node),
                           (char *)table, TABSIZE,
                           sizeof(struct node), node_compare);
                if (node_ptr != NULL) {
                        (void)printf("string = %20s, length = %d\n",
                                node_ptr->string, node_ptr->length);
                } else {
                        (void)printf("not found: %s\n", node.string);
                    }
                }
        }
        /*
                This routine compares two nodes based on an
                alphabetical ordering of the string field.
        */
        int
        node_compare(node1, node2)
        struct node *node1, *node2;
        {
                return strcmp(node1->string, node2->string);
        }
```

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

hsearch(3C), lsearch(3C), qsort(3C), tsearch(3C).

DIAGNOSTICS

A NULL pointer is returned if the key cannot be found in the table.

BUGS

A random entry is returned if more than one entry matches the selection criteria.

NAME

catread - MPE/RTE-style message catalog support

SYNOPSIS

```
int catread (fd, set_num, msg_num, msg_buf, buflen [,arg]...)
int fd, set_num, msg_num, buflen;
char *msg_buf, *arg;
```

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

Native Language Support:
8-bit data, customs, messages

DESCRIPTION

Catread is layered on *getmsg(3C)* for supporting message catalog applications from MPE/RTE. Refer to the external specifications for message catalogs on these systems for use of this routine.

The message read from the catalog may have embedded formatting information in the form ![*digit*]. An exclamation mark followed by a digit *n* is replaced by the *n*th argument string. If exclamation marks are not numbered, they are replaced by the arguments in serial order. Either all or none must be numbered.

If successful, returns the number of non-null bytes placed in the buffer.

DIAGNOSTICS

Catread returns a negative integer if *set_num* or *msg_num* are not found in the catalog.

SEE ALSO

gencat(1), getmsg(3C), hpnls(7).

NAME

clock - report CPU time used

SYNOPSIS

long clock ( )

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Clock returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2) or *system*(3S).

The resolution of the clock varies depending on the hardware and software configuration. See HARDWARE DEPENDENCIES for the various vales.

SEE ALSO

times(2), wait(2), system(3S).

HARDWARE DEPENDENCIES

Series 200:	The clock resolution is 20 milliseconds.
Series 500:	The clock resolution is 10 milliseconds as a default.

BUGS

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes).

NAME

toupper, tolower, _toupper, _tolower, toascii - translate characters

SYNOPSIS

#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Toupper and *tolower* have as domain the range of *getc*(3S): the integers from -1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

The macros *_toupper* and *_tolower* accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. The macro *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results. Use of this form will never work with foreign character sets.

Toascii yields its argument with all bits turned off that are not part of a standard 7 bit ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ascii(7), ctype(3C), getc(3S), nl_conv(3C).

NAME

crypt – generate password encryption

SYNOPSIS

char *crypt (key, salt)
char *key, *salt;

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

DESCRIPTION

Crypt is the password encryption function. It is based on the NBS Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

Key is a user's typed password. *Salt* is a two–character string chosen from the set [**a–zA–Z0–9./**]; this *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

SEE ALSO

login(1), passwd(1), getpass(3C), passwd(5)

BUGS

The return value points to static data that is overwritten by the next call to *crypt*(3C).

NAME

ctermid - generate file name for terminal

SYNOPSIS

#include <stdio.h>
char *ctermid (s)
char *s;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Ctermid generates the path name of the controlling terminal for the current process, and stores it in a string.

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to *ctermid*, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least **L_ctermid** elements; the path name is placed in this array and the value of *s* is returned. The constant **L_ctermid** is defined in the *<stdio.h>* header file.

NOTES

The difference between *ctermid* and *ttyname*(3C) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a string (**/dev/tty**) that will refer to the terminal if used as a file name. Thus *ttyname* is useful only if the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C).

NAME

ctime, nl_ctime, daylight, localtime, gmtime, asctime, nl_asctime, timezone, tzset, tzname - convert date and time to string

SYNOPSIS

#include <time.h>

char *ctime (clock)
long *clock;

char *nl_ctime (clock, format, langid)
long *clock; char *format; int langid;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

char *nl_asctime (tm, format, langid)
struct tm *tm; char *format; int langid;

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset ( )

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: System V

Native Language Support:
8-bit data, customs, messages

DESCRIPTION

Ctime converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970, and returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Sun Sep 16 01:03:52 1973\n\0
```

NL_ctime 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 the 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.

Localtime and *gmtime* return pointers to "tm" structures, described below. *Localtime* corrects for the time zone and any summer time zone corrections (Daylight Savings time in the US), according to the **TZ** string in the user's environment. *Gmtime* converts directly to Greenwich Mean Time (GMT), which is the time the HP-UX System uses.

Asctime converts a "tm" structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

NL_asctime, like *nl_ctime*, allows the date string to be formatted, and month and weekday names to be in the user's native language. However, like *asctime* , it takes "tm" as its argument.

Declarations of all the functions and externals, and the "tm" structure, are in the <*time.h*> header file. The structure declaration is:

```
struct tm {
        int tm_sec;         /* seconds (0 - 59) */
        int tm_min;         /* minutes (0 - 59) */
        int tm_hour;        /* hours (0 - 23) */
        int tm_mday;        /* day of month (1 - 31) */
        int tm_mon;         /* month of year (0 - 11) */
        int tm_year;        /* year - 1900 */
        int tm_wday;        /* day of week (Sunday = 0) */
        int tm_yday;        /* day of year (0 - 365) */
        int tm_isdst;
};
```

Tm_isdst is non–zero if a summer time zone correction such as Daylight Savings time is in effect.

The external **long** variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, *timezone* is 5*60*60); the external variable *daylight* is non–zero if and only if you have specified a summer time zone correction in your **TZ** environment variable. The values of the external variables *timezone*, *daylight*, and *tzname* are set from the environment variable **TZ** by the function *tzset*, which may be called directly, or indirectly through the functions *localtime, ctime,* or *nl_ctime.* **TZ** is set by default when the user logs on, to a value in the local /etc/profile file (see *profile*(5)).

HARDWARE DEPENDENCIES

Series 200/500:

Tztab(5) is not currently supported.

SEE ALSO

time(2), getenv(3C), langinfo(3C), profile(5), environ(7), hpnls(7).

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii - classify characters

SYNOPSIS

#include <ctype.h>

int isalpha (c)
int c;

. . .

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *Isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value **EOF** (see *stdio*(3S)).

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) through 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except false for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).
isascii	*c* is an ASCII character, code less than 0200.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

SEE ALSO

nl_ctype(3C), stdio(3S), ascii(7).

NAME

curses - CRT screen handling and optimization package

SYNOPSIS

#include <curses.h>
cc [flags] files **-lcurses** [libraries]

HP-UX COMPATIBILITY

Level:	HP-UX/NUCLEUS
Origin:	System V

DESCRIPTION

These routines provide a means for updating screens with reasonable optimization. To ensure proper initialization, the routine *initscr()* must be called before any other routines that deal with windows and screens are used. The *endwin()* routine should be called before exiting to restore conditions as they existed prior to program entry. Character–at–a–time input without echoing (used in most interactive, screen oriented–programs), is obtained by calling *"nonl(); cbreak(); noecho();"* after calling *initscr()*.

The full–*curses* interface provides a means for manipulating window data structures. **Windows** can be thought of as two–dimensional arrays of characters representing all or part of a CRT screen. A default window called **stdscr** is supplied. Other windows can be created by using **newwin**. Windows are referred to by variables declared "WINDOW *"; the type WINDOW is defined in *curses.h* to be a C structure. These data structures are manipulated with functions described below. Two simple (and widely used) examples are **move** and **addch**. (More–general versions of these functions are provided. Their names begin with 'w', to signify that you can specify the window to be used. Routines not beginning with 'w' affect only **stdscr**.) After manipulation, *refresh()* is called to make the user's CRT screen look like **stdscr**.

Mini–Curses is a subset of *curses*. It only supports manipulation of the standard window. To invoke this subset, use –DMINICURSES as a **cc** option. This level is smaller and faster than full *curses*.

If the environment variable TERMINFO is defined, any program using *curses* checks for a local terminal definition before checking in the standard place. For example, if the standard place is **/users/lib/terminfo**, and TERM is set to "hp2623", the compiled file is normally found in **/users/lib/terminfo/h/hp2623** (the "h" is copied from the first letter of "hp2623" to avoid creation of huge directories). However, if TERMINFO is set to **/users/mark/myterms**, *curses* first checks **/users/mark/myterms/h/hp2623**, then, if that fails, checks **/usr/lib/terminfo/h/hp2623**. This is useful when developing experimental definitions and when write permission in **/users/lib/terminfo** is not available.

SEE ALSO

terminfo(5) and termcap(3).

FUNCTIONS

All routines listed here are fully accessible to full *curses*. Those marked with an asterisk are also available to *Mini–Curses*.

addch(ch)*	add a character to (like putchar) (wraps to next line at end of line)
addstr(str)*	calls addch with each character in *str*
attroff(attrs)*	turn off attributes named
attron(attrs)*	turn on attributes named
attrset(attrs)*	set current attributes to *attrs*
baud rate( )*	current terminal speed
beep( )*	sound beep on terminal
box(win, vert, hor)	draw a box around edges of *win* *vert* and *hor* are chars to use for vert.

	and hor. edges of box
cbreak()*	set cbreak mode
clear()*	clear *stdscr*
clearok(win, bf)	clear screen before next redraw of *win*
clrtobot()	clear to bottom of *stdscr*
clrtoeol()	clear to end of line on *stdscr*
delay_output(ms)*	insert ms millisecond pause in output
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete *win*
doupdate()	update screen from all wnooutrefresh
draino(ms)	drain output to ms milliseconds
echo()*	set echo mode
endwin()*	end window modes
erase()*	erase *stdscr*
erasechar()*	return user's erase character
fixterm()	restore tty to "in curses" state
flash()*	flash screen or beep
flushinp()*	throw away any typeahead
getch()	get a char from tty
getstr(str)	get a string through *stdscr*
gettmode()	dummy entry point. Does nothing.
getyx(win, y, x)	get (y, x) co-ordinates
has_ic()*	true if terminal can do insert character
has_il()*	true if terminal can do insert line
idlok(win, bf)*	use terminal's insert/delete line if bf != 0
inch()	get char at current (y, x) co-ordinates
initscr()*	initialize screens
insch(c)	insert a char
insertln()	insert a line
intrflush(win, bf)	interrupts flush output if bf is TRUE
keypad(win, bf)	enable keypad input
killchar()*	return current user's kill character
leaveok(win, flag)	OK to leave cursor anywhere after refresh if flag!=0 for *win*, otherwise cursor must be left at current position.
longname()	return verbose name of terminal
meta(win, flag)*	allow meta characters on input if flag != 0
move(y, x)*	move to (y, x) on *stdscr*
mvaddch(y, x, ch)*	move(y, x) then addch(ch)
mvaddstr(y, x, str)*	move(y, x) then addstr(str)
mvcur(oldrow, oldcol, newrow, newcol)	low-level cursor motion
mvdelch(y, x)	like delch, but move(y, x) first
mvgetch(y, x)	etc.
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, by, bx)	
mvwinch(win, y, x)	
mvwinsch(win, y, x, c)	
mvwprintw(win, y, x, fmt, args)	
mvwscanw(win, y, x, fmt, args)	
napms(ms)	suspend program for ms milliseconds
newpad(nlines, ncols)	create a new pad with given dimensons s
newterm(type, fpout, fpin)*	set up new terminal of given type to I/O on fpout/fpin.
newwin(lines, cols, begin_y, begin_x)	create a new window.
nl()*	set newline mapping
nocbreak()*	unset cbreak mode
nodelay(win, bf)	enable nodelay input mode through getch
noecho()*	unset echo mode
nonl()*	unset newline mapping
noraw()*	unset raw mode
overlay(win1, win2)	overlay win1 on win2
overwrite(win1, win2)	overwrite win1 on top of win2
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)	like prefresh but with no output until doupdate called
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)	refresh from pad starting with given upper left corner of pad with output to given portion of screen
printw(fmt, arg1, arg2, ...)	printf on *stdscr*
raw()*	set raw mode
refresh()*	make current screen look like *stdscr*
resetterm()*	set tty modes to "out of curses" state
resetty()*	reset tty flags to stored value
saveterm()*	save current modes as "in curses" state
savetty()*	store current tty flags
scanw(fmt, arg1, arg2, ...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win, flag)	allow terminal to scroll if flag != 0
set_term(new)*	set the current terminal to new
setscrreg(t, b)	set user scrolling region to lines t through b
setterm(type)	establish terminal with given type
setupterm(term, filenum, errret)	initialize specified terminal
standend()*	clear standout mode attribute
standout()*	set standout mode attribute
subwin(win, lines, cols, begin_y, begin_x)	create a subwindow
touchwin(win)	change all of *win*
traceoff()	dummy entry point. Does nothing
traceon()	dummy entry point. Does nothing
typeahead(fd)	use file descriptor fd to check typeahead
unctrl(ch)*	printable version of *ch*
waddch(win, ch)	add char to *win*
waddstr(win, str)	add string to *win*
wattroff(win, attrs)	turn off *attrs* in *win*
wattron(win, attrs)	turn on *attrs* in *win*
wattrset(win, attrs)	set attrs in *win* to *attrs*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
wdelch(win, c)	delete char from *win*
wdeleteln(win)	delete line from *win*

werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win, str)	get a string through *win*
winch(win)	get char at current (y, x) in *win*
winsch(win, c)	insert char into *win*
winsertln(win)	insert line into *win*
wmove(win, y, x)	set current (y, x) co-ordinates on *win*
wnoutrefresh(win)	refresh but no screen output
wprintw(win, fmt, arg1, arg2, ...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win, fmt, arg1, arg2, ...)	scanf through *win*
wsetscrreg(win, t, b)	set scrolling region of *win*
wstandend(win)	clear standout attribute in *win*
wstandout(win)	set standout attribute in *win*

TERMINFO LEVEL ROUTINES

These routines should be called by programs wishing to deal directly with the *terminfo* database. Due to the low level of this interface, it is discouraged. Initially, *setupterm* should be called. This will define the set of terminal dependent variables defined in *terminfo*(5). The include files **<curses.h>** and **<term.h>** should be included to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through *tparm* to instantiate them. All *terminfo* strings (including the output of *tparm*) should be printed with *tputs* or *putp* . Before exiting, *resetterm* should be called to restore the tty modes. (Programs desiring shell escapes or suspending with control–Z can call *resetterm* before the shell is called and *fixterm* after returning from the shell.)

fixterm( )	restore tty modes for terminfo use (called by setupterm)
resetterm( )	reset tty modes to state before program entry
setupterm(term, fd, rc)	read in database. Terminal type is the character string *term*, all output is to UNIX System file descriptor *fd*. A status value is returned in the integer pointed to by *rc*: 1 is normal. The simplest call would be *setupterm(0, 1, 0)* which uses all defaults.
tparm(str, p1, p2, ..., p9)	instantiate string str with parms p_i.
tputs(str, affcnt, putc)	apply padding info to string *str*. *affcnt* is the number of lines affected, or 1 if not applicable. *Putc* is a putchar–like function to which the characters are passed, one at a time.
putp(str)	handy function that calls tputs (str, 1, putchar).
vidputs(attrs, putc)	output the string to put terminal in video attribute mode *attrs*, which is any combination of the attributes listed below. Chars are passed to putchar–like function *putc*.
vidattr(attrs)	Like vidputs but outputs through putchar.

TERMCAP COMPATIBILITY ROUTINES

These routines were included as a conversion aid for programs that use *termcap*. Their parameters are the same as for *termcap*, but they are emulated using the *terminfo* database. They may be removed at a later date.

tgetent(bp, name)	look up termcap entry for name
tgetflag(id)	get boolean entry for id
tgetnum(id)	get numeric entry for id
tgetstr(id, area)	get string entry for id
tgoto(cap, col, row)	apply parms to given cap
tputs(cap, affcnt, fn)	apply padding to cap calling fn as putchar

ATTRIBUTES

The following video attributes can be passed to the functions *attron*,*attroff*,*attrset*.

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_BLANK	Blanking (invisible)
A_PROTECT	Protected
A_ALTCHARSET	Alternate character set

FUNCTION KEYS

The following function keys are returned by *getch* if *keypad* has been enabled and the function is supported. Note that some of these may not be currently supported due to lack of definitions in *terminfo,* or because the terminal does not transmit a unique code when the key is pressed.

Name	*Value*	*Key name*
KEY_BREAK	0401	break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 is reserved.
KEY_F(n)	(KEY_F0+(n))	Formula for fn.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	soft (partial) reset (unreliable)
KEY_RESET	0531	reset or hard reset (unreliable)
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left)

WARNING

The plotting library *plot*(3X) and the *curses* library *curses*(3X) both use the names *erase*( ) and *move*( ). The *curses* versions are macros. If you need both libraries, put the *plot*(3X) code in a different source file than the *curses*(3X) code, and/or **#undef** *move*( ) and *erase*( ) in the *plot*(3X) code.

NAME

cuserid - get character login name of the user

SYNOPSIS

#include <stdio.h>

char *cuserid (s)
char *s;

HP-UX COMPATIBILITY

Level:	HP-UX/NUCLEUS
Origin:	System V

DESCRIPTION

Cuserid generates a character-string representation of the login name that the owner of the current process is logged in under. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The constant **L_cuserid** is defined in the **<stdio.h>** header file.

DIAGNOSTICS

If the login name cannot be found, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character **(\0)** will be placed at *s[0]*.

BUGS

Cuserid uses *getpwnam*(3C); thus the results of a user's call to the latter will be obliterated by a subsequent call to the former.

SEE ALSO

getlogin(3C), getpwent(3C).

NAME

dial - establish an out–going terminal line connection

SYNOPSIS

#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
int fd;

HP–UX COMPATIBILITY

Level: HP–UX/STANDARD

Origin: System V

Dial returns a file–descriptor for a terminal line open for read/write. The argument to *dial* is a CALL structure (defined in the <*dial.h*> header file).

When finished with the terminal line, the calling program must invoke *undial* to release the semaphore that has been set during the allocation of the terminal device.

The definition of CALL in the <*dial.h*> header file is:

```
typedef struct {
        struct termio  *attr;          /* pointer to termio attribute struct */
        int            baud;           /* transmission data rate */
        int            speed;          /* 212A modem: low=300, high=1200 */
        char           *line;          /* device name for out–going line */
        char           *telno;         /* pointer to tel–no digits string */
        int            modem;          /* specify modem control for direct lines */
        char           *device;         /*Will hold the name of the device usedd
                                       to make a connection */
        int            dev_len;        /* The length of the device used to
                                       make connection */
} CALL;
```

The CALL element *speed* is intended only for use with an outgoing dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high– or low–speed setting on the 212A modem. Note that the 113A modem or the low–speed setting of the 212A modem will transmit at any rate between 0 and 300 bits per second. However, the high–speed setting of the 212A modem transmits and receivers at 1200 bits per second only. The CALL element *baud* is for the desired transmission baud rate. For example, one might set *baud* to 110 and *speed* to 300 (or 1200). However, if **speed** set to 1200 **baud** must be set to high (1200).

If the desired terminal line is a direct line, a string pointer to its device–name should be placed in the *line* element in the CALL structure. Legal values for such terminal device names are kept in the *L–devices* file. In this case, the value of the *baud* element need not be specified as it will be determined from the *L–devices* file.

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. Such numbers may consist only of symbols described on the *acu*(7). The termination symbol will be supplied by the *dial* function, and should not be included in the *telno* string passed to *dial* in the CALL structure.

The CALL element *modem* is used to specify whether or not modem control is required for direct lines. This element should be non–zero if modem control is required for the line. The CALL element *attr* is a pointer to a *termio* structure, as defined in the *termio.h* header file. A NULL value for this pointer element may be passed to the *dial* function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud–rate.

The CALL element *device* is used to hold the device name (cul..) that establishes the connection.

The CALL element *dev_len* is the length of the device name that is copied into the array device.

FILES

/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..*tty-device*

SEE ALSO

uucp(1C), alarm(2), read(2), write(2), acu(4), termio(4).

DIAGNOSTICS

On failure, a negative value indicating the reason for the failure will be returned. Mnemonics for these negative indices as listed here are defined in the <*dial.h*> header file.

```
INTRPT     -1      /* interrupt occurred */
D_HUNG     -2      /* dialer hung (no return from write) */
NO_ANS     -3      /* no answer within 10 seconds */
ILL_BD     -4      /* illegal baud-rate */
A_PROB     -5      /* acu problem (open() failure) */
L_PROB     -6      /* line problem (open() failure) */
NO_Ldv     -7      /* can't open LDEVS file */
DV_NT_A    -8      /* requested device not available */
DV_NT_K    -9      /* requested device not known */
NO_BD_A    -10     /* no device available at requested baud */
NO_BD_K    -11     /* no device known at requested baud */
```

WARNINGS

Including the **<dial.h>** header file automatically includes the **<termio.h>** header file.

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

An *alarm*(2) system call for 3600 seconds is made (and caught) within the *dial* module for the purpose of "touching" the *LCK..* file and constitutes the device allocation semaphore for the terminal device. Otherwise, *uucp*(1C) may simply delete the *LCK..* entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a *read*(2) or *write*(2) system call, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from *read*s should be checked for **(errno==EINTR)**, and the *read* possibly reissued.

NAME

opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

```
#include <ndir.h>

DIR *opendir(filename)
char *filename;

struct direct *readdir(dirp)
DIR *dirp;

long telldir(dirp)
DIR *dirp;

seekdir(dirp, loc)
DIR *dirp;
long loc;

rewinddir(dirp)
DIR *dirp;

closedir(dirp)
DIR *dirp;
```

HP-UX COMPATIBILITY

Level:	HP-UX/STANDARD
Origin:	UCB

DESCRIPTION

The purpose of this library package is to provide functions which allow programs to read directory entries without having to know the actual directory format associated with the file system. This allows programs to be ported from one file system to another. Therefore, this is the recommended way to read directory entries.

Opendir opens the directory named by *filename* and associates a *directory stream* with it. *Opendir* returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer **NULL** is returned if *filename* cannot be accessed, if *filename* is not a directory, or if sufficient memory cannot be allocated for a buffer of size DIRBLKSIZ blocks (see HARDWARE DEPENDENCIES).

Readdir returns a pointer to the next directory entry. It returns **NULL** upon reaching the end of the directory or detecting an invalid *seekdir* operation.

Telldir returns the current location, in bytes, associated with the named *directory stream.*

Seekdir sets the position of the next *readdir* operation on the *directory stream. Loc* is a byte offset within the directory file. The new position reverts to the one associated with the *directory stream* when the *telldir* operation was performed. Values returned by *telldir* are good only for the lifetime of the DIR pointer from which they are derived. If the directory is closed and then re-opened, the *telldir* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir* value immediately after a call to *opendir* and before any calls to *readdir.*

Rewinddir resets the position of the named *directory stream* to the beginning of the directory.

Closedir causes the named *directory stream* to be closed, and the structure associated with the DIR pointer to be freed.

See */usr/include/ndir.h* for a description of the fields available in a directory entry. The preferred way to search the current directory for entry "name" is:

```
len = strlen(name);
dirp = opendir(".");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp)) {
```

```
                if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
                        closedir(dirp);
                        return FOUND;
                }
        }
        closedir(dirp);
        return NOT_FOUND;
```

HARDWARE DEPENDENCIES

Series 200:
Malloc(3C) is used to allocate memory.

Series 500:
Malloc(3C) is used to allocate memory.

SEE ALSO

/usr/include/ndir.h, close(2), lseek(2), open(2), read(2).

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 - generate uniformly distributed pseudo-random numbers

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

SYNOPSIS

double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval $[0.0,~1.0).$

Functions *lrand48* and *nrand48* return non-negative long integers uniformly distributed over the interval $[0,~2 sup 31).$

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval $[-2 sup 31 ,~2 sup 31).$

Functions *srand48, seed48* and *lcong48* are initialization entry points, one of which should be invoked before either *drand48, lrand48* or *mrand48* is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if *drand48, lrand48* or *mrand48* is called without a prior call to an initialization entry point.) Functions *erand48, nrand48* and *jrand48* do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, $X sub i ,$ according to the linear congruential formula

X sub{n+1}~=~(aX sub n^+^c) sub{roman mod~m}~~~~~~~~n>=0.

The parameter $m^=^2 sup 48$; hence 48-bit integer arithmetic is performed. Unless *lcong48* has been invoked, the multiplier value a and the addend value c are given by

a~mark =~roman 5DEECE66D^sub 16~=~roman 273673163155^sub 8
c~lineup =~roman B^sub 16~=~roman 13^sub 8 .

The value returned by any of the functions *drand48, erand48, lrand48, nrand48, mrand48* or *jrand48* is computed by first generating the next 48-bit $X sub i$ in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of $X sub i$ and transformed into the returned value.

The functions *drand48, lrand48* and *mrand48* store the last 48-bit $X sub i$ generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions *erand48, nrand48* and *jrand48* require the calling program to provide storage for the successive $X sub i$ values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of $X sub i$ into the array and pass it as an argument. By using different arguments, functions *erand48, nrand48* and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *srand48* sets the high-order 32 bits of $X sub i$ to the 32 bits contained in its argument. The low-order 16 bits of $X sub i$ are set to the arbitrary value $roman 330E sub 16 .$

The initializer function *seed48* sets the value of $X sub i$ to the 48-bit value specified in the argument array. In addition, the previous value of $X sub i$ is copied into a 48-bit internal buffer, used only by *seed48,* and a pointer to this buffer is the value returned by *seed48.* This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last $X sub i$ value, and then use this value to reinitialize via *seed48* when the program is restarted.

The initialization function *lcong48* allows the user to specify the initial $X sub i ,$ the multiplier value $a,$ and the addend value $c.$ Argument array elements *param[0-2]* specify $X sub i ,$ *param[3-5]* specify the multiplier $a,$ and *param[6]* specifies the 16-bit addend $c.$ After *lcong48* has been called, a subsequent call to either *srand48* or *seed48* will restore the "standard" multiplier and addend values, a and $c,$ specified on the previous page.

SEE ALSO

rand(3C).

NAME

ecvt, fcvt, gcvt, nl_gcvt - convert floating–point number to string

SYNOPSIS

char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
int ndigit;
char *buf;

char *nl_gcvt (value, ndigit, buf, langid)
double value;
int ndigit;
char *buf;
int langid;

HP–UX COMPATIBILITY

Level: HP–UX/RUN ONLY

Origin: System V

Native Language Support:
8-bit data, customs, messages

DESCRIPTION

Ecvt converts *value* to a null–terminated string of *ndigit* digits and returns a pointer thereto. The high-order digit is non–zero, unless the value is zero. The low–order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non–zero, otherwise it is zero.

Fcvt is identical to *ecvt*, except that the correct digit has been rounded for printf "%f" (FORTRAN F–format) output of the number of digits specified by *ndigit.*

Gcvt converts the *value* to a null–terminated string in the array pointed to by *buf* and returns *buf.* It attempts to produce *ndigit* significant digits in FORTRAN F–format if possible, otherwise E–format, ready for printing. A minus sign, if there is one, or a decimal point will be included as part of the returned string. Trailing zeros are suppressed.

NL_gcvt differs from *gcvt* only in that it uses *langid* to determine what the radix character should be (e.g., '.' or ','). If *langid* is not valid, or information for *langid* has not been installed, the radix character defaults to a period.

SEE ALSO

printf(3S), hpnls(7), langid(7).

BUGS

The values returned by *ecvt* and *fcvt* point to a single static data array whose content is overwritten by each call.

NAME

end, etext, edata - last locations in program

SYNOPSIS

extern char end;
extern char etext;
extern char edata;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialized data region, and *end* above the uninitialized data region. Note that the definition of each of these is implementation-dependent. See *HARDWARE DEPENDENCIES* below.

When execution begins, the program break (the first location beyond the data) coincides with *end*, but the program break may be reset by the routines of *brk*(2), *malloc*(3C), standard input/output (*stdio*(3S)), the profile (**-p**) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by **sbrk(0)** (see *brk*(2)).

HARDWARE DEPENDENCIES

Series 500:

End is the lowest heap address available to the user. *Etext* and *Edata* are not supported.

Memallc(2) is more efficient than *malloc*(3C) for setting the program break.

SEE ALSO

cc(1), brk(2), malloc(3C), stdio(3S).

NAME

erf, erfc - error function and complementary error function

SYNOPSIS

#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Erf returns the error function of x, defined as $\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$.

Erfc, which returns 1.0 - *erf(x)*, is provided because of the extreme loss of relative accuracy if *erf(x)* is called for large x and the result subtracted from 1.0 (e.g., for $x = 5$, 12 places are lost).

SEE ALSO

exp(3M).

NAME

exp, log, log10, pow, sqrt - exponential, logarithm, power, square root functions

SYNOPSIS

```
#include <math.h>

double exp (x)            float fexp (x)
double x;                ‡float x;

double log (x)            float flog (x)
double x;                ‡float x;

double log10 (x)          float flog10 (x)
double x;                ‡float x;

double pow (x, y)         float fpow (x,y)
double x, y;             ‡float x,y;

double sqrt (x)           float fsqrt (x)
double x;                ‡float x;

                               ‡ see important note below
```

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: System V

DESCRIPTION

Exp returns e^x.

Log returns the natural logarithm of x. The value of x must be positive.

Log10 returns the logarithm base ten of x. The value of x must be positive.

Pow returns x^y. If x is zero, y must be positive. If x is negative, y must be an integer.

Sqrt returns the non-negative square root of x. The value of x may not be negative.

IMPORTANT NOTE: The corresponding single-precision routines *fexp*, *flog*, *flog10*, *fpow*, and *fsqrt* expect true single-precision arguments, and therefore cannot be called from standard C. They are provided for support of FORTRAN and Pascal.

HARDWARE DEPENDENCIES

Series 200/500:

The algorithms used are those from HP 9000 BASIC.

DIAGNOSTICS

Exp sets *errno* to **ERANGE** and returns **HUGE** when the correct value would overflow, or 0 when the correct value would underflow.

Log and *log10* return **-HUGE** and set *errno* to **EDOM** when x is non-positive. A message indicating DOMAIN error (or SING error when x is 0) is printed on the standard error output.

Pow returns 0 and sets *errno* to **EDOM** when x is 0 and y is non-positive, or when x is negative and y is not an integer. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for *pow* would overflow or underflow, *pow* returns **±HUGE** or 0 respectively, and sets *errno* to **ERANGE**.

Sqrt returns 0 and sets *errno* to **EDOM** when x is negative. A message indicating DOMAIN error is printed on the standard error output.

Error handling is identical for both single- and double-precision routines, except for one consideration: In any situation where the double-precision routine would return ±HUGE, the corresponding single-precision routine returns ±MAXFLOAT.

These error–handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

hypot(3M), matherr(3M), sinh(3M).

NAME

fclose, fflush - close or flush a stream

SYNOPSIS

#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Fclose causes any buffered data for the named *stream* to be written out, and the *stream* to be closed. Buffers allocated by the standard input/output system are freed.

Fclose is performed automatically for all open files upon calling *exit*(2).

Fflush causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

DIAGNOSTICS

These functions return 0 for success, and **EOF** if any error (such as trying to write to a file that has not been opened for writing) was detected.

SEE ALSO

close(2), exit(2), fopen(3S), setbuf(3S).

NAME

ferror, feof, clearerr, fileno - stream status inquiries

SYNOPSIS

#include <stdio.h>

int ferror (stream)
FILE
***stream;**

int feof (stream)
FILE
***stream;**

void clearerr (stream)
FILE
***stream;**

int fileno (stream)
FILE
***stream;**

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise zero. Unless cleared by *clearerr*, or unless the specific *stdio* routine so indicates, the error indication lasts until the stream is closed.

Feof returns non-zero when **EOF** has previously been detected reading the named input *stream*, otherwise zero.

Clearerr resets the error indicator and **EOF** indicator to zero on the named *stream*.

Fileno returns the integer file descriptor associated with the named *stream*; see *open*(2).

NOTE

All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO

open(2), fopen(3S).

NAME

floor, ceil, fmod, fabs - floor, ceiling, remainder, absolute value functions

SYNOPSIS

#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Floor returns the largest integer (as a double-precision number) not greater than x.

Ceil returns the smallest integer not less than x.

Fmod returns the floating-point remainder of the division of x by y: zero if y is zero or if x/y would overflow; otherwise the number f with the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

Fabs returns the absolute value of x, $|x|$.

SEE ALSO

abs(3C).

NAME

fopen, freopen, fdopen - open or re–open a stream file; convert file to stream

SYNOPSIS

#include <stdio.h>

FILE *fopen (file_name, type)
char *file_name, *type;

FILE *freopen (file_name, type, stream)
char *file_name, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

DESCRIPTION

Fopen opens the file named by *file_name* and associates a *stream* with it. *Fopen* returns a pointer to the FILE structure associated with the *stream*.

File_name points to a character string that contains the name of the file to be opened.

Type is a character string having one of the following values:

"r"	open for reading
"w"	truncate or create for writing
"a"	append; open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end–of–file

Freopen substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. *Freopen* returns a pointer to the FILE structure associated with *stream*.

Freopen is typically used to attach the preopened *streams* associated with **stdin**, **stdout** and **stderr** to other files.

Fdopen associates a *stream* with a file descriptor. File descriptors are obtained from *open*, *dup*, *creat*, or *pipe*(2), which open files but do not return pointers to a FILE structure *stream*. Streams are necessary input for many of the Section 3S library routines. The *type* of *stream* must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end–of–file.

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. *Fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

SEE ALSO

creat(2), dup(2), open(2), fclose(3S), pipe(2), fclose(3S), fseek(3S), popen(3S).

DIAGNOSTICS

Fopen and *freopen* return a NULL pointer if *file–name* cannot be accessed, if there are too many open files, or if the arguments are incorrect.

Fdopen returns a **NULL** if there are too many open files, or if the arguments are ill–formed.

NAME

fread, fwrite - buffered binary input/output to a stream file

SYNOPSIS

```
#include <stdio.h>

int fread (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Fread copies, into an array pointed to by *ptr*, *nitems* items of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. *Fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream,* or if *nitems* items have been read. *Fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *Fread* does not change the contents of *stream*.

Fwrite appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. *Fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *Fwrite* does not change the contents of the array pointed to by *ptr*.

The argument *size* is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

SEE ALSO

read(2), write(2), fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S).

DIAGNOSTICS

Fread and *fwrite* return the number of items read or written. If *size* or *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

NAME

frexp, ldexp, modf - split floating–point into mantissa and exponent

SYNOPSIS

double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp;

double modf (value, iptr)
double value, *iptr;

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

DESCRIPTION

Every non–zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) x is in the range $0.5 \leq | x | < 1.0$, and the "exponent" n is an integer.

Frexp returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by *frexp* are zero.

Ldexp returns the quantity $value * 2^{exp}$.

Modf returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

DIAGNOSTICS

If *ldexp* would cause overflow, ±**HUGE** is returned (according to the sign of *value*), and *errno* is set to **ERANGE**.
If *ldexp* would cause underflow, zero is returned and *errno* is set to **ERANGE**.

NAME

fseek, rewind, ftell - reposition a file pointer in a stream

SYNOPSIS

#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

long rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according as *ptrname* has the value 0, 1, or 2.

Rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0).

Fseek and *rewind* undo any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output. *Rewind* also does an implicit *clearerr*(3s) call.

Ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

SEE ALSO

lseek(2), fopen(3S), popen(3S), ungetc(3S).

DIAGNOSTICS

Fseek returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; in particular, *fseek* may not be used on a terminal, or on a file opened via *popen*(3S).

Ftell returns -1 for error conditions.

WARNING

Although on HP-UX an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX systems requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

NAME

ftw - walk a file tree

SYNOPSIS

#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Ftw recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure (see *stat*(2)) containing information about the object, and an integer. Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

Ftw visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero. If *fn* returns a nonzero value, *ftw* stops its tree traversal and returns whatever value was returned by *fn*. If *ftw* detects an error, it returns -1, and sets the error type in *errno*.

Ftw uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *Depth* must not be greater than the number of file descriptors currently available for use. *Ftw* will run more quickly if *depth* is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3C).

BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.
It could be made to run faster and use less storage on deep structures at the cost of considerable complexity.
Ftw uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

NAME

gamma, signgam - log gamma function

SYNOPSIS

#include <math.h>

double gamma (x)
double x;

extern int signgam;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Gamma returns $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as $\int_0^{\infty} e^{-t} t^{x-1} dt$. The sign of $\Gamma(x)$ is returned in the external integer *signgam*. The argument x may not be a non-positive integer. (*Gamma* is defined over the reals excluding the non-positive integers).

The following C program fragment might be used to calculate Γ:

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
        error( );
y = signgam * exp(y);
```

where LN_MAXDOUBLE is the least value that causes *exp*(3M) to return a range error, and is defined in the <*values.h*> header file.

DIAGNOSTICS

For non-positive integer arguments **HUGE** is returned, and *errno* is set to **EDOM**. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, *gamma* returns **HUGE** and sets *errno* to **ERANGE**.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

exp(3M), matherr(3M), values(5).

NAME

getc, getchar, fgetc, getw - get character or word from a stream file

SYNOPSIS

#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ( )

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Getc returns the next character (i.e., byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. *Getchar* is defined as *getc(stdin)*. *Getc* and *getchar* are macros and so cannot be used if a function is necessary; for example one cannot have a function pointer point to them.

Fgetc behaves like *getc*, but is a function rather than a macro. *Fgetc* runs more slowly than *getc*, but it takes less space per invocation and its name can be passed as an argument to a function.

Getw returns the next word (i.e. *int* in C) from the named input *stream*. *Getw* increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. *Getw* assumes no special alignment in the file.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S).

DIAGNOSTICS

These functions return the constant **EOF** at end-of-file or upon an error. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *getw* errors.

WARNING

If the integer value returned by *getc*, *getchar*, or *fgetc* is stored into a character variable and then compared against the integer constant **EOF**, the comparison may never succeed, because sign-extension of a character on widening to integer is machine-dependent.

BUGS

Because it is implemented as a macro, *getc* treats incorrectly a *stream* argument with side effects. In particular, **getc(*f++)** does not work sensibly. *Fgetc* should be used instead.
Because of possible differences in word length and byte ordering, files written using *putw* are machine–dependent, and may not be read using *getw* on a different (non–HP–UX) processor.

NAME

getcwd - get path–name of current working directory

SYNOPSIS

char *getcwd (buf, size)
char *buf;
int size;

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

DESCRIPTION

Getcwd returns a pointer to the current directory path–name. The value of *size* must be at least two greater than the length of the path–name to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free.*

The function is implemented by using *popen*(3S) to pipe the output of the *pwd*(1) command into the specified string space.

EXAMPLE

```
char *cwd, *getcwd();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror("pwd");
        exit(1);
}
printf("%s\n", cwd);
```

SEE ALSO

pwd(1), malloc(3C), popen(3S).

DIAGNOSTICS

Returns **NULL** with *errno* set if *size* is not large enough, or if an error ocurrs in a lower–level function.

NAME

getenv - return value for environment name

SYNOPSIS

char *getenv (name)
char *name;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Getenv searches the environment list (see *environ*(7)) for a string of the form *name=value*, and returns a pointer to the *value* in the current environment if such a string is present, otherwise a NULL pointer. *Name* may be either the desired name, null-terminated, or of the form *name=value,* in which case *getenv* uses the portion to the left of the "=" as the search key.

SEE ALSO

exec(2), putenv(3C), environ(5).

NAME

getfsent, getfsspec, getfsfile, getfstype, setfsent, endfsent - get file system descriptor file entry

HP-UX COMPATIBILITY

Level:	Large Machine/HP Extension
Origin:	HP and UCB

SYNOPSIS

#include <checklist.h>

struct checklist *getfsent()

struct checklist *getfsspec(spec)
char *spec;

struct checklist *getfsfile(file)
char *file;

struct checklist *getfstype(type)
char *type;

int setfsent()

int endfsent()

DESCRIPTION

Getfsent, *getfsspec*, *getfsfile*, and *getfstype* each return a pointer to an object with the following structure containing the broken-out fields of a line in the /etc/checklist file. The structure is declared in the <checklist.h> header file:

```
struct checklist{
        char    *fs_spec;    /* special file name       */
        char    *fs_bspec;   /* block special file name */
        char    *fs_file;    /* file sys directory name */
        char    *fs_type;    /* type: ro, rw, sw, xx    */
        int     fs_passno;   /* fsck pass number        */
        int     fs_freq;     /* backup frequency        */
};
```

The fields have meanings described in *checklist*(5). If the block special file name, the file system directory name, the type and the pass number are not all defined on the associated line in /etc/checklist, these routines will return pointers to NULL in the fs_bspec, fs_file and fs_type fields and -1 in the fs_passno field. Fs_freq is reserved for future use. If the fs_freq field is not present on the line these routines will return -1 in the fs_freq field.

Getfsent reads the next line of the file, opening the file if necessary.

Setfsent opens and rewinds the file.

Endfsent closes the file.

Getfsspec and *getfsfile* sequentially search from the beginning of the file until a matching special file name or file system file name is found, or until EOF is encountered. *Getfstype* does likewise, matching on the file system type field.

FILES

/etc/checklist

SEE ALSO

checklist(5)

DIAGNOSTICS

Null pointer (0) returned on EOF, invalid entry or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent - get group file entry

SYNOPSIS

```
#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )

struct group *fgetgrent (f)
FILE *f;
```

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System V

DESCRIPTION

Getgrent, *getgrgid* and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the **/etc/group** file. Each line contains a "group" structure, defined in the <*grp.h*> header file.

```
struct      group {
            char        *gr_name;  /* the name of the group */
            char        *gr_passwd;             /* the encrypted group password */
            int         gr_gid;     /* the numerical group ID */
            char        **gr_mem; /* vector of pointers to member names */
};
```

Getgrent when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. *Getgrgid* searches from the beginning of the file until a numerical group id matching *gid* is found and returns a pointer to the particular structure in which it was found. *Getgrnam* searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *Endgrent* may be called to close the group file when processing is complete.

Fgetgrent returns a pointer to the next group structure in the stream *f*, which matches the format of **/etc/group**.

FILES

/etc/group

SEE ALSO

getlogin(3C), getpwent(3C), group(5).

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getlogin - get login name

SYNOPSIS

char *getlogin ( );

HP-UX COMPATIBILITY

Level:	HP-UX/NUCLEUS
Origin:	System V

DESCRIPTION

Getlogin returns a pointer to the login name as found in **/etc/utmp**. It may be used in conjunction with *getpwnam* to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, it returns a **NULL** pointer. The correct procedure for determining the login name is to call *cuserid*, or to call *getlogin* and if it fails to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(5).

DIAGNOSTICS

Getlogin returns the **NULL** pointer if *name* is not found.

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

getmsg - get message from a catalog

SYNOPSIS

```
char *getmsg (fd, set_num, msg_num, buf, buflen)
int fd, set_num, msg_num, buflen;
char buf[];
```

HP-UX COMPATIBILITY

Level:	HP-UX/STANDARD
Origin:	HP

Native Language Support:
8-bit data, customs, messages

DESCRIPTION

Getmsg attempts to read up to *buflen-1* bytes of a message string into the area pointed to by *buf*. A null byte is inserted to terminate the string placed in the buffer.

Fd is the file descriptor returned by a call to *open(2)* the catalog containing the messages. *Set_num* is available to group messages together into a logical unit. For instance, messages in Finnish could be grouped in set number 6 to match the language id for Finnish (See *currlangid(3C)* and *langid(7))*.

DIAGNOSTICS

Returns a pointer to an empty (null) string if *fd* is invalid or *set_num* or *msg_num* is not in the catalog.

SEE ALSO

gencat(1), insertmsg(1), read(2), hpnls(7).

NAME

getopt, optarg, optind, opterr - get option letter from argument vector

SYNOPSIS

int getopt (argc, argv, optstring)
int argc;
char **argv, *opstring;

extern char *optarg;
extern int optind, opterr;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Getopt returns the next option letter in *argv* (starting from *argv[1]*) that matches a letter in *optstring*. *Optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. *Optarg* is set to point to the start of the option argument on return from *getopt*.

Getopt places in *optind* the *argv* index of the next argument to be processed. Because *optind* is external, it is normally initialized to zero automatically before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **EOF**. The special option **--** may be used to delimit the end of the options; **EOF** will be returned, and **--** will be skipped.

DIAGNOSTICS

Getopt prints an error message on *stderr* and returns a question mark (**?**) when it encounters an option letter not included in *optstring*. This error message may be disabled by setting *opterr* to zero.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

Options can be any ASCII characters except colon (:), question mark (?), or null (\0). It is impossible to distinguish between a ? used as a legal option, and the character that *getopt* returns when it encounters an invalid option character in the input.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern char *optarg;
        extern int optind;
        .
        .
        .
        while ((c = getopt(argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bproc( );
                        break;
                case 'f':
                        ifile = optarg;
                        break;
                case 'o':
                        ofile = optarg;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                fprintf(stderr, "usage: . . . ");
                exit (2);
        }
        for ( ; optind < argc; optind++) {
                if (access(argv[optind], 4)) {
        :
}
```

SEE ALSO

getopt(1).

NAME

getpass - read a password

SYNOPSIS

char *getpass (prompt)
char *prompt;

HP-UX COMPATIBILITY

Level:	HP-UX/NUCLEUS
Origin:	System V

DESCRIPTION

Getpass reads up to a newline or **EOF** from the file **/dev/tty**, after prompting on the standard error output with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. If **/dev/tty** cannot be opened, a **NULL** pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

FILES

/dev/tty

SEE ALSO

crypt(3C).

WARNING

The above routine uses <**stdio.h**>, which causes it to increase the size of programs not otherwise using standard I/O, more than might be expected.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpw - get name from UID

SYNOPSIS

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

HP-UX COMPATIBILITY

Level:	HP-UX/NUCLEUS
Origin:	System V

DESCRIPTION

Getpw searches the password file for a user id number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. *Getpw* returns non–zero if *uid* cannot be found. The line is null–terminated.

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(5).

DIAGNOSTICS

Getpw returns non–zero on error.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase, more than might be expected, the size of programs not otherwise using standard I/O.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent - get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )

struct passwd *fgetpwent (f)
FILE *f;
```

HP-UX COMPATIBILITY

Level:	HP-UX/NUCLEUS
Origin:	System V

DESCRIPTION

Getpwent, *getpwuid* and *getpwnam* each returns a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/passwd** file. Each line in the file contains a "passwd" structure, declared in the <*pwd.h*> header file:

```
struct passwd {
        char            *pw_name;
        char            *pw_passwd;
        int             pw_uid;
        int             pw_gid;
        char            *pw_age;
        char            *pw_comment;
        char            *pw_gecos;
        char            *pw_dir;
        char            *pw_shell;
};
struct comment {
        char            *c_dept;
        char            *c_name;
        char            *c_acct;
        char            *c_bin;
};
```

This structure is declared in <*pwd.h*> so it is not necessary to redeclare it.

The *pw_comment* field is unused; the others have meanings described in *passwd*(5).

Getpwent when first called returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. *Getpwuid* searches from the beginning of the file until a numerical user id matching *uid* is found and returns a pointer to the particular structure in which it was found. *Getpwnam* searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *Endpwent* may be called to close the password file when processing is complete.

Fgetpwent returns a pointer to the next passwd structure in the stream *f*, which matches the format of **/etc/passwd**.

FILES

/etc/passwd

SEE ALSO

getlogin(3C), getgrent(3C), passwd(4).

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

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

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

DESCRIPTION

Gets reads characters from the standard input stream, *stdin,* into the array pointed to by *s*, until a new–line character is read or an end–of–file condition is encountered. The new–line character is discarded and the string is terminated with a null character.

Fgets reads characters from the *stream* into the array pointed to by *s*, until *n*-1 characters are read, or a new–line character is read and transferred to *s*, or an end–of–file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), puts(3S), scanf(3S).

DIAGNOSTICS

If end–of–file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

NAME

getutent, getutid, getutline, pututline, setutent, endutent, utmpname - access utmp file entry

SYNOPSIS

#include <types.h>
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

void utmpname (file)
char *file;

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: System V

DESCRIPTION

Getutent, *getutid* and *getutline* each return a pointer to a structure of the following type:

```
struct      utmp {
            char          ut_user[8]; /* User login name */
            char          ut_id[4];   /* /etc/inittab id (usually line #) */
            char          ut_line[12];          /* device name (console, lnxx) */
            short         ut_pid;     /* process id */
            short         ut_type;    /* type of entry */
            struct        exit_status {
               short         e_termination;  /* Process termination status */
               short         e_exit; /* Process exit status */
            } ut_exit;                /* The exit status of a process
                                       * marked as DEAD_PROCESS. */
            time_t        ut_time;    /* time entry was made */
};
```

Getutent reads in the next entry from a *utmp*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

Getutid searches forward from the current point in the *utmp* file until it finds an entry with a *ut_type* matching *id->ut_type* if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME or NEW_TIME. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS or DEAD_PROCESS, then *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id->ut_id*. If the end of file is reached without a match, it fails.

Getutline searches forward from the current point in the *utmp* file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* string matching the *line->ut_line* string. If the end of file is reached without a match, it fails.

Pututline writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of *pututline* will have searched for the proper entry using one of the *getut* routines. If so, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add a new entry to the end of the file.

Setutent resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

Endutent closes the currently open file.

Utmpname allows the user to change the name of the file examined, from **/etc/utmp** to any other file. It is most often expected that this other file will be **/etc/wtmp**. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. *Utmpname* does not open the file. It just closes the old file if it is currently open and saves the new file name.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

A **NULL** pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

COMMENTS

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurrences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

NAME

gpio_get_status – return status lines of GPIO card

SYNOPSIS

int gpio_get_status (eid)
int eid;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Gpio_get_status enables you to read the status register of the GPIO interface associated with the device file identified by *eid*. *Eid* is an entity identifier of an open GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2). The current state of each status line on the interface card is mapped to the value returned, with STS0 mapped to the least significant bit. Only x least-significant bits are used, where x is the number of status lines available on the hardware interface being used.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:

For the current GPIO card, x is 2.

RETURN VALUE

Gpio_get_status returns the value of the status register of the GPIO interface associated with *eid*, and −1 if an error was encountered.

DIAGNOSTICS

Gpio_get_status fails under the following conditions and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to a GPIO device file [ENOTTY].

NAME

gpio_set_ctl – set control lines on GPIO card

SYNOPSIS

int gpio_set_ctl (eid, value)
int eid, value;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Gpio_set_ctl enables you to set the control register of a GPIO interface. *Eid* is an entity identifier of an open GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Value* is the value to be written into the control register of the GPIO interface associated with *eid*.

Value is mapped onto the control lines on the interface card, with the least significant bit mapped to CTL0. Only the x least significant bits are used, where x is the number of control lines available on the hardware interface being used.

HARDWARE DEPENDENCIES

Series 200/300/500
Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:
For the current GPIO card, x is 2.

RETURN VALUE

Gpio_set_ctl returns 0 if successful, and –1 if an error was encountered.

DIAGNOSTICS

Gpio_set_ctl fails under the following circumstances and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to a GPIO device file [ENOTTY].

NAME

hpib_abort – stop activity on specified HP-IB bus

SYNOPSIS

int hpib_abort (eid);
int eid;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Hpib_abort terminates activity on the addressed HP-IB bus by pulsing the IFC line. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Hpib_abort also sets the REN line and clears the ATN line. The status of the SRQ line is not affected. The interface must be the system controller of the bus.

RETURN VALUE

Hpib_abort returns 0 (zero) if successful, or –1 if an error was encountered.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

DIAGNOSTICS

Hpib_abort fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY];

the specified interface is not the system controller [EIO].

NAME

hpib_bus_status – return status of HP-IB interface

SYNOPSIS

int hpib_bus_status (eid, status);
int eid, status;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Hpib_bus_status enables you to determine selected status information about an HP-IB channel. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Status* is an integer determining what status information is returned for a particular call. The values defined for *status* and their associated meanings are:

Value	Meaning
0	Is the channel currently in remote state?
1	What is the current state of the SRQ line?
2	What is the current state of the NDAC line?
3	Is the channel currently system controller?
4	Is the channel currently active controller?
5	Is the channel currently addressed as talker?
6	Is the channel currently addressed as listener?
7	What is the channel's bus address?

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:

A bug in the HP27110A HP-IB interface causes an erroneous report of the state of the SRQ line. There is a small window when *hpib_bus_status*(eid, 1) reports that the SRQ line is clear when in reality it is set. OR-ing together five successive readings of the state of the SRQ line yields a reading of about 99% accuracy.

The remote state status is not defined when the interface is the active controller, although reading remote state status in such a situation is not an error.

Series 200/300:

The status of those lines being driven by the interface is undefined, although reading them in such a situation is not an error. Non-active controllers cannot sense the SRQ line. Active listeners cannot sense the NDAC line.

RETURN VALUE

Hpib_bus_status's return value depends upon the value of *status*, as follows:

Status	Return Value	Meaning
—	-1	Error condition.
0 – 6	0	False condition (line is clear).
0 – 6	1	True condition (line is set).
7	0 – 30	Bus address of interface card.

DIAGNOSTICS

Hpib_bus_status fails under the following conditions, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY];

status is outside the range [0-7] [EINVAL].

NAME

hpib_card_ppoll_resp – control response to parallel poll on HP-IB

SYNOPSIS

int hpib_card_ppoll_resp (eid,flag);
int eid,flag;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Hpib_card_ppoll_resp enables an interface to enable (or disable) itself for parallel polls. It also controls the sense, and determines the line on which the response is sent. This gives the interface the ability to either ignore or respond to a parallel poll depending upon whether or not it is enabled to respond.

Eid is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer having one of the following bit patterns:

Bit Pattern	**Meaning**
10000	Disable parallel poll response.
0SPPP	Enable parallel poll response, where S = sense of the response, and PPP = 3-bit binary number specifying the line on which the response is sent (0 – 7 octal).

HARDWARE DEPENDENCIES

Series 200/300/500
Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:
Note that the HP 27110A/B HP-IB interface cards do not support programmatic parallel poll response configuration.

RETURN VALUE

Hpib_card_ppoll_resp returns 0 (zero) if successful, or –1 if an error was encountered.

SEE ALSO

hpib_ppoll(3I) and hpib_ppoll_resp_ctl(3I).

DIAGNOSTICS

Hpib_card_ppoll_resp fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY];

NAME

hpib_eoi_ctl – control EOI mode for HP-IB file

SYNOPSIS

int hpib_eoi_ctl (eid, flag);
int eid, flag;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Hpib_eoi_ctl enables you to turn EOI mode on or off. *Eid* is an entity identifier of an open HP-IB raw device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer which, if non-zero, enables EOI mode, and otherwise disables it.

EOI mode causes the last byte of all subsequent write operations to be written out with the EOI line asserted, signifying the end of the data transmission. By default, EOI mode is disabled when the device file is opened.

Entity ids for the same device file obtained by separate *open*(2) requests have their own EOI modes associated with them. Entity ids for the same device file obtained by *dup*(2) or inherited by a *fork*(2) request share the same EOI mode. In the latter case, if one process enables EOI mode, then EOI mode is in effect for all such file descriptors.

RETURN VALUE

Hpib_eoi_ctl returns a 0 (zero) if successful, or –1 if an error was encountered.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

DIAGNOSTICS

Hpib_eoi_ctl fails under any of the following circumstances and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB device file [ENOTTY].

NAME

hpib_io – perform I/O with an HP-IB channel from buffers

SYNOPSIS

#include <dvio.h>
int hpib_io(eid, iovec, iolen)
int eid;
struct iodetail *iovec;
int iolen;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Hpib_io enables you to perform and control read and/or write operations on the specified HP-IB bus. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Iovec* is a pointer to an array of structures of the form:

```
struct iodetail {
        char mode;
        char terminator;
        int  count;
        char *buf;
};
```

The *iodetail* structure is defined in the include file **libdvio.h**. *Iolen* specifies the number of structures in *iovec.*

The *mode* parameter in the *iodetail* structure describes what is to be done during I/O on the buffer pointed to by *buf. Mode* is constructed by OR-ing flags from the following list:

Only one of the following two flags *must* be specified:

HPIBREAD	Perform a read of the HP-IB bus, placing data into the accompanying buffer.
HPIBWRITE	Perform a write to the HP-IB bus, using data from the accompanying buffer.

The following flags may be used in most combinations (not all combinations are valid), or not at all:

HPIBATN	Data is written with ATN enabled.
HPIBEOI	Data written is terminated with EOI (this flag is ignored when HPIBATN is enabled).
HPIBCHAR	Data read is terminated with the character given in the *terminator* element of the *iodetail* structure.

Terminator describes the termination character, if any, that should be checked for on input. *Count* is an integer specifying the maximum number of bytes to be transferred.

A read operation terminates when either *count* is matched, an EOI is detected, or the designated *terminator* is detected (if HPIBCHAR is set in *mode*).

A write operation terminates when *count* is matched, and the final byte is sent with EOI asserted (if HPIBEOI is set in *mode*).

If HPIBATN is set in *mode*, then write operations occur with ATN enabled. Setting HPIBATN for a read operation is ignored and has no effect.

The members of the *iovec* array are accessed in order.

RETURN VALUES

If all transactions are successful, *hpib_io* returns a zero and updates the *count* element in each structure in the *iovec* array to reflect the actual number of bytes read or written.

If an error is encountered during a transaction defined by an element of *iovec*, *hpib_io* returns without completing any transactions that might follow. In particular, if an error occurs, *hpib_io* returns a −1, and the *count* element of the transaction which caused the error is set to −1.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

DIAGNOSTICS

Hpib_io fails under any of the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY];

a timeout occurs [EIO];

eid is not the active controller [EIO].

NAME

hpib_pass_ctl – change active controllers on HP-IB

SYNOPSIS

int hpib_pass_ctl (eid, ba)
int eid, ba;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Hpib_pass_ctl passes control of a bus to an inactive controller on that bus. The inactive controller becomes the active controller of that bus. *Eid* is an entity identifer of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Ba* is the bus address of the intended device.

Not all devices can accept control. Pass control passes only active control of the bus. It cannot pass system control of the bus. The specified interface must be the current active controller but need not be the system controller. The pass control operation does not suspend your program if the inactive controller does not take active control of the bus. However, the interface is no longer active controller.

RETURN VALUE

Hpib_pass_ctl returns 0 (zero) if successful, or –1 if an error was encountered.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

DIAGNOSTICS

Hpib_pass_ctl fails under any of the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY];

the interface is not the active controller [EIO];

ba does not refer to a valid bus address [EINVAL].

NAME

hpib_ppoll – conduct parallel poll on HP-IB bus

SYNOPSIS

int hpib_ppoll (eid);
int eid;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Hpib_ppoll conducts a parallel poll on an HP-IB bus. *eid* is a file descriptor of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Devices enabled to respond and that are in need of service can then assert the appropriate DIO line. This enables the controller to determine which devices, if any, need service at a given time. *Hpib_ppoll* raises attention (ATN) and end or identify (EOI) lines for 25 microseconds before reading the response. The interface must be the active controller to conduct a parallel poll.

RETURN VALUE

Hpib_ppoll returns an integer value whose least significant byte corresponds to the byte formed by the 8 data input/output (DIO) lines. Devices enabled to respond to a parallel poll do so on the appropriate DIO line. DIO line 0 corresponds to the least significant bit in the response byte. A –1 return value indicates that an error occurred.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

DIAGNOSTICS

Hpib_ppoll fails under the following situations, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY];

the interface is not current the active controller [EIO].

NAME

hpib_ppoll_resp_ctl – Define interface parallel poll response

SYNOPSIS

int hpib_ppoll_resp_ctl (eid, response)
int eid, response;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Eid is an entity identifier of an open HP-IB raw bus device file, obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Hpib_ppoll_resp_ctl defines a response to be sent when an active controller performs a parallel poll on an HP-IB interface. The value of *response* indicates whether this computer does or does not need service. A non-zero *response* value indicates that service is required. This statement only sets up a potential response; no actual response if generated when the statement is executed. The sense of the response and the line number to respond on are set by *hpib_card_ppoll_resp*(3) or by the active controller. When first opened, the default response and sense are 0.

RETURN VALUE

Hpib_ppoll_resp_ctl returns 0 if the response is successfully set, or -1 if an error has occured.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

DIAGNOSTICS

Hpib_ppoll_resp_ctl fails under the following situations, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF]

eid does not refer to a raw HP-IB device file [ENOTTY]

SEE ALSO

hpib_ppoll(3I), hpib_card_ppoll_resp(3I)

NAME

hpib_ren_ctl – control the Remote Enable line on HP-IB

SYNOPSIS

int hpib_ren_ctl (eid, flag);
int eid, flag;

HP-UX COMPATIBILITY

Level: Device I/O Library – HP-UX/EXTENDED

Origin: HP

DESCRIPTION

Hpib_ren_ctl enables/disables the Remote Enable (REN) line depending upon the value of *flag*. *Eid* is an entity identifer of an open HP-IB raw bus device file obtained from an *open*(2), *dup*2), *fcntl*(2),*or creat*(2)*call.* *Flag* is an integer which, if non-zero, enables the REN line, and otherwise disables it.

Hpib_ren_ctl, in conjunction with *hpib_send_cmnd*(3), enables you to place devices into the remote state or local state. The REN line is normally enabled at all times, and is in this state at power-up. Only the system controller may enable/disable the REN line.

RETURN VALUE

Hpib_ren_ctl returns 0 (zero) if successful, or –1 if an error was encountered.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

DIAGNOSTICS

Hpib_ren_ctl fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY];

the interface is not the system controller [EIO].

NAME

hpib_rqst_srvce – allow interface to enable SRQ line on HP-IB

SYNOPSIS

int hpib_rqst_srvce (eid, cv);
int eid, cv;

HP-UX COMPATIBILITY

Level: Device I/O Library – HP-UX/EXTENDED

Origin: HP

DESCRIPTION

Hpib_rqst_srvce specifies the response byte that the interface sends when it is serially polled by the active controller. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Cv* is an integer control value representation of the desired response byte.

Hpib_rqst_srvce optionally enables the SRQ line depending upon the response byte. If bit 6 of the response byte is set, the SRQ line is enabled. It remains enabled until the active controller conducts a serial poll or until the computer executes the request function with bit 6 cleared. The SRQ line is not enabled, however, as long as the interface is active controller. If bit 6 is set, the interface remembers its response byte, and enables the SRQ line when control is passed to another device on the bus.

The response byte looks as follows:

Bit	**Meaning**
0	SPOLL bit (least significant bit of response byte)
1	SPOLL bit
2	SPOLL bit
3	SPOLL bit
4	SPOLL bit
5	SPOLL bit
6	SRQ line
7	SPOLL bit (most significant bit of response byte)

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:

Note that the HP 27110A/B HP-IB interface cards allow only bit 6 to be set. All other bits remain cleared.

RETURN VALUE

Hpib_rqst_srvce returns 0 (zero) if successful, or –1 if an error was encountered.

DIAGNOSTICS

Hpib_rqst_srvce fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY].

NAME

hpib_send_cmnd – send command bytes over HP-IB

SYNOPSIS

int hpib_send_cmnd (eid, ca, length);
int eid, length;
char *ca;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Hpib_send_cmnd enables you to send arbitrary bytes of information on the HP-IB with the ATN line asserted. This enables you to configure and control the bus. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Ca* is a character pointer to a string of bytes to be written to the HP-IB bus as commands. *Length* is an integer specifying the number of bytes in the string pointed to by *ca.*

The interface must currently be the active controller in order to send commands over the bus.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Note that, for all HP-IB interfaces, both built-in and plug-in, the most significant bit of each byte is overwritten with a parity bit. All commands are written with odd parity.

RETURN VALUE

Hpib_send_cmnd returns 0 (zero) if successful, or –1 if an error was encountered.

DIAGNOSTICS

Hpib_send_cmnd fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY];

the interface is not currently the active controller [EIO].

NAME

hpib_spoll – conduct a serial poll on HP-IB bus

SYNOPSIS

int hpib_spoll (eid, ba);
int eid, ba;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Hpib_spoll conducts a serial poll of the specified device. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Ba* is the bus address of the intended device.

Hpib_spoll polls a single device for its response byte. The information stored in the response byte is device specific with the exception of bit 6. If bit 6 of the response byte is set, the addressed device has asserted the SRQ line, and is requesting service. (Note that the least significant (right-most) bit of the response byte is bit 0.)

Not all devices respond to the serial poll function. Consult the device documentation. Specifying a device that does not support serial polling may cause a timeout error or suspend your program indefinitely (see *hpib_rqst_srvce*(3)). The interface cannot serial poll itself. The interface must be the active controller.

RETURN VALUE

If *hpib_spoll* is successful, the device response byte is returned in the least significant byte of the return value. Otherwise, –1 is returned, indicating an error.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

SEE ALSO

hpib_rqst_srvce(3I).

DIAGNOSTICS

Hpib_spoll fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY];

the device polled did not respond before timeout, or the interface is not the active controller [EIO];

ba is the address of the polling interface itself or is an invalid bus address [EINVAL].

NAME

hpib_status_wait – wait until the requested status condition becomes true

SYNOPSIS

int hpib_status_wait (eid, status);
int eid,status;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Hpib_status_wait enables you to wait until a specific condition has occurred before returning. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Status* is an integer specifying what information is returned. The possible values for *status* and their associated meanings are:

Status	**Meaning**
1	Wait until the SRQ line is enabled.
4	Wait until this channel is the active controller.
5	Wait until this channel is addressed as talker.
6	Wait until this channel is addressed as listener.

The wait is subject to the current timeout in effect. If a timeout occurs before the desired condition occurs, the function returns with an error.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:

When an *hpip_status_wait* is in progress, all other bus activity is held off until it has completed. Therefore, it is strongly recommended that a timeout be in effect before all *hpib_status_wait* calls.

RETURN VALUE

Hpib_status_wait returns zero when the condition requested becomes true. A value of –1 is returned if an error occurs. A –1 is also returned if a timeout occurs before the desired condition becomes true.

DIAGNOSTICS

Hpib_status_wait fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY];

a timeout occured [EIO];

status contains an invalid value [EINVAL].

NAME

hpib_wait_on_ppoll – wait until a particular parallel poll value occurs

SYNOPSIS

int hpib_wait_on_ppoll (eid, mask, sense);
int eid, mask, sense;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Hpib_wait_on_ppoll waits for a parallel poll response to occur on one or more lines. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Mask is an integer that specifies on which line the parallel poll response is expected. *Mask*'s value is obtained from an 8-bit binary number, each bit of which corresponds to one of the eight lines. For example, if you want to wait for a response on lines 2 or 6, the correct binary number is 01000100. This converts to a decimal equivalent of 68, which is the number you should assign to *mask.*

Sense simply specifies what response you are expecting on the selected lines. *Sense* is constructed in the same way as *mask* – eight bits for eight lines. If a bit is set, then the function returns when the line corresponding that bit is *cleared.* Similarly, if a bit in *sense* is clear, the function returns when the corresponding line is *set.* Using the previous example, a *sense* = 00000100 = 4 (decimal) causes the function to return when line 6 is set, and return when line 2 is cleared.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:

When an *hpib_wait_on_ppoll* is in progress, all other bus activity is held off until it has completed. Therefore, it is strongly recommended that a timeout be in effect before all *hpib_wait_on_ppoll* calls.

RETURN VALUE

Hpib_wait_on_ppoll returns a value of –1 if an error or timeout condition occurs. A successful completion of the function returns the response byte XOR-ed with the *sense* value and AND-ed with the *mask.*

DIAGNOSTICS

Hpib_wait_on_ppoll fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY];

a timeout occured [EIO];

the interface is not currently the active controller [EIO].

NAME

hsearch, hcreate, hdestroy - manage hash search tables

SYNOPSIS

#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Hsearch is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *Item* is a structure of type ENTRY (defined in the *<search.h>* header file) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *Action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

Hcreate allocates sufficient space for the table, and must be called before *hsearch* is used. *Nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

Hdestroy destroys the search table, and may be followed by another call to *hcreate*.

EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it out.

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

struct info {           /* this is the info stored in the table */
        int age, room;  /* other than the key. */
};
#define NUM_EMPL    5000    /* # of elements in search table */

main( )
{
        /* space to store strings */
        char string_space[NUM_EMPL*20];
        /* space to store employee info */
        struct info info_space[NUM_EMPL];
        /* next avail space in string_space */
        char *str_ptr = string_space;
        /* next avail space in info_space */
```

```
                    struct info *info_ptr = info_space;
                    ENTRY item, *found_item, *hsearch( );
                    /* name to look for in table */
                    char name_to_find[30];
                    int i = 0;

                    /* create table */
                    (void) hcreate(NUM_EMPL);
                    while (scanf("%s%d%d", str_ptr, &info_ptr->age,
                            &info_ptr->room) != EOF && i++ < NUM_EMPL) {
                            /* put info in structure, and structure in item */
                            item.key = str_ptr;
                            item.data = (char *)info_ptr;
                            str_ptr += strlen(str_ptr) + 1;
                            info_ptr++;
                            /* put item into table */
                            (void) hsearch(item, ENTER);
                    }

                    /* access table */
                    item.key = name_to_find;
                    while (scanf("%s", item.key) != EOF) {
                        if ((found_item = hsearch(item, FIND)) != NULL) {
                            /* if item is in the table */
                            (void)printf("found %s, age = %d, room = %d\n",
                                    found_item->key,
                                    ((struct info *)found_item->data)->age,
                                    ((struct info *)found_item->data)->room);
                        } else {
                            (void)printf("no such employee %s\n",
                                    name_to_find)
                        }
                    }
            }
```

SEE ALSO

bsearch(3C), lsearch(3C), malloc(3C), malloc(3X), string(3C), tsearch(3C).

DIAGNOSTICS

Hsearch returns a NULL pointer if either the action is **FIND** and the item could not be found or the action is **ENTER** and the table is full.

Hcreate returns zero if it cannot allocate sufficient space for the table.

WARNING

Hsearch and *hcreate* use *malloc*(3C) to allocate space.

BUGS

Only one hash search table may be active at any given time.

NAME

hypot - Euclidean distance function

SYNOPSIS

#include <math.h>

double hypot (x, y)
double x, y;

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

DESCRIPTION

Hypot returns

sqrt(x * x + y * y),

taking precautions against unwarranted overflows.

DIAGNOSTICS

When the correct value would overflow, *hypot* returns **HUGE** and sets *errno* to **ERANGE.**

These error–handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M), sqrt(3F).

NAME

initgroups - initialize group access list

SYNOPSIS

initgroups(name, basegid)
char *name;
int basegid;

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: UCB

DESCRIPTION

Initgroups reads through the group file and sets up, using the *setgroups*(2) call, the group access list for the user specified in *name*. The *basegid* is automatically included in the groups list. Typically this value is given as the group number from the password file.

FILES

/etc/logingroup

SEE ALSO

setgroups(2)

DIAGNOSTICS

Initgroups returns -1 if it was not invoked by the super-user.

BUGS

Initgroups uses the routines based on *getgrent*(3). If the invoking program uses any of these routines, the group structure will be overwritten in the call to *initgroups*.

On most systems, no one seems to keep /etc/logingroup up to date.

NAME

intrapoff, intrapon - disable/enable integer trap handler

SYNOPSIS

int intrapoff()

int intrapon()

HP-UX COMPATIBILITY

Level:	HP-UX/NON-STANDARD
Origin:	HP
Remarks:	*Intrapoff* and *intrapon* are implemented on the Series 500 only.

DESCRIPTION

The Series 500 architecture has a single trap handler for both integer overflow (an integer value greater than 2^31–1) and integer divide–by–zero. By default, an operation which results in integer overflow or integer divide–by–zero invokes the integer trap handler. Any integer divide–by–zero generates the signal SIGFPE. As a side effect, any integer overflow also invokes the integer trap handler. The trap handler recognizes integer overflow as a special case and simply returns to the calling routine. A user sees no difference in results, but could see a severe performance degradation depending on how often the trap handler is invoked.

Intrapoff disables this integer trap handler. Integer overflow and integer divide–by–zero do not invoke the integer trap handler. Instead, integer divide–by–zero returns a large integer (2^31–1). Integer overflow operations simply overflow into the most significant bit. There is no performance penalty since the trap handler is not entered.

A program doing many integer overflows could see a significant performance improvement. A user must take care however, since integer divide–by–zero does not give signal SIGFPE while the integer trap handler is disabled.

Intrapon restores the default condition. Integer divide–by–zero and integer overflow operations invoke the integer trap handler. Integer divide–by–zero gives signal SIGFPE; integer overflow results in a performance penalty caused by entering and leaving the integer trap handler.

When *intrapoff* is used, the integer trap handler is disabled at that procedural level and all levels below it. It is not disabled for any procedural level above the procedure within which *intrapoff* was called. For example,

```
a();
{
          b();        /* Call function b. */
}
b();
{
          intrapoff();
          c();        /* Call function c. */
}
c();
{
          /* Do some work. */
}
```

The integer trap handler is disabled for functions b and c. It is automatically re–enabled on exit from function b. The integer trap handler can also be re–enabled at any time using *intrapon.*

EXAMPLES

The math library routine *rand* generates random integers using:

randx = randx * (((1103515245L + 12345)>>16) & 0x7ffff)

where randx is an unsigned integer. The value assigned to randx is often greater than 2^31–1. To avoid the performance degradation of entering the integer trap handler each time this occurs, the integer trap can be turned off before the assignment using *intrapoff*.

NAME

io_burst – perform low overhead I/O with an HP-IB channel

SYNOPSIS

#include <dvio.h>
io_burst (eid, flag)

HP-UX COMPATIBILITY

Level:	Device I/O Library-HP-UX/EXTENDED NON-STANDARD
Origin:	HP
Remarks:	*Io_burst* is implemented on the Series 200/300 only.

DESCRIPTION

Io_burst enables you to perform low-overhead burst transfers on the specified HP-IB bus. *Eid* is the entity identifier for an open HP-IB bus device file returned by a previous call to *open(2), dup(2), creat(2)* or to *fcntl(2)* with an F_DUPD command option. *Flag* is an integer which, if non-zero, enables burst mode, and otherwise disables it.

In burst mode, memory-mapped I/O address space assigned to the interface card select code is mapped directly into user address space such that the user can transfer data directly to or from the interface card. This eliminates the need for kernel calls and their associated overhead . Burst mode affects only *read(2), write(2), hpib_io(3),* and *hpib_send_cmnd(3)* calls. All other operations are unaffected. When burst mode is enabled, the interface is locked and no other processes are allowed to use the interface until burst mode is disabled.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Timeouts for *read*(2), *write*(2), *hpib_io*(3), and *hpib_send_cmnd*(3) do not work while in burst mode. However, these commands can be interrupted by signals.

RETURN VALUE

Io_burst returns zero if successful or -1 if an error was encountered.

DIAGNOSTICS

Io_burst fails under any of the following circumstances and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB bus device file [ENOTTY];

a timeout occurs [EIO];

WARNING

Enabling burst mode locks the interface and should not be used with any interface supporting a system disc or swap device.

SEE ALSO

read(2), write(2), hpib_io(3I), hpib_send_cmnd(3I)

NAME

io_eol_ctl – set up read termination character on special file

SYNOPSIS

int io_eol_ctl (eid, flag, match);
int eid, flag, match;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Io_eol_ctl enables you to specify a character to be used in terminating a read operation from the specified file id.

Eid is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer which enables or disables character-match termination. A non-zero value enables character-match termination, while a zero value disables it. *Match* is an integer containing the numerical equivalent of the termination character. *Match* is ignored if *flag* is zero. When in 8-bit mode, the lower 8 bits of *match* are used as the termination character. In 16-bit mode, the lower 16 bits are used.

Upon opening a file, the default condition is character-match termination disabled. When enabled, the character specified by *match* is checked for during read operations. The read is terminated upon receipt of this character, or upon any of the other termination conditions normally in effect for this file. Examples of other conditions are satisfying the specified byte count, and receiving a character when the EOI line is asserted (HP-IB). When the read is terminated by a *match* character, this character is the last character returned in the buffer.

File descriptors for the same device file obtained by separate *open*(2) requests have their own termination characters associated with them. File descriptors for the same device file inherited by a *fork*(2) request share the same termination character. In the latter case, if one process changes the termination character, the new termination character is in effect for all such file descriptors.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:

When termination is requested in 16-bit mode, the upper byte of the halfword is examined first, and then the lower byte. If the lower byte matches the termination character, all is as expected. However, if the upper byte matches, the following action is taken:

both the upper and lower bytes are moved into the given buffer, and

the count returned is odd, indicating that there is a lower byte following the matching upper byte. This information is passed to the upper level software to deal with as it pleases.

RETURN VALUE

Io_eol_ctl returns 0 (zero) if successful, or –1 if an error was encountered.

SEE ALSO

io_width_ctl(3I).

DIAGNOSTICS

Io_eol_ctl fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to a channel device file [ENOTTY].

NAME

io_get_term_reason – determine how last read terminated

SYNOPSIS

int io_get_term_reason (eid);
int eid;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Io_get_term_reason returns the termination reason for the last read made on this file descriptor. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

All file descriptors descending from an *open*(2) request (such as from *dup*(2) or *fork*(2)) set this status. For example, if the calling process had opened this file descriptor, and later forked, the status returned would be from the last read done by either the calling process or its child.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:

If the last read had multiple applicable termination reasons, such as having EOI asserted on the last byte when that byte was the termination match character (see *io_eol_ctl*(3)), the highest numbered reason is used (in this case, 4). Since interactive terminals are treated as record-oriented devices when they are in cooked mode, the termination reason is 4 when terminated by a new-line character. If no read has been done on the file descriptor since it was opened, the termination reason is 0.

Series 200/300:

PSTS is checked only at the beginning of a transfer. An interrupt caused by an EIR will also terminate a transfer. The termination reason in this case is also 4.

SEE ALSO

io_eol_ctl(3I).

RETURN VALUE

Io_get_term_reason returns a value indicating how the last read on the specified file descriptor was terminated. This value is interpreted as follows (note that combinations are possible):

Value	Description
–1	An error was encountered while making this function request.
0	Last read encountered some abnormal termination reason not covered by any of the other reasons.
1	Last read terminated by reading the number of bytes requested.
2	Last read terminated by detecting the specified termination character.
4	Last read terminated by detecting some device-imposed termination condition. Examples are: EOI for HP-IB, PSTS line on GPIO, or some other end-of-record condition, such as the physical end-of-record mark on a 9-track tape.

DIAGNOSTICS

Io_get_term_reason fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to a channel device file [ENOTTY].

NAME

io_interrupt_ctl – enable/disable interrupts for the associated eid.

SYNOPSIS

int io_interrupt_ctl (eid, enable_flag)
int eid, enable_flag;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP
Remarks:	*Io_interrupt_ctl* is implemented on the Series 500 only.

DESCRIPTION

Eid is an entity identifier of an open raw HP-IB bus or GPIO device file, obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Enable_flag* is an integer which enables or disables interrupts for the associated *eid.* A non-zero value enables interrupts.

Interrupts may be disabled or enabled by the user as desired. When an interrupt occurs for a given *eid*, the interrupts associated with this *eid* are automatically disabled from reoccurring. Interrupts for this *eid* can be re-enabled by using *io_interrupt_ctl.*

RETURN VALUE

io_interrupt_ctl returns 0 (zero) if successful, or –1 if an error was encountered.

DIAGNOSTICS

Io_interrupt_ctl fails under the following situations and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF]

eid does not refer to a device that supports interrupts [ENOTTY]

no interrupt conditions were specified for this *eid* [EINVAL]

SEE ALSO

io_on_interrupt(3I)

NAME

io_on_interrupt – device interrupt (fault) control

SYNOPSIS

```
#include <dvio.h>
int (*io_on_interrupt (eid, causevec, handler))()
int eid;
struct interrupt_struct *causevec;
int (*handler)();

handler (eid, causevec)
int eid;
struct interrupt_struct *causevec;
```

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP
Remarks:	*Io_on_interrupt* is implemented on the Series 500 only.

DESCRIPTION

Eid is an entity identifier of an open raw HP-IB bus or GPIO device file, obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Causevec is a pointer to a structure of the form:

```
struct interrupt_struct {
                          int cause;
                          int mask;
};
```

The *interrupt_struct* structure is defined in the file **dvio.h**.

The *cause* parameter is a bit vector specifying which of the interrupt or fault events will cause the handler routine to be invoked. The interrupt causes are often specific to the type of interface being considered. As well, certain exception (error) conditions can be handled using the *io_on_interrupt* capability. Specifying a zero-valued *cause* vector effectively turns off the interrupts for that eid.

The *mask* parameter is used when an HP-IB parallel poll interrupt is being defined. *Mask* is an integer that specifies which parallel poll response lines are of interest. *Mask*'s value is obtained from an 8-bit binary number, each bit of which corresponds to one of the eight lines. For example, if you want an interrupt handler invoked for a response on lines 2 or 6, the correct binary number is 01000100. This converts to a decimal equivalent of 68, which is the number you should assign to *mask.*

When an interrupt that is to be caught occurs during a *read*, *write*, *open*, or *ioctl* system call on a slow device (like a terminal; but not a file), during a *pause* system call, during a *sigpause*(2) system call, or during a *wait* system call that does not return immediately due to the existence of a previously stopped or zombie process, the interrupt handling function will be executed and the interrupted system call will return a -1 to the calling process with *errno* set to EINTR.

Interrupt handlers are not inherited across a *fork*(2). All *eid* for the same device file produced by *dup*(2) share the same handler.

An interrupt for a given eid is implicitly disabled after the occurrence of the event. The interrupt condition can be re-enabled by using *io_interrupt_ctl*(3I).

Upon the occurrence of an event specified by *cause*, the receiving process is to execute the interrupt handler function pointed to by *handler.* When the handler returns the user process resumes at the point of execution left when the event occurred.

Handler will be passed two parameters, the *eid* associated with the event, and a pointer to a *causevec* structure. The cause of the interrupt can be determined by the value returned in the *cause* field of the *causevec* structure. If the interrupt handler was invoked due to a parallel poll interrupt, then the *mask* field of the *causevec* structure will contain the parallel poll response byte XOR-ed with the *sense* and AND-ed with the *mask*

HPIB INTERRUPTS

This section describes interrupt causes specific to an HP-IB device. For an HP-IB device the cause is a bit vector which is used as follows: To enable a given event, the appropriate bit (in cause), shown below, must be set to 1:

SRQ	SRQ and active controller.
TLK	Talker addressed.
LTN	Listener addressed.
TCT	Controller in charge.
IFC	IFC has been asserted
REN	Remote enable
DCL	Device clear
GET	Group execution trigger
PPOLL	Parallel poll

GPIO INTERRUPTS

This section describes interrupt causes specific to a gpio device. For a gpio device the cause is a bit vector which is used as follows. To enable a given event, the appropriate bit (in cause), shown below, must be set to 1:

EIR	External interrupt
SIE0	Status line 0
SIE1	Status line 1

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:

The 5.0 HP_UX system does not support parallel-poll interrupts. The internal HP-IB supplied with the Model 550 cannot support talker-addressed, listener-addressed, controller-in-charge, and remote-enable interrupt. GPIO interrupts on the EIR line are not supported.

RETURN VALUE

Io_on_interrupt returns a pointer to the previous handler if the new handler is successfully installed, otherwise it returns a -1 and errno is set.

DIAGNOSTICS

Io_on_interrupt can fail for any of the following reasons:

Fildes does not refer to an open file [EBADF]

Fildes does not refer to a GPIO or a raw HP-IB device file [ENOTTY]

Handler points to an illegal address [EFAULT]

causevec points to an illegal address. [EFAULT]

SEE ALSO

io_interrupt_ctl(3I), pause(2), sigpause(2)

NAME

io_reset – reset an I/O interface

SYNOPSIS

int io_reset (eid);
int eid;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Io_reset resets the interface associated with the device file that was opened. The specific actions performed by *Io_reset* are hardware dependent. *Eid* is an entity identifer of an open raw HP-IB bus or GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Io_reset also causes an interface to go through its self-test, and returns a failure indication if the interface fails its test.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:

When an HP-IB interface is reset, the interface is returned to its power on state if system controller, otherwise, the parallel poll response is set to zero and the interrupt mask is set to zero.

When a GPIO interface is reset, the interfae is returned to its power on state.

Series 200/300:

When an HP-IB interface is reset, the interrupt mask is set to 0, the parallel poll response is set to 0, the serial poll response is set to 0, the HP-IB address is assigned, the IFC line is pulsed (if system controller), the card is put on line, and REN is set (if system controller).

When a GPIO interface is reset, the peripheral reset line is pulled low, the PCTL line is placed in the clear state, and if the DOUT CLEAR jumper is installed, the data out lines are all cleared. The interrupt enable bit is also cleared.

Interface selftest is not supported.

RETURN VALUE

Io_reset returns 0 (zero) if successful, or –1 if an error was encountered.

DIAGNOSTICS

Io_reset fails under any of the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY].

NAME

io_speed_ctl – inform system of required transfer speed

SYNOPSIS

int io_speed_ctl (eid, speed);
int eid, speed;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Io_speed_ctl enables you to select the data transfer speed (within the limits of the hardware) for a data path used for a particular interface. The transfer method (i.e., DMA, fast-handshake) chosen by the system is determined by the speed requirements.

Eid is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Speed* is an integer specifying the data transfer speed in K-bytes per second (one K-byte equals 1024 bytes).

DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:

The Series 500 always provides DMA for the fastest possible I/O speed. Therefore, *io_speed_ctl(3I)* is a nonoperative condition.

Series 200/300:

For values of *speed* less than 7 the system will use an interrupt transfer. For larger values DMA will be used if available otherwise the system will use an interrupt transfer. The default transfer method is DMA.

RETURN VALUE

Io_speed_ctl returns 0 if successful, and –1 otherwise.

DIAGNOSTICS

Io_speed_ctl fails under the following conditions, and sets *errno* to the value enclosed in square brackets:

eid does not refer to an open file [EBADF].

eid does not refer to a supported device file [ENOTTY].

NAME

io_timeout_ctl – establish a time limit for I/O operations

SYNOPSIS

int io_timeout_ctl (eid, time);
int eid;
long time;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Io_timeout_ctl enables you to assign a timeout value to the specified file descriptor. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Time* is a 32-bit integer value specifying the length of the timeout in microseconds.

This timeout applies to future read and write requests on this file descriptor. If a read or write request does not complete within the specified time limit, the request is aborted and returns an error indication. The *errno* value for a timed-out request is EIO, specifying that a timeout has occurred.

Although the timeout value is specified in microseconds, the resolution of the timeout is system-dependent. For example, a particular system might have a resolution of 10 milliseconds, in which case the specified timeout value is rounded up to the next 10 msec boundary. A timeout value of zero means that the system never causes a timeout. When a file is opened, a zero timeout value is assigned by default.

File descriptors for the same device file obtained by separate *open*(2) requests have their own timeout values associated with them. File descriptors for the same device file obtained by *dup*(2) or inherited by a *fork*(2) request share the same timeout value. In the latter case, if one process changes the timeout, the new timeout is in effect for all such file descriptors.

HARDWARE DEPENDENCIES

Series 200/300/500

Eid is an integer file descriptor (fildes) that identifies an open device special file.

Series 500:

The timeout resolution is 10 msec. If an I/O operation is aborted due to a timeout, an *errinfo*(2) value of 56 is returned.

Series 200 and 300:

The default timeout for Series 200/300 GPIO interface is 15 seconds. Timeout resolution is 20 msec.

RETURN VALUE

Io_timeout_ctl returns 0 (zero) if successful, or –1 if an error was encountered.

DIAGNOSTICS

Io_timeout_ctl fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to a channel device file [ENOTTY].

NAME

io_width_ctl – set width of data path

SYNOPSIS

int io_width_ctl (eid, width)
int eid, width;

HP-UX COMPATIBILITY

Level:	Device I/O Library – HP-UX/EXTENDED
Origin:	HP

DESCRIPTION

Io_width_ctl enables you to select the width of the data path to be used for a particular interface. *Eid* is an entity identifier of an open device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Width* is an integer specifying the width of the data path in bits.

The allowable widths are system and hardware dependent. An error is given if an invalid width is specified. Specifying a width with this function sets the width for all users of the device file associated with the given file descriptor. When first opened, the default width is 8 bits.

HARDWARE DEPENDENCIES

Series 200/300/500:

For the GPIO interface only widths of 8 and 16 bits are currently supported. For the HP-IB interface only width of 8 bits is supported.

RETURN VALUE

Io_width_ctl returns 0 if successful, and –1 if an error was encountered.

DIAGNOSTICS

Io_width_ctl fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

eid does not refer to an open file [EBADF];

eid does not refer to an HP-IB raw bus device file [ENOTTY].

the specified *width* is not supported on this device file [EINVAL].

NAME

l3tol, ltol3 - convert between 3–byte integers and long integers

SYNOPSIS

void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;

HP–UX COMPATIBILITY

Level: Bell File System – HP–UX/RUN ONLY

Origin: System V

DESCRIPTION

L3tol converts a list of *n* three–byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

Ltol3 performs the reverse conversion from long integers (*lp*) to three–byte integers (*cp*).

These functions are useful for file–system maintenance where the block numbers are three bytes long.

SEE ALSO

fs(5).

BUGS

Because of possible differences in byte ordering, the numerical values of the long integers are machine–dependent.

NAME

langinfo, langtoid, idtolang, currlangid - information on user's native language as given by NLS

SYNOPSIS

#include <langinfo.h>

char *langinfo(langid, item)
int langid, item;

int langtoid(langname)
char *langname;

char *idtolang(langid)
int langid;

int currlangid()

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

Native Language Support:
8-bit data, customs, messages

DESCRIPTION

Langinfo retrieves a null-terminated string containing information unique to a language or cultural area. For example *langinfo(currlangid(), DAY_1)* returns a pointer to the string "Dom" if LANG (see *environ*(7)) is set to "portuguese", and "sun" if LANG is set to "finnish". The following *items* have been defined.

D_T_FMT	string for formatting *date(1)*
DAY_1	Name of the first day of the week ("Sunday" in English)
. . .	. . .
DAY_7	Name of the seventh day of the week
ABDAY_1	Abbreviated name of the first day of the week ("Sun" in English)
. . .	. . .
ABDAY_7	Abbreviated name of the seventh day of the week
MON_1	Name of the first month in the Gregorian year
. . .	. . .
MON_12	Name of the twelfth month
ABMON_1	Abbreviated name of the first month
. . .	. . .
ABMON_12	Abbreviated name of the twelfth month
RADIXCHAR	radix character ("decimal point" in English)
THOUSEP	separator for thousands
YESSTR	affirmative response for yes/no questions
NOSTR	negative response for yes/no questions
CRNCYSTR	symbol for currency preceded by '–' if it precedes the number, For example, "–f" would be used for Dutch, "+ Kr" for Danish.

Currlangid looks for a LANG string in the user's environment. If it finds it, it returns the corresponding integer listed in *langid*(7). Otherwise it returns 0 to indicate a default to native–computer, the method used before Native Language Support (**NLS**) was available.

Idtolang takes the integer *langid* and attempts to return the corresponding character string defined in *langid*(7). If *langid* is not found, an empty string is returned.

Langtoid is the reverse of *idtolang*, trying to convert a string to a language ID, and returning 0 to indicate native–computer if a match cannot be found.

SEE ALSO

getenv(3C), environ(7), hpnls(7), langid(7).

BUGS

Langinfo returns a pointer to a static area which is overwritten on each call.

NAME

logname - return login name of user

SYNOPSIS

char *logname( )

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Logname returns a pointer to the null-terminated login name; it extracts the **$LOGNAME** variable from the user's environment.

This routine is kept in **/lib/libPW.a**.

FILES

/etc/profile

SEE ALSO

env(1), login(1), profile(5), environ(7).

BUGS

The return values point to static data whose content is overwritten by each call.

This method of determining a login name is subject to forgery.

NAME

lsearch, lfind - linear search and update

SYNOPSIS

char *lsearch ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

char *lfind ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. **Key** points to the datum to be sought in the table. **Base** points to the first element in the table. **Nelp** points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. **Compar** is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non–zero otherwise.

Lfind is the same as *lsearch* except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer–to–element, and cast to type pointer–to–character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer–to–character, the value returned should be cast into type pointer–to–element.

EXAMPLE

This fragment will read in ≤ TABSIZE strings of length ≤ ELSIZE and store them in a table, eliminating duplicates.

```
#include <stdio.h>

#define TABSIZE 50
#define ELSIZE 120

        char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
        unsigned nel = 0;
        int strcmp( );
        . . .
        while (fgets(line, ELSIZE, stdin) != NULL &&
           nel < TABSIZE)
                (void) lsearch(line, (char *)tab, &nel,
                           ELSIZE, strcmp);
        . . .
```

SEE ALSO

bsearch(3C), hsearch(3C), tsearch(3C).

DIAGNOSTICS

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns NULL and *lsearch* returns a pointer to the newly added element.

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

malloc, free, realloc, calloc - main memory allocator

SYNOPSIS

char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

HP–UX COMPATIBILITY

Level: HP–UX/RUN ONLY

Origin: System V

DESCRIPTION

Malloc and *free* provide a simple general–purpose memory allocation package. *Malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, but its contents are left undisturbed.

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

Malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls *sbrk* (see *brk*(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, then *realloc* will ask *malloc* to enlarge the arena by *size* bytes and will then move the data to the new space.

Realloc also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*; thus sequences of *free*, *malloc* and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

SEE ALSO

brk(2), malloc(3X).

DIAGNOSTICS

Malloc, *realloc* and *calloc* return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by *ptr* may be destroyed.

BUGS

Free does not check its pointer argument for validity. When passed a null pointer (value 0), it causes a memory fault.

NOTE

Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer. For an alternate, more flexible implementation, see *malloc*(3X).

NAME

malloc, free, realloc, calloc, mallopt, mallinfo - fast main memory allocator

SYNOPSIS

#include <malloc.h>

char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

int mallopt (cmd, value)
int cmd, value;

struct mallinfo mallinfo (max)

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

DESCRIPTION

Malloc and *free* provide a simple general–purpose memory allocation package, which runs considerably faster than the *malloc*(3C) package. It is found in the library "malloc", and is loaded if the option "-lmalloc" is used with *cc*(1) or *ld*(1).

Malloc returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, and its contents will usually have been destroyed (but see *mallopt* below for a way to change this behavior).

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Mallopt provides for control over the allocation algorithm. The available values for *cmd* are:

M_MXFAST	Set *maxfast* to *value*. The algorithm allocates all blocks below the size of *maxfast* in large groups and then doles them out very quickly. The default value for *maxfast* is 24.
M_NLBLKS	Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *Numlblks* must be greater than 1. The default value for *numlblks* is 100.
M_GRAIN	Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *Grain* must be greater than 0. The default value of *grain* is the smallest number of bytes which will allow alignment of any data type. Value will be rounded up to a multiple of the default when *grain* is set.
M_KEEP	Preserve data in a freed block until the next *malloc*, *realloc*, or *calloc*. This option is provided only for compatibility with the old version of *malloc* and is not

recommended.

These values are defined in the <*malloc.h*> header file.

Mallopt may be called repeatedly, but may not be called after the first small block is allocated.

Mallinfo provides instrumentation describing space usage, but may not be called until the first small block is allocated. The *max* argument to mallinfo should always be specified as 0 for compatibility with other systems. It returns the structure:

```
struct mallinfo {
        int arena;          /* total space in arena */
        int ordblks;        /* number of ordinary blocks */
        int smblks;         /* number of small blocks */
        int hblkhd;         /* space in holding block headers */
        int hblks;          /* number of holding blocks */
        int usmblks;        /* space in small blocks in use */
        int fsmblks;        /* space in free small blocks */
        int uordblks;       /* space in ordinary blocks in use */
        int fordblks;       /* space in free ordinary blocks */
        int keepcost;       /* space penalty if keep option */
                            /* is used */
}
```

This structure is defined in the <*malloc.h*> header file.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

SEE ALSO

brk(2), malloc(3C).

DIAGNOSTICS

Malloc, realloc and *calloc* return a NULL pointer if there is not enough available memory. When *realloc* returns NULL, the block pointed to by *ptr* is left intact. If *mallopt* is called after any allocation of a small block or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

WARNINGS

This package usually uses more data space than *malloc*(3C).
The code size is also bigger than *malloc*(3C).
Note that unlike *malloc*(3C), this package does not preserve the contents of a block when it is freed, unless the M_KEEP option of *mallopt* is used.
Undocumented features of *malloc*(3C) have not been duplicated.

NAME

matherr - error–handling function

SYNOPSIS

#include <math.h>

int matherr (x)
struct exception *x;

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

DESCRIPTION

Matherr is invoked by functions in the Math Library when errors are detected. Users may define their own procedures for handling errors, by including a function named *matherr* in their programs. *Matherr* must be of the form described above. When an error occurs, a pointer to the exception structure *x* will be passed to the user-supplied *matherr* function. This structure, which is defined in the <*math.h*> header file, is as follows:

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
};
```

The element *type* is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

DOMAIN	argument domain error
SING	argument singularity
OVERFLOW	overflow range error
UNDERFLOW	underflow range error
TLOSS	total loss of significance
PLOSS	partial loss of significance

The element *name* points to a string containing the name of the function that incurred the error. The variables *arg1* and *arg2* are the arguments with which the function was invoked. *Retval* is set to the default value that will be returned by the function unless the user's *matherr* sets it to a different value.

If the user's *matherr* function returns non–zero, no error message will be printed, and *errno* will not be set.

If *matherr* is not supplied by the user, the default error–handling procedures, described with the math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, *errno* is set to EDOM or ERANGE and the program continues.

EXAMPLE

```
#include <math.h>

int
matherr(x)
register struct exception *x;
{
        switch (x->type) {
        case DOMAIN:
                /* change sqrt to return sqrt(-arg1), not 0 */
                if (!strcmp(x->name, "sqrt")) {
                        x->retval = sqrt(-x->arg1);
                        return (0); /* print message and set errno */
                }
        case SING:
                /* all other domain or sing errors, print message and abort */
                fprintf(stderr, "domain error in %s\n", x->name);
                abort( );
        case PLOSS:
                /* print detailed error message */
                fprintf(stderr, "loss of significance in %s(%g) = %g\n",
                        x->name, x->arg1, x->retval);
                return (1); /* take no other action */
        }
        return (0); /* all other errors, execute default procedure */
}
```

DEFAULT ERROR HANDLING PROCEDURES

	Types of Errors					
type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
errno	EDOM	EDOM	ERANGE	ERANGE	ERANGE	ERANGE
BESSEL:	-	-	-	-	M, 0	*
y0, y1, yn (arg ≤ 0)	M, -H	-	-	-	-	-
EXP:	-	-	H	0	-	-
LOG, LOG10:						
(arg < 0)	M, -H	-	-	-	-	-
(arg = 0)	-	M, -H	-	-	-	-
POW:	-	-	±H	0	-	-
neg ** non-int	M, 0	-	-	-	-	-
0 ** non-pos						
SQRT:	M, 0	-	-	-	-	-
GAMMA:	-	M, H	H	-	-	-
HYPOT:	-	-	H	-	-	-
SINH:	-	-	±H	-	-	-
COSH:	-	-	H	-	-	-
SIN, COS, TAN:	-	-	-	-	M, 0	*
ASIN, ACOS, ATAN2:	M, 0	-	-	-	-	-

ABBREVIATIONS

*	As much as possible of the value is returned.
M	Message is printed (EDOM error).
H	HUGE is returned.
-H	-HUGE is returned.
±H	HUGE or -HUGE is returned.
0	0 is returned.

NAME

memccpy, memchr, memcmp, memcpy, memset - memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memccpy copies characters from memory area **s2** into **s1**, stopping after the first occurrence of character **c** has been copied, or after **n** characters have been copied, whichever comes first. It returns a pointer to the character after the copy of **c** in **s1**, or a NULL pointer if **c** was not found in the first **n** characters of **s2**.

Memchr returns a pointer to the first occurrence of character **c** in the first **n** characters of memory area s, or a NULL pointer if **c** does not occur.

Memcmp compares its arguments, looking at the first **n** characters only, and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**. (*n* less than or equal to zero yields equality). This routine uses **unsigned char** for character comparison on HP-UX. This may not be true for other implementatitons.

Memcpy copies **n** characters from memory area **s2** to **s1**. It returns **s1**.

Memset sets the first **n** characters in memory area **s** to the value of character **c**. It returns **s**.

NOTE

For user convenience, all these functions are declared in the optional <*memory.h*> header file.

BUGS

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

mktemp - make a unique file name

SYNOPSIS

char *mktemp (template)
char *template;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Mktemp replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of *template*. The string in *template* should look like a file name with six trailing **X**s; *mktemp* will replace the **X**s with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate the name of an existing file. If there are less than 6 **X**s, the letter will be dropped first, and then high order digits of the process ID will be dropped.

RETURN VALUE

Mktemp returns its argument except when it runs out of letters, in which case the result is a pointer to the empty string "".

SEE ALSO

getpid(2).

SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

BUGS

It is possible to run out of letters.

Mktemp does not check to see if the file name part of *template* exceeds the maximum length of a file name.

NAME

monitor - prepare execution profile

SYNOPSIS

#include <mon.h>
void monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)(), (*highpc)();
WORD *buffer;
int bufsize, nfunc;

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

Remarks:

Monitor is implemented on Series 200 only.

DESCRIPTION

An executable program created by **cc −p** automatically includes calls for *monitor* with default parameters; *monitor* need not be called explicitly except to gain fine control over profiling.

Monitor is an interface to *profil*(2). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* WORDs (defined in the <*mon.h*> header file). *Monitor* arranges to record a histogram of periodically sampled program counter values and counts of calls to certain functions in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *Lowpc* must not equal 0 for this use of *monitor*. Not more than *nfunc* call counts can be kept; only calls to functions that were compiled with the **−p** profiling option of *cc*(1) are recorded. For results to be significant, especially where there are small, heavily-used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext();
...
monitor(2, etext, buf, bufsize, nfunc);
```

Etext lies just above all the program text (see *end*(3C)).

To stop execution monitoring and write the results on the file **mon.out**, use

```
monitor(0);
```

Prof(1) can then be used to examine the results.

FILES

mon.out

SEE ALSO

cc(1), prof(1), profil(2).

NAME

nl_toupper, nl_tolower - translate characters for use with NLS

SYNOPSIS

int nl_toupper (c, langid)
int c, langid;

int nl_tolower (c, langid)
int c, langid;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	HP

Native Language Support:
8-bit data, customs, messages

DESCRIPTION

These routines are extensions of their counterparts in *conv(3C)*. They function in the same way, but have a second parameter whose value is expected to be one of the values defined in *langid(7)*. If *langid* is not one of these legal values, or if shift information for *langid* has not been installed, they function as *toupper()* and *tolower()*.

SEE ALSO

conv(3C), ascii(7), hpnls(7), kana8(7), langid(7), roman8(7).

NAME

nl_isalpha, nl_isupper, nl_islower, nl_isalnum, nl_ispunct, nl_isprint, nl_isgraph - classify characters for use with NLS

SYNOPSIS

```
#include <nl_ctype.h>

int nl_isalpha (c, langid)
int c; int langid;

. . .
```

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: HP

Native Language Support:
8-bit data, customs, messages

DESCRIPTION

These routines classify character-coded integer values by table lookup. *Langid* is as defined in *langid(7)*. Each is a predicate returning 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. will be used, returning 0 for values above octal 0200.

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.

DIAGNOSTICS

If the argument to any of these is not in the domain of the function, the result is undefined.

SEE ALSO

ctype(3C), stdio(3S), ascii(7), hpnls(7) kana8(7), roman8(7).

NAME

strcmp8, strncmp8, strcpm16, strncmp16 - non ASCII string collation used by NLS

SYNOPSIS

int strcmp8 (s1, s2, langid, status)
unsigned char *s1, *s2;
int langid,*status;

int strncmp8 (s1, s2, n, langid, status)
unsigned char *s1, *s2;
int n, langid, *status;

int strcmp16 (s1, s2, file_name, status)
unsigned char *s1, *s2, *file_name;
int *status;

int strncmp16 (s1, s2, n, file_name, status)
unsigned char *s1, *s2, *file_name;
int n, *status;

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

Native Language Support:
8-bit and 16-bit data, customs, messages

DESCRIPTION

These functions do not check for overflow of any receiving string.

Strcmp8 compares string *s1* and *s2* according to the collating sequence specified by *langid* (See langid(7)). An integer greater than, equal to, or less than 0 is returned, according as *s1* is greater than, equal to, or less than *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. *Strncmp8* makes the same comparison but looks at most *n* characters.

Strcmp16 compares strings *s1* and *s2* according to the 16-bit collating sequence table in *file_name* (See col_seq_16). Strings *s1* and *s2* may contain 16-bit character substrings in 8-bit canonical form. An integer greater than, equal to, or less than 0, according as *s1* is greater than, equal to, or less than *s2*. *Strncmp16* makes the same comparison but looks at most *n* characters.

The integer pointed to by *status* is set to one of the following non-zero values defined in */usr/include/langinfo.h* if an abnormal condition is encountered.

ENOCFFILE – the file */usr/lib/nls/config* is missing

ENOCONV – the entry for the language sought is not in the file */usr/lib/nls/config*

ENODIR – the directory */usr/lib/nls/$LANG* cannot be accessed

ENOLFILE – the data file */usr/lib/nls/$LANG/collate8* or *file_name* is missing

EBADREAD – the data file */usr/lib/nls/$LANG/collate8* or *file_name* exists but seems to be corrupted

SEE ALSO

col_seq_16(5), col_seq_8(5), hpnls(7), langid(7).

NAME

nlist - get entries from name list

SYNOPSIS

#include <nlist.h>

int nlist (file–name, nl)
char *file–name;
struct nlist *nl;

HP-UX COMPATIBILITY

Level:	HP–UX/STANDARD
Origin:	System V

Remarks:

Nlist is currently implemented on the Series 200 and the Integral PC only.

The use of symbol table type and value information is inherently non–portable. Use of *nlist* should reduce the effort required to port a program which uses such information, but complete portability across all implementations of HP-UX cannot be expected.

DESCRIPTION

Nlist examines the name list in the executable file whose name is pointed to by *file–name*, and selectively extracts a list of values and puts them in the array of nlist structures pointed to by *nl*. The name list *nl* consists of an array of structures initially containing names of variables; once *nlist* has been called, the information is augmented with types and values. The list is terminated with a null name; that is, a null string is in the name position of the structure. Each variable name is looked up in the name list of the file. If the name is found, the type and value information is inserted into the structure. If the name is not found, the type and value fields are set to zero. The structure **nlist** is defined in the include file *nlist.h.* See *a.out*(5) and *nlist*(5) for more discussion of the symbol table structure.

The file must have the organization and symbol table described for an a.out file in *a.out.h*(5). The information is extracted from the symbol table used by the link editor, *ld*(1).

The list of names provided by the caller is in an array of structures, each containing a pointer to a string giving the name for which information is desired. The rest of the structure contains a type field, a value field, and possibly other machine specific information which will be filled in by the nlist call. The list is terminated with a null name pointer. The structure updated by this call is described in *nlist*(5). The name pointer in the nlist structure will always be the first field regardless of how the other fields may change across implementations.

On machines which have such a file, this subroutine is useful for examining the system name list kept in the file **/hp–ux**. In this way programs can obtain system addresses that are up to date.

NOTES

The <*nlist.h*> header file is automatically included by <*a.out.h*> for compatability. However, if the only information needed from <*a.out.h*> is for use of *nlist*, then including <*a.out.h*> is discouraged.

SEE ALSO

a.out(5), nlist(5).

DIAGNOSTICS

All nlist structure fields are set to 0 if the file cannot be found or if it is not a valid object file containing a linker symbol table.

Nlist returns -1 upon error; otherwise it returns 0.

NAME

perror, errno, sys_errlist, sys_nerr - system error messages

SYNOPSIS

void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: System V

Native Language Support:
8-bit data, customs, messages

DESCRIPTION

Perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new-line. *Sys_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

If the user's *LANG* shell variable is set, *perror* also attempts to return a translation of the error message.

HARDWARE DEPENDENCIES

Series 500:

The error indicator *errinfo* is implemented in addition to *errno*, enabling you to obtain a more detailed description of the error. See *errinfo*(2).

Translated messages not accessed.

Series 200:

Translated messages not accessed.

SEE ALSO

errinfo(2), errno(2).

NAME

popen, pclose - initiate pipe I/O to/from a process

SYNOPSIS

#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

DESCRIPTION

The arguments to *popen* are pointers to null–terminated strings containing, respectively, a shell command line and an I/O mode, either **r** for reading or **w** for writing. *Popen* creates a pipe between the calling program and the command to be executed. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is **w**, by writing to the file *stream*; and one can read from the standard output of the command, if the I/O mode is **r**, by reading from the file *stream.*

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter and a type **w** as an output filter.

SEE ALSO

pipe(2), wait(2), fclose(3S), fopen(3S), system(3S).

DIAGNOSTICS

Popen returns a NULL pointer if files or processes cannot be created, or if the shell cannot be accessed.

Pclose returns -1 if *stream* is not associated with a "*popen*ed" command.

BUGS

If the original and "*popen*ed" processes concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. Problems with an output filter may be forestalled by careful buffer flushing, e.g. with *fflush*; see *fclose*(3S).

NAME

printf, fprintf, sprintf - print formatted output

SYNOPSIS

```
#include <stdio.h>

int printf (format [ , arg ] ... )
char *format;

int fprintf (stream, format [ , arg ] ... )
FILE *stream;
char *format;

int sprintf (s, format [ , arg ] ... )
char *s, format;
```

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: System V

DESCRIPTION

Printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places "output", followed by the null character (**\0**), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the **\0** in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *arg*s under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *arg*s. The results are undefined if there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s are simply ignored.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

> Zero or more *flags*, which modify the meaning of the conversion specification.
>
> An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '-', described below, has been given) to the field width. If the field width for an s conversion is preceded by a 0, the string is right adjusted with zero-padding on the left.
>
> A *precision* that gives the minimum number of digits to appear for the **d**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e** and **f** conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (**.**) followed by a decimal digit string; a null digit string is treated as zero.
>
> An optional **l** (ell) specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*, or an optional **h** specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a short integer *arg*. A **l** before any other conversion character is ignored.
>
> A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

-	The result of the conversion will be left-justified within the field.
+	The result of a signed conversion will always begin with a sign (+ or -).
blank	If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.
#	This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x or X** conversion, a non-zero result will have **0x or 0X** prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X	The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
f	The float or double *arg* is converted to decimal notation in the style "[-]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.
e,E	The float or double *arg* is converted in the style "[-]d.ddde±ddd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains exactly three digits.
g,G	The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.
c	The character *arg* is printed.
s	The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (**\0**) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.
%	Print a %; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

SEE ALSO

ecvt(3C), putc(3S), scanf(3S), stdio(3S), and vprintf(3S).

NAME

printmsg, fprintmsg, sprintmsg - print formatted output with numbered arguments

SYNOPSIS

#include <stdio.h>

int printmsg (format [, arg] ...)
char *format;

int fprintmsg (stream, format [, arg] ...)
FILE *stream;
char *format;

int sprintmsg (s, format [, arg] ...)
char *s, *format;

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: HP

Native Language Support:
8-bit data, customs, messages

DESCRIPTION

Printmsg , *fprintmsg* , and *sprintmsg* are derived from their counterparts in *printf(3S)*, with the amplification that the conversion character **%** is replaced by the sequence **%***digit***$** . *Digit* is a decimal digit *n* in the range 1–9, and indicates that this conversion should be applied to the *n*th argument, rather than to the next unused one. All other aspects of formatting are unchanged. All conversion specifications must contain the **%***digit***$** sequence, and it is the user's responsibility to make sure the numbering is correct. All parameters must be used exactly once.

EXAMPLE

To create a language independent date and time printing routine we would write

```
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"
```

and for German usage to a string

```
"%1$s, %3$d %2$s %4$d:%5$.2d"
```

the resulting outputs will be "Sunday, July 3, 10:02", and "Sonntag, 3 Juli 10:02" , assuming that the proper strings have been passed in.

SEE ALSO

getmsg(3C), printf(3S), hpnls(7).

NAME

putc, putchar, fputc, putw - put character or word on a stream

SYNOPSIS

#include <stdio.h>

int putc (c, stream)
int c;
FILE *stream;

int putchar (c)
int c;

int fputc (c, stream)
int c;
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Putc writes the character *c* onto the output *stream* (at the position where the file pointer, if defined, is pointing). *Putchar*(*c*) is defined as *putc*(*c, stdout*). *Putc* and *putchar* are macros.

Fputc behaves like *putc*, but is a genuine function rather than a macro; it may therefore be used as an argument. *Fputc* runs more slowly than *putc*, but it takes less space per invocation and its name can be passed as an argument to a function.

Putw writes the word (i.e., **int** in C) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. *Putw* neither assumes nor causes special alignment in the file.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line–buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of *freopen* (see *fopen*(3S)) will cause it to become buffered or line–buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line–buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new–line character is written or terminal input is requested). *Fflush* can also be used to explicitly write the buffer. *Setbuf*(3S) or *setvbuf*(3S) may be used to change the stream's buffering strategy.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fwrite(3S), getc(3S), fread(3S), printf(3S), puts(3S), setbuf(3S).

DIAGNOSTICS

On success, these functions each return the value they have written. On failure, they return the constant **EOF**. This will occur if the file *stream* is not open for writing or if the output file cannot be grown. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

Line buffering may cause confusion or malfunctioning of programs which use standard I/O routines but use read(2) themselves to read from standard input. In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to fflush(3) the standard output before going off and computing so that the output will appear.

BUGS

Because it is implemented as a macro, *putc* treats incorrectly a *stream* argument with side effects.

In particular, **putc(c, *f++);** doesn't work sensibly. *Fputc* should be used instead.
Because of possible differences in word length and byte ordering, files written using *putw* are machine–dependent, and may not be read using *getw* on a different (non–HP–UX) processor. For this reason *putw* should be used with care.

NAME

putenv - change or add value to environment

SYNOPSIS

int putenv (string)
char *string;

HP–UX COMPATIBILITY

Level: HP–UX/RUN ONLY

Origin: System V

DESCRIPTION

String points to a string of the form *"name=value." Putenv* makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment. The space used by *string* is no longer used once a new string–defining *name* is passed to *putenv*.

DIAGNOSTICS

Putenv returns non–zero if it was unable to obtain enough space via *malloc* for an expanded environment, otherwise zero.

SEE ALSO

exec(2), getenv(3C), malloc(3C), environ(5).

WARNINGS

Putenv manipulates the environment pointed to by *environ*, and can be used in conjunction with *getenv*. However, *envp* (the third argument to *main*) is not changed.
This routine uses *malloc*(3C) to enlarge the environment.
After *putenv* is called, environmental variables are not in alphabetical order.
A potential error is to call *putenv* with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

NAME

putpwent - write password file entry

SYNOPSIS

#include <pwd.h>
#include <stdio.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;

HP-UX COMPATIBILITY

Level:	HP-UX/NUCLEUS
Origin:	System V

DESCRIPTION

Putpwent is the inverse of *getpwent*(3C). Given a pointer to a *passwd* structure as created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwent* writes a line on the stream *f*, which matches the format of **/etc/passwd**.

DIAGNOSTICS

Putpwent returns non-zero if an error was detected during its operation, otherwise zero.

SEE ALSO

getpwent(3C).

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

NAME

puts, fputs - put a string on a stream

SYNOPSIS

#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Puts writes the null-terminated string pointed to by *s*, followed by a new-line character, to the standard output stream *stdout.*

Fputs writes the null-terminated string pointed to by *s* to the named output *stream.*

Neither function writes the terminating null character. Note that *puts* appends a new-line character, but *fputs* does not.

DIAGNOSTICS

Both routines return **EOF** on error. This will happen if the routines try to write on a file that has not been opened for writing.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S).

NOTES

Puts appends a new-line character while *fputs* does not.

NAME

qsort - quicker sort

SYNOPSIS

void qsort ((char *) base, nel, sizeof (*base), compar)
unsigned nel;
int (*compar)( );

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. It sorts vectors of arbitrarily-sized elements based on user-supplied size information and a comparison routine, in place

Base points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function passed as *compar* must return an integer less than, equal to, or greater than zero as a consequence of whether its first argument is to be considered less than, equal to, or greater than the second. This is the same return convention that *strcmp* uses.

NOTES

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
The order in the output of two items which compare as equal is unpredictable.

SEE ALSO

sort(1), bsearch(3C), lsearch(3C), string(3C).

BUGS

If *width* is zero, a divide-by-zero error is generated.

NAME

rand, srand - simple random–number generator

SYNOPSIS

int rand ( )

void srand (seed)
unsigned seed;

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

DESCRIPTION

Rand uses a multiplicative congruential random–number generator with period 2^{32} that returns successive pseudo–random numbers in the range from 0 to 2^{15}-1.

Srand can be called at any time to reset the random–number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTE

The spectral properties of *rand* leave a great deal to be desired. *Drand48*(3C) provides a much better, though more elaborate, random–number generator.

SEE ALSO

drand48(3C).

NAME

regcmp, regex - compile and execute regular expression

SYNOPSIS

char *regcmp (string1 [, string2, ...], (char *)0)
char *string1, *string2, ...;

char *regex (re, subject[, ret0, ...])
char *re, *subject, *ret0, ...;

extern char *__loc1;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Regcmp compiles a regular expression and returns a pointer to the compiled form. *Malloc*(3C) is used to create space for the vector. It is the user's responsibility to free unneeded space so allocated. A NULL return from *regcmp* indicates an incorrect argument. *Regcmp*(1) has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *Regex* returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer *__loc1* points to where the match began. *Regcmp* and *regex* were mostly borrowed from the editor, *ed*(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[ ] * . ^ These symbols retain their current meaning.

$ Matches the end of the string; **\n** matches a new-line.

- Within brackets the minus means *through*. For example, [**a-z**] is equivalent to [**abcd...xyz**]. The - can appear as itself only if used as the first or last character. For example, the character class expression []-] matches the characters] and -.

+ A regular expression followed by + means *one or more times*. For example, **[0-9]+** is equivalent to **[0-9][0-9]***.

{m} {m,} {m,u}
Integer values enclosed in { } indicate the number of times the preceding regular expression is to be applied. The value *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. The value {m,} is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.

(...)$*n* The value of the enclosed regular expression is to be returned. The value will be stored in the *(n+1)*th argument following the subject argument. At most ten enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.

(...) Parentheses are used for grouping. An operator, e.g., *, +, { }, can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped to be used as themselves.

EXAMPLES

Example 1:

```
char *cursor, *newcursor, *ptr;
        ...
newcursor = regex((ptr = regcmp("^\n", 0)), cursor);
```

```
        free(ptr);
```

This example will match a leading new–line in the subject string pointed at by cursor.

Example 2:

```
        char ret0[9];
        char *newcursor, *name;
                ...
        name = regcmp("([A-Za-z][A-za-z0-9_]{0,7})$0", 0);
        newcursor = regex(name, "123Testing321", ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (cursor+11). The string "Testing3" will be copied to the character array *ret0*.

Example 3:

```
        #include "file.i"
        char *string, *newcursor;
                ...
        newcursor = regex(name, string);
```

This example applies a precompiled regular expression in **file.i** (see *regcmp*(1)) against *string*.

This routine is kept in **/lib/libPW.a.**

SEE ALSO

ed(1), regcmp(1), malloc(3C).

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required. The following user–supplied replacement for *malloc*(3C) reuses the same vector saving time and space:

```
        /* user's program */
                ...
        char *
        malloc(n)
        unsigned n;
        {
                static char rebuf[512];
                return (n <= sizeof rebuf) ? rebuf : NULL;
        }
```

NAME

scanf, fscanf, sscanf - formatted input conversion, read from stream file

SYNOPSIS

#include <stdio.h>

int scanf (format [, pointer] ... **)**
char *format;

int fscanf (stream, format [, pointer] ... **)**
FILE *stream;
char *format;

int sscanf (s, format [, pointer] ... **)**
char *s, *format;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Scanf reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional **l** (ell) or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except "[" and "c", white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

%	a single % is expected in the input at this point; no assignment is done.
d	a decimal integer is expected; the corresponding argument should be an integer pointer.
u	an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.
o	an octal integer is expected; the corresponding argument should be an integer pointer.
x	a hexadecimal integer is expected; the corresponding argument should be an integer pointer.
e,f,g	a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optional +, -, or space, followed by an integer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which will be added automatically. The input field is terminated by a white-space character. Note that *scanf* will not read a null string.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset,* and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (ˆ), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first-last*, thus [0123456789] may be expressed [0-9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating **\0**, which will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d**, **u**, **o**, and **x** may be preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list. The **l** or **h** modifier is ignored for other conversion characters.

Scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, **EOF** is returned.

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E−1 thompson
```

will assign to *n* the value **3**, to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0**. Or:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0-9]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

SEE ALSO

getc(3S), printf(3S), strtod(3C), strtol(3C).

NOTE

Trailing white space (including a new–line) is left unread unless matched in the control string.

DIAGNOSTICS

These functions return **EOF** on end of input and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

NAME

setbuf, setvbuf - assign buffering to a stream file

SYNOPSIS

#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Setbuf may be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer input/output will be completely unbuffered.

A constant **BUFSIZ,** defined in the **<stdio.h>** header file, tells how big an array is needed:

char buf[BUFSIZ];

Setvbuf may be used after a stream has been opened but before it is read or written. *Type* determines how *stream* will be buffered. Legal values for *type* (defined in stdio.h) are:

_IOFBF	causes input/output to be fully buffered.
_IOLBF	causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.
_IONBF	causes input/output to be completely unbuffered.

If *buf* is not the **NULL** pointer, the array it points to will be used for buffering instead of an automatically allocated buffer (from *malloc*(3C) or *memallc*(2)). *Size* specifies the size of the buffer to be used. The constant **BUFSIZ** in **<stdio.h>** is suggested as a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

By default, output to a terminal is line buffered and all other input/output is fully buffered.

HARDWARE DEPENDENCIES

Series 500:
The system call *memallc*(2) is used instead of *malloc.*

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S), stdio(3S).

DIAGNOSTICS

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non–zero value. Otherwise, the value returned will be zero.

NOTE

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;

int _setjmp(env)
jmp_buf env;

void _longjmp(env, val)
jmp_buf env;
int val;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* (whose type, *jmp_buf*, is defined in the *<setjmp.h>* header file) for later use by *longjmp*. It returns the value 0.

Longjmp restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*. *Longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. All accessible data have values as of the time *longjmp* was called.

Upon the return from a *setjmp* call caused by a *longjmp*, the values of any non-static local variables are undefined. Depending on such values renders code using non-static local variables non-portable.

Setjmp and *longjmp* save and restore the signal mask (see *sigvector*(2)), while *_setjmp* and *_longjmp* manipulate only the stack and registers. This distinction is only significant for programs which use *sigvector(2), sigblock(2),* and/or *sigsetmask(2).*

If a *longjmp* is executed and the environment in which the *setjmp* was executed no longer exists, errors can occur. The conditions under which the environment of the *setjmp* no longer exists include: exiting the procedure which contains the *setjmp* call, and exiting an inner block with temporary storage (e.g. a block with declarations in C, a *with* statement in Pascal). This condition may or may not be detectable. An attempt is made by determining if the stack frame pointer in *env* points to a location not in the currently active stack. If this is the case, *longjmp* will return a -1. Otherwise, the *longjmp* will occur, and if the environment no longer exists, the contents of the temporary storage of an inner block are unpredictable. This condition may also cause unexpected process termination. If the procedure has been exited the results are unpredictable.

Passing *longjmp* a pointer to a buffer not created by *setjmp*, or a buffer that has been modified by the user, can cause all the problems listed above, and more.

Some implementations of Pascal support a *try/recover* mechanism, which also creates stack marker information. If a *longjmp* operation occurs in a scope which is nested inside a try/recover, and the corresponding *setjmp* is not inside the scope of the try/recover, the recover block will not be executed and the currently active recover block will become the one enclosing the *setjmp* (if there is one).

NOTE

A call to *longjmp* to leave the guaranteed stack space reserved by *sigspace (2)* may remove the guarantee that the ordinary execution of the program will not extend into the guaranteed space. It may also cause the program to forever loose its ability to automatically increase the stack size, and the program may then be limited to the guaranteed space.

SEE ALSO

sigvector(2), sigblock(2), sigsetmask(2), sigspace(2), signal(2).

WARNING

If *longjmp* is called even though *env* was never primed by a call to *setjmp*, or when the last such call was in a function which has since returned, absolute chaos is guaranteed.

NAME

sinh, cosh, tanh - hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh (x)              float fsinh (x)
double x;                   ‡float x;

double cosh (x)              float fcosh (x)
double x;                   ‡float x;

double tanh (x)              float ftanh (x)
double x;                   ‡float x;

                                  ‡ see important note below
```

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: System V

DESCRIPTION

Sinh, *cosh*, and *tanh* return, respectively, the hyperbolic sine, cosine and tangent of their argument. These are double-precision routines.

IMPORTANT NOTE: The corresponding single-precision routines *fsinh*, *fcosh*, and *ftanh* expect true single-precision arguments, and therefore cannot be called from standard C. They are provided for support of FORTRAN (Pascal does not support or use hyperbolic functions).

DIAGNOSTICS

Sinh and *cosh* set *errno* to **ERANGE** and return **HUGE** (*sinh* may return **-HUGE** for negative *x*) when the correct value would overflow.

Error handling is identical for both single- and double-precision routines, except for one consideration: In any situation where the double-precision routine would return ±HUGE, the corresponding single-precision routine returns ±MAXFLOAT.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

sleep - suspend execution for interval

SYNOPSIS

unsigned long sleep (seconds)
unsigned long seconds;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred. The caller's alarm catch routine is executed just before the *sleep* routine returns. If the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

Seconds must be less than 2^{32}.

SEE ALSO

alarm(2), pause(2), signal(2).

NAME

sputl, sgetl - access long integer data in a machine–independent fashion.

SYNOPSIS

void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

DESCRIPTION

Sputl takes the four bytes of the long integer *value* and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

Sgetl retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of *sputl* and *sgetl* provides a machine–independent way of storing long numeric data in a file in binary form without conversion to characters.

A program which uses these functions must be loaded with the object–file access routine library **libld.a**.

NAME

ssignal, gsignal - software signals

SYNOPSIS

#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;

HP-UX COMPATIBILITY

Level:	HP-UX/STANDARD
Origin:	System V

DESCRIPTION

Ssignal and *gsignal* implement a software facility similar to *signal*(2). This facility is used by the Standard C Library to enable users to indicate the disposition of error conditions, and is also made available to users for their own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. A call to *ssignal* associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to *gsignal*. Raising a software signal causes the action established for that signal to be *taken*.

The first argument to *ssignal* is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user-defined) *action function* or one of the manifest constants **SIG_DFL** (default) or **SIG_IGN** (ignore). *Ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns **SIG_DFL**.

Gsignal raises the signal identified by its argument, *sig*:

> If an action function has been established for *sig*, then that action is reset to **SIG_DFL** and the action function is entered with argument *sig*. *Gsignal* returns the value returned to it by the action function.
>
> If the action for *sig* is **SIG_IGN**, *gsignal* returns the value 1 and takes no other action.
>
> If the action for *sig* is **SIG_DFL**, *gsignal* returns the value 0 and takes no other action.
>
> If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

SEE ALSO

signal(2).

NOTES

There are some additional signals with numbers outside the range 1 through 15 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the Standard C Library.

NAME

stdio - standard buffered input/output stream file package

SYNOPSIS

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar* and *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use or act as if they use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE.** *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the <stdio.h> header file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant **NULL** (0) designates a nonexistent pointer.

An integer-constant **EOF** (-1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

An integer constant **BUFSIZ** specifies the size of the buffers used by the particular implementation.

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

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

The functions and constants mentioned in the entries of sub-class 3S of this manual are declared in that header file and need no further declaration. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *ferror*, *feof*, *clearerr*, and *fileno*.

A constant **_NFILE** defines the maximum number of open files allowed per process.

SEE ALSO

open(2), close(2), lseek(2), pipe(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpfile(3S), tmpnam(3S), ungetc(3S).

DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

NAME

ftok - standard interprocess communication package

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(path, id)
char *path;
char id;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the *msgget*(2), *semget*(2), and *shmget*(2) system calls to obtain interprocess communication identifiers. One suggested method for forming a key is to use the *ftok* subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If some standard is not adhered to, it will be possible for unrelated processes to unintentionally interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

Ftok returns a key based on *path* and *id* that is usable in subsequent *msgget*, *semget*, and *shmget* system calls. *Path* must be the path name of an existing file that is accessible to the process. *Id* is a character which uniquely identifies a project. Note that *ftok* will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same file name but different *ids*.

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2).

DIAGNOSTICS

Ftok returns **(key_t) -1** if *path* does not exist or if it is not accessible to the process.

WARNING

If the file whose *path* is passed to *ftok* is removed when keys still refer to the file, future calls to *ftok* with the same *path* and *id* will return an error. If the same file is recreated, then *ftok* is likely to return a different key than it did the original time it was called.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok - character string operations

SYNOPSIS

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s;
int c;

char *strrchr (s, c)
char *s;
int c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

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

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

These functions operate on null-terminated strings. The arguments **s1, s2** and **s** point to strings (arrays of characters terminated by a null character). The functions *strcat*, *strncat*, *strcpy*, and *strncpy* all alter **s1**. These functions do not check for overflow of the array pointed to by **s1**.

Strcat appends a copy of string **s2** to the end of string **s1**. *Strncat* appends at most **n** characters. It copies less if *s2* is shorter than *n* characters. Each returns a pointer to the null-terminated result (the original value of *s1*).

Strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**. (**NULL** values for *s1* and *s2* are treated the same as pointers to null strings.) *Strncmp* makes the same comparison but looks at at most **n** characters (*n* less than or equal to zero yields equality). Both of these routines use **unsigned char** for character comparison.

Strcpy copies string **s2** to **s1**, stopping after the null character has been copied. *Strncpy* copies exactly **n** characters, truncating **s2** or adding null characters to **s1** if necessary. The result will not be null–terminated if the length of **s2** is **n** or more. If the lengthe of **s2** is less than **n**, characters from the first null in **s2** to the **n**th character are copied as nulls. Each function returns **s1**.

Note that *strncpy* should not be used to copy *n* bytes of an arbitrary structure. If that structure contains a null byte anywhere, *strncpy* will terminate the copy when it encounters the null byte, thus copying fewer than *n* bytes.

Strlen returns the number of characters in **s**, not including the terminating null character.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character **c** (an 8–bit ASCII value) in string **s**, or a NULL pointer if **c** does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a NULL pointer if no character from **s2** exists in **s1**.

Strspn (*strcspn*) returns the length of the initial segment of string **s1** which consists entirely of characters from (not from) string **s2**.

Strtok considers the string **s1** to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string **s2**. The first call (with pointer **s1** specified) returns a pointer to the first character of the first token, and will have written a null character into **s1** immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string **s1** immediately following that token. In this way subsequent calls will work through the string **s1** until no tokens remain. The separator string **s2** may be different from call to call. When no token remains in **s1**, a NULL pointer is returned.

HARDWARE DEPENDENCIES

Series 200:

N is limited by the process size.

Series 500:

N is limited to about 500 Mbytes.

NOTE

For user convenience, all these functions are declared in the optional <*string.h*> header file.

BUGS

The copy operations cannot check for overflow of any receiving string. **NULL** destinations cause errors; **NULL** sources are treated as zero–length strings.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

strtod, atof, nl_strtod, nl_atof - convert string to double–precision number

SYNOPSIS

double strtod (str, ptr)
char *str, **ptr;

double atof (str)
char *str;

double nl_strtod (str, ptr, langid)
char *str, **ptr;
int langid;

double nl_atof (str, langid)
char *str;
int langid;

HP–UX COMPATIBILITY

Level:	HP–UX/RUN ONLY
Origin:	System V

Native Language Support:
8-bit data, customs, messages

DESCRIPTION

Strtod returns as a double–precision floating–point number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

Strtod recognizes an optional string of "white–space" characters (as defined by *isspace* in *ctype*(3C)), then an optional sign, then a string of digits optionally containing a decimal point then an optional **e** or **E** followed by an optional sign or space, followed by an integer.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, **ptr* is set to *str*, and zero is returned.

Atof(str) is equivalent to *strtod(str, (char **)NULL)*.

NL_strtod and *nl_atof* are similar to the above routines, but use *langid* to determine what the radix character should be (e.g. '.' or ','). If *langid* is not valid, or information for *langid* has not been installed, the radix character defaults to a period.

SEE ALSO

ctype(3C), scanf(3S), strtol(3C), hpnls(7), langid(7).

DIAGNOSTICS

If the correct value would cause overflow, ±**HUGE** is returned (according to the sign of the value), and *errno* is set to **ERANGE**.
If the correct value would cause underflow, zero is returned and *errno* is set to **ERANGE**.

NAME

strtol, atol, atoi - convert string to integer

SYNOPSIS

long strtol (str, ptr, base)
char *str, **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Strtol returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters (as defined by *isspace* in *ctype*(3C)) are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base thusly: After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Atol(str) is equivalent to *strtol(str, (char **)NULL, 10)*.

Atoi(str) is equivalent to *(int) strtol(str, (char **)NULL, 10)*.

SEE ALSO

atof(3C), ctype(3C), scanf(3S), strtod(3C).

HARDWARE DEPENDENCIES

Series 200/500:
Atoi and *atol* are identical.

BUGS

Overflow conditions are ignored.

NAME

swab - swap bytes

SYNOPSIS

```
void swab (from, to, nbytes)
char *from, *to;
int nbytes;
```

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between byte-swapped and non-byte-swapped machines. *Nbytes* should be even and non-negative. If *nbytes* is odd and positive *swab* uses *nbytes*-1 instead. If *nbytes* is negative, *swab* does nothing.

NAME

system - issue a shell command

SYNOPSIS

#include <stdio.h>

int system (string)
char *string;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

System causes the *string* to be given to *sh*(1) as input, as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

FILES

/bin/sh

SEE ALSO

sh(1), exec(2).

DIAGNOSTICS

System forks to create a child process that in turn exec's **/bin/sh** in order to execute *string.* If the fork or exec fails, *system* returns a negative value and sets *errno*.

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs - emulate /etc/termcap access routines

SYNOPSIS

```
tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Termcap(3) functions extract and use capabilities from the compiled terminal capability data bases (see *terminfo(5)).* They are emulation routines that are provided as a part of the *curses(3X)* library.

Tgetent extracts the compiled entry for terminal *name* into buffers accessible by the programmer. Unlike previous termcap routines, all capability strings (except cursor addressing and padding information) are already compiled and stored internally upon return from *tgetent.* The buffer pointer *bp* is redundant in the emulation, and is ignored. It should not be relied upon to point to meaningful information. *Tgetent* returns -1 if it cannot access the *terminfo* directory, 0 if there is no capability file for *name,* and 1 if all goes well. If a *TERMINFO* environment variable is set, *tgetent* first looks for *TERMINFO/?/name* (where ? is the first character of *name*), and if that file is not accessible, it looks for */usr/lib/terminfo/?/name.*

Tgetnum gets the numeric value of capability *id,* returning -1 if it is not given for the terminal. *Tgetnum* is useful only with capabilities having numeric values.

Tgetflag returns 1 if the specified capability is present in the terminal's entry, and 0 if it is not. *Tgetflag* is useful only with capabilities that are boolean in nature (i.e. either present or missing in *terminfo*(5)).

Tgetstr returns a pointer to the string value of capability *id.* In addition, if *area* is not a NULL pointer, *tgetstr* will place the capability in the buffer at *area* and advance the area pointer. The returned string capability is compiled except for cursor addressing and padding information. *Tgetstr* is useful only with capabilities having string values. *Tgetstr* returns a NULL pointer if the capability is not available on the terminal or *id* is not a string capability.

Tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline.* (Programs which call *tgoto* should be sure to turn off the TAB3 bit(s), since *tgoto* may now output a tab. See *termio*(4). Note that programs using *termcap* should in general turn off TAB3 anyway since some terminals use control-I for other functions, such as nondestructive space.) If a

% sequence is given which is not understood, then *tgoto* returns OOPS.

Tputs decodes the padding information of the string *cp. Affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable. *Outc* is a routine which is called with each character in turn. The *terminfo* variable **pad_char** should contain a pad character to be used (from the **pc** capability) if a null (^@) is inappropriate.

FILES

/usr/lib/libcurses.a	-lcurses library
/usr/lib/terminfo/?/*	data bases

SEE ALSO

ex(1), termio(4), terminfo(5).

NAME

tmpfile - create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile ()

HP-UX COMPATIBILITY

Level:	HP-UX/NUCLEUS
Origin:	System V

DESCRIPTION

Tmpfile creates a temporary file using a name generated by *tmpnam*(3S), and returns a corresponding FILE pointer. If the file cannot be opened, an error message is printed using *perror*(3C), and a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update ("w+").

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C), perror(3C), tmpnam(3S).

NAME

tmpnam, tempnam - create a name for a temporary file

SYNOPSIS

#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System V

DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

Tmpnam always generates a file name using the path–prefix defined as **P_tmpdir** in the *<stdio.h>* header file unless *<stdio.h>* has been locally modified. If *s* is NULL, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least **L_tmpnam** bytes, where **L_tmpnam** is a constant defined in *<stdio.h>*; *tmpnam* places its result in that array and returns *s*.

Tempnam allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is NULL or points to a string which is not a name for an appropriate directory, the path–prefix defined as **P_tmpdir** in the *<stdio.h>* header file is used. If that directory is not accessible, **/tmp** will be used as a last resort. This entire sequence can be up–staged by providing an environment variable **TMPDIR** in the user's environment, whose value is the name of the desired temporary–file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary–file name.

Tempnam uses *malloc*(3C) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* (see *malloc*(3C)). If *tempnam* cannot return the expected result for any reason, i.e. *malloc*(3C) failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

NOTES

These functions generate a different file name each time they are called.

Files created using these functions and either *fopen*(3S) or *creat*(2) are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink*(2) to remove the file when its use is ended.

File names are initially of the form [a–z][a–z][a–z]XXXXXX in the directory specified by either *dir* (in *tempnam*) or **L_tmpnam** (in *tmpnam)* and are then passed to *mktemp* before returning the result.

SEE ALSO

creat(2), unlink(2), fopen(3S), malloc(3C), mktemp(3C), tmpfile(3S).

BUGS

If called more than 17,576 times in a single process, these functions will start recycling previously used names.
Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these

functions or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

NAME

sin, cos, tan, asin, acos, atan, atan2 - trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin (x)              float fsin (x)
double x;                  ‡float x;

double cos (x)              float fcos (x)
double x;                  ‡float x;

double tan (x)              float ftan (x)
double x;                  ‡float x;

double asin (x)             float fasin (x)
double x;                  ‡float x;

double acos (x)             float facos (x)
double x;                  ‡float x;

double atan (x)             float fatan (x)
double x;                  ‡float x;

double atan2 (y, x)         float fatan2 (y, x)
double y, x;               ‡float y, x;

                                ‡ see important note below
```

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: System V

DESCRIPTION

Sin, *cos* and *tan* return respectively the sine, cosine and tangent of their argument, *x*, measured in radians.

Asin returns the arcsine of *x*, in the range -π/2 to π/2.

Acos returns the arccosine of *x*, in the range 0 to π.

Atan returns the arctangent of *x*, in the range -π/2 to π/2.

Atan2 returns the arctangent of *y/x*, in the range -π to π, using the signs of both arguments to determine the quadrant of the return value.

IMPORTANT NOTE: The corresponding single–precision routines *fsin*, *fcos*, *ftan*, *fasin*, *facos*, *fatan*, and *fatan2* expect true single–precision arguments, and therefore cannot be called from standard C. They are provided for support of FORTRAN and Pascal.

HARDWARE DEPENDENCIES

Series 200/500:

The approximate limit for the values passed to these functions is 2.98E8 for *sin* and *cos*, 1.49E8 for *tan*, 1.29E4 for *fsin* and *fcos*, and 6.43E3 for *ftan*.

The algorithms used for all functions except *atan2* and *fatan2* are from HP 9000 BASIC.

DIAGNOSTICS

Sin, *cos*, and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return zero when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. *errno* is set to **ERANGE.**

If the magnitude of the argument of *asin* or *acos* is greater than one, or if both arguments of *atan2* are zero, zero is returned and *errno* is set to **EDOM**. In addition, a message indicating

DOMAIN error is printed on the standard error output.

Error handling is identical for both single– and double–precision routines, except for one consideration: In any situation where the double–precision routine would return ±HUGE, the corresponding single–precision routine returns ±MAXFLOAT.

These error–handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

tsearch, tfind, tdelete, twalk - manage binary search trees

SYNOPSIS

#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tfind ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Tsearch, tfind, tdelete, and *twalk* are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine, *compar*. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Tsearch is used to build and access the tree. **Key** is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. **Rootp** points to a variable that points to the root of the tree. A NULL value for the variable pointed to by **rootp** denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like *tsearch*, *tfind* will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* will return a NULL pointer. The arguments for *tfind* are the same as for *tsearch.*

Tdelete deletes a node from a binary search tree. The arguments are the same as for *tsearch.* The variable pointed to by **rootp** will be changed if the deleted node was the root of the tree. *Tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

Twalk traverses a binary search tree. **Root** is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *Action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum { preorder, postorder, endorder, leaf } VISIT;* (defined in the *<search.h>* header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

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

struct node {           /* pointers to these are stored in the tree */
        char *string;
        int length;
};
char string_space[10000];       /* space to store strings */
struct node nodes[500];         /* nodes to store */
struct node *root = NULL;       /* this points to the root */

main( )
{
        char *strptr = string_space;
        struct node *nodeptr = nodes;
        void print_node( ), twalk( );
        int i = 0, node_compare( );

        while (gets(strptr) != NULL && i++ < 500)  {
                /* set node */
                nodeptr->string = strptr;
                nodeptr->length = strlen(strptr);
                /* put node into the tree */
                (void) tsearch((char *)nodeptr, &root,
                          node_compare);
                /* adjust pointers, so we don't overwrite tree */
                strptr += nodeptr->length + 1;
                nodeptr++;
        }
        twalk(root, print_node);
}
/*
        This routine compares two nodes, based on an
        alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
struct node *node1, *node2;
{
        return strcmp(node1->string, node2->string);
}
/*
        This routine prints out a node, the first time
        twalk encounters it.
*/
```

```
void
print_node(node, order, level)
struct node **node;
VISIT order;
int level;
{
        if (order == preorder || order == leaf)  {
                (void)printf("string = %20s,  length = %d\n",
                        (*node)->string, (*node)->length);
        }
}
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.
A NULL pointer is returned by *tsearch, tfind* and *tdelete* if **rootp** is NULL on entry.
If the datum is found, both *tsearch* and *tfind* return a pointer to it. If not, *tfind* returns NULL, and *tsearch* returns a pointer to the inserted item.

WARNINGS

The **root** argument to *twalk* is one level of indirection less than the **rootp** arguments to *tsearch* and *tdelete*.
There are two nomenclatures used to refer to the order in which tree nodes are visited. *Tsearch* uses preorder, postorder and endorder to respectively refer to visting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

BUGS

If the calling function alters the pointer to the root, results are unpredictable.

NAME

ttyname, isatty - find name of a terminal

SYNOPSIS

char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Ttyname returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*.

Isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

FILES

/dev/*, /dev/*/*

DIAGNOSTICS

Ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory **/dev**.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ttyslot - find the slot in the utmp file of the current user

SYNOPSIS

int ttyslot ( )

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by actually scanning the file **/etc/inittab** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

FILES

/etc/inittab
/etc/utmp

SEE ALSO

getut(3C), ttyname(3C).

DIAGNOSTICS

A value of 0 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

NAME

ungetc - push character back into input stream

SYNOPSIS

#include <stdio.h>

int ungetc (c, stream)
int c;
FILE *stream;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

Ungetc inserts the character *c* into the buffer associated with an input *stream*. That character, *c*, will be returned by the next *getc(3S)* call on that *stream*. *Ungetc* returns *c*, and leaves the file *stream* unchanged.

One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered. In the case that *stream* is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

If *c* equals **EOF**, *ungetc* does nothing to the buffer and returns **EOF**.

Fseek(3S) erases all memory of inserted characters.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S).

DIAGNOSTICS

Ungetc returns **EOF** if it cannot insert the character.

NAME

vprintf, vfprintf, vsprintf - print formatted output of a varargs argument list

SYNOPSIS

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

int vprintf (format, ap)
char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *format;
va_list ap;
```

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

vprintf, *vfprintf*, and *vsprintf* are the same as *printf*, *fprintf*, and *sprintf* respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by *varargs*(5).

EXAMPLE

The following demonstrates how *vfprintf* could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
 *      error should be called like
 *              error(function_name, format, arg1, arg2...);
 */
/*VARARGS0*/
void
error(va_alist)
/* Note that the function_name and format arguments cannot be
 *       separately declared because of the definition of varargs.
 */
va_dcl
{
        va_list args;
        char *fmt;

        va_start(args);
        /* print out name of function causing error */
        (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
        fmt = va_arg(args, char *);
        /* print out remainder of message */
        (void)vfprintf(stderr, fmt, args);
```

```
                va_end(args);
                (void)abort( );
        }
```

SEE ALSO

printf(3S), varargs(5).

NAME

intro - introduction to special files

HP-UX COMPATIBILITY

Remarks: In general, device drivers are not portable across systems; however, every effort has been made to make their behavior portable. Due to variation in hardware, this is not always possible. Programs which use these drivers directly are at higher than average risk of not being portable.

DESCRIPTION

This section describes various special files that refer to specific HP peripherals and device drivers. The names of the entries are usually derived from the type of device being described (disc, plotter, etc.), not the names of the special files themselves. Characteristics of both the hardware device and the corresponding HP-UX device driver are discussed where applicable.

The devices are divided into two catagories, **unblocked** and **blocked.** An unblocked device is also called a **raw** or character-mode device. An unblocked device, such as a line printer, uses a character special file.

Blocked devices, as the name implies, transfer data in blocks via the system's normal buffering mechanism. Block devices use block special files.

For specific details about the default special files shipped with your system, consult the System Administrator Manual for your system.

The desired name is associated with a specific device when *mknod*(1M) is used to create a special file for that device. The following naming convention is recommended for use when creating special files (special file names are are independent of the hardware):

[**r**] dev_id [prod_no] [model_initial] [**s**|**d**|**i**] [.] [digit]

where:

r	if present, indicates that the device is treated as a raw device; otherwise, the file is a block-mode file.
dev_id	consists of one of the following mnemonics:

ct	CS/80 catridge tape drive
hd	hard disc
mt	9-track tape drive
fd	flexible disc
lp	line printer
dig	digitizer
plt	plotter or graphics CRT display

prod_no	HP product number for the device;
model_initial	the letter suffix (if any) associated with the device model number;
s|**d**|**i**	used with the HP 9895A to specify **S**ingle-sided, **D**ouble-sided, or **I**BM media format;
.digit	used when two or more identical devices are connected to the system; for example, if two HP 2631G printers are connected to the system, their special file names would be *lp2631g* and *lp2631g.1* (they could also be named *lp* and *lp.1*).

HARDWARE DEPENDENCIES

Series 500:

Block special files cannot be opened for reading or writing.

The IBM format capability in the HP 9895A is not officially supported on HP-UX.

NAME

ct - cartridge tape access

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	HP

DESCRIPTION

This page describes the actions of the general HP-UX cartridge tape drivers when referring to a cartridge tape as either a block-mode or character-mode (raw) device.

Cartridge tapes are designed for use as "streaming" devices, and are not designed to start and stop frequently. Like discs, they are technically "random access" devices, but such access is less efficient and causes more tape and drive wear than streaming mode. While it is possible to use a cartridge tape as a file system or random access storage device, such use will dramatically reduce the life expectancy of tape cartridges and the tape drive itself.

Any CS/80 cartridge tape unit, whether built into a disc drive or operated as a standalone device, can be accessed as a blocked or raw device.

Block special files access cartridge tapes through normal system buffering mechanisms. Buffering is handled such that concurrent accesses through multiple opens or mounts on the same physical device do not get out of phase. Block special files can be read and written without regard to physical cartridge tape records. Each I/O operation results in one or more logical block transactions. Use of this mode is discouraged because it increases wear on tapes and drives.

Character special files provide a *raw* interface for transferring data directly between the cartridge tape and the user's read or write buffer. A single read or write operation always results in exactly one I/O transaction. This is considerably more efficient than block-mode I/O which can require several transactions to transfer the same amount of information and cannot handle the transfer directly between the drive and user space.

Tcio(1) is provided on some systems so you can take advantage of the efficiencies of raw I/O, while also making optimal use of the streaming capabilities of the cartridge tape drives. During writes, buffers small transactions into larger data blocks that are optimal for cartridge tapes, and reverses the process during reads. It is particularly designed for use as a complement to *cpio* for handling backups.

During raw I/O, there may be implementation-dependent restrictions on the alignment of the user buffer in memory and its maximum size. Also, each transfer must occur on a record boundary and must read a whole number of records. Record size is hardware dependent, but is usually 1024 bytes. Use of *tcio (1)* hides all these issues.

Each raw access is independent of other raw accesses and of block accesses to the same physical device. Thus, transfers are not guaranteed to occur in any particular order. Having multiple programs access the cartridge tape is, in effect, random access, and is subject to the warnings above.

In raw I/O, each operation is completed to the device before the call returns. For block-mode writes, data may be cached until it is convienient for the system to write it. In addition, block-mode reads potentially do a one (or more) block read-ahead. The interaction of block-mode and raw access to the same cartridge tape is not specified, and in general is unpredictable. Because block-mode writes can be delayed, it is possible for a program to generate requests much more rapidly than the drive can actually process them. Flushing a large number of requests could take several minutes, and during that time the system will not have use of the buffers taken by these requests, and thus will suffer a possibly severe performance degradation. If the tape drive and the system disc share a single controller, disc activity may be severely limited or stopped until the buffers are flushed.

The name of a raw device (its character special file name) is typically the same as the name of the corresponding blocked device (its block special file name) prefixed with an "r".

HARDWARE DEPENDENCIES

Series 500:

Block–special devices cannot be opened for I/O.

SEE ALSO

mkdev(1M), mknod(1M), tcio(1), intro(4), disc(4), mt(4), and the *HP–UX System Administrator Manual* provided with your system.

NAME
disc - direct disc access

HP-UX COMPATIBILITY
Level: HP-UX/RUN ONLY

Origin: HP

DESCRIPTION
This page describes the actions of the general HP-UX disc drivers when referring to a disc as either a blocked or unblocked (raw or character special) device.

Block special files access discs via the system's normal buffering mechanism. Buffering is done in such a way that concurrent access through multiple opens or mounts of the same physical device do not get out of phase. Block special files may be read and written without regard to physical disc records. Each I/O operation results in one or more logical block transactions.

There is also a *raw* interface via a character special file which provides for direct transmission between the disc and the user's read or write buffer. A single read or write operation results in exactly one transaction. Therefore raw I/O is considerably more efficient when many bytes are transmitted in a single operation because blocked disc access requires potentially several transactions and does not transmit directly to user space.

In raw I/O, there may be implementation dependent restrictions on the alignment of the user buffer in memory. Also, each transfer must occur on a sector boundary and must read a whole number of sectors. The sector size is a hardware dependent value (1024 bytes is the generally preferred value).

Each raw access is independent of other raw accesses and of block accesses to the same physical device. Thus, transfers are not guaranteed to occur in any particular order.

In both raw and blocked I/O, each operation is completed to the device before the call returns. In addition, blocked I/O potentially does a one (or more) block read-ahead.

The name of a raw device (its character special file name) is typically the same as the name of the corresponding blocked device (its block special file name) prefixed with an "r".

SEE ALSO
intro(4), mkdev(1M), mknod(1M), and the *HP-UX System Administrator Manual* included with your system.

WARNING
On some systems, having both a mounted file system and a block special file open on the same device is asking for trouble; this should be avoided if possible. This is because it may be possible for some files to have private buffers in some systems.

Like discs, the cartridge tape units in command set 80 disc drives are also accessed as blocked or raw devices. However, using a cartridge tape as a file system will severely limit the life expectancy of the tape drive. Tapes should only be used for system back-up and other needs where data must be stored on tape, such as for transport or other uses.

NAME

CRT graphics – information for CRT graphics devices

HP-UX COMPATIBILITY

Level:	HP-UX/NON-STANDARD
Origin:	HP
Remarks:	This information is valid for Series 200 and 300 only.

DESCRIPTION

CRT graphics devices are frame-buffer based raster displays. These devices use memory mapped I/O to obtain much higher performance than is possible with tty-based graphics terminals. CRT graphics devices should only be accessed through the STARBASE libraries. They cannot be piped or redirected to because they are not serial devices.

Special (device) files for CRT graphics devices are character special files with major number 12.

The minor number for CRT graphics devices is of the form:

```
0xSSTTXX
```

where SS is a one-byte select code number, TT is a one-byte type specifier, and XX is an unused byte (should be zero).

The type field in the minor number is defined as follows:

0 auto-configures to one of the following:

- a) low resolution graphics device at physical address 0x520000 (if present).
- b) high resolution graphics device at physical address 0x560000 if low resolution device at 0x520000 not present.

1 high resolution graphics device at physical address 0x560000 (unless there is no low resolution device at 0x520000, in which case type 1 is invalid).

2 high or low resolution graphics device at the select code specified by the select code field in the minor number.

Communication with a CRT graphics device is begun with an *open* system call. Multiple processes may concurrently have the graphics device open.

Close shuts down the file descriptor associated with the graphics device. If the close is for the last system wide open on the device then the graphics device is also unmapped from the user address space; otherwise it is left mapped into the user address space (see GCUNMAP below).

Read and *write* system calls are undefined and will always return ENXIO.

Ioctl is used to control the graphics device. The valid ioctl commands (see **<sys/graphics.h>**) are:

GCID return the identity of the CRT graphics device. The possible identities are:

1 = 98204A

2 = 9826A

3 = 9836A

4 = 9836C

5 = 98627A

6 = 98204B

7 = 9837

8 = 98700

9 = hp9000s300 displays

GCON, GCOFF
: turn graphics "on" or "off". May be a no-op for some devices.

GCAON, GCAOFF
: turn alpha "on" or "off". May be a no-op for some devices.

GCMAP
: map the CRT graphics device into the user address space at the address specified in the ioctl argument. The argument is 'char **arg'. The value *arg is used as a requested address. The actual mapping address is then returned in *arg. If *arg is 0 then the system selects the first available address (see GCLOCK/GCUNLOCK below).

GCUNMAP
: remove the mapping of the CRT graphics device from the user address space.

GCLOCK
: ensure exclusive use of the CRT graphics device.

GCUNLOCK
: relinquish exclusive use of the CRT graphics device.

For all frame buffers the data bytes scan from left to right and from top to bottom. Some displays map in control areas which must be skipped over to reach the frame buffer. Some devices map individual bits to pixels, (dots on the screen.) Some map bytes or parts of bytes to pixels. Lsb stands for least significant bit; msb stands for most significant bit.

98204A and 9826A
: There are 300 lines of 100 bytes each. Only the odd numbered bytes are used. There is a one bit per pixel, with msb left, and lsb right.

9836A and 98204B
: There are 390 lines of 64 bytes each. There is a one bit per pixel, with msb left, and lsb right.

9836C
: Starting 0x800 bytes from the base address, there are 390 lines of 512 bytes each. All bytes are used. There is one byte per pixel. The bottom four bits of each byte are a color map index for that pixel.

98627A
: Starting 0x8000 bytes from the base address, there are 3 buffers of 0x8000 bytes each. The 3 buffers are the data for red, green, and blue. There is one bit per pixel, with msb left, and lsb right. There are 64 bytes per line. The number of lines depends on the setting of control registers.

9837
: Starting 0x10000 bytes from the base address, there are 768 lines of 1024 bytes each. There is one pixel per byte. The lsb of each byte corresponds to a pixel.

98700
: Starting 0x10000 bytes from the base address, there are 768 lines of 1024 bytes each. There is one pixel per byte. Each byte corresponds to the color map index of a pixel.

Series 300 Displays:

These displays have registers describing the display size. The following code computes frame buffer width and height and determines what portion of the frame buffer is being displayed.

```
/* unsigned char *base = <base address for display mapping>; */
buffer_width = (base[5] << 8) + base[7];
buffer_height = (base[9] << 8) + base[11];
displayed_width = (base[13] << 8) + base[15];
displayed_height = (base[17] << 8) + base[19];
not_square = ((base[23] & 1) == 1);
```

Starting 0x10000 bytes from the base address, there are <buffer_width> lines of <buffer_height> bytes each. There is one pixel per byte. Each byte corresponds to the color map index of a pixel. On a monochrome display, the byte value is either 0, (black), or 1 (white). If ((base[23] & 1) == 1) then pixels are twice as high as they are wide, and may be used in pairs to produce square double pixels.

One shared memory descriptor (see *shmget*(2)) is used for each graphics device. Each shared memory descriptor is accessible only through its graphics interface. Thus, any attempt to access them through *shmat*(2)), *shmctl*(2)), *shmdt*(2)), etc. results in EACCESS errors.

ERRORS

[ENXIO]	no such device or read/write not supported.
[ENOSPC]	cannot allocate required resources for mapping.
[ENOMEM]	cannot allocate sufficient memory for mapping.
[ENOTTY]	bad ioctl command, or an ioctl was attempted on an open file.

SEE ALSO

mknod(8).

NAME

hpib - hpib interface information

HP-UX COMPATIBILITY

Level: HP-UX/NON-STANDARD

Origin: HP

DESCRIPTION

HP-IB is Hewlett-Packard's implementation of the Institute of Electrical and Electronic Engineers Standard Digital Interface for Programmable Instrumentation. For more information about the standard, consult any of the following documents:

IEEE Std 488-1978
IEC Pub 625-1
ANSI MC1.1

A read operation on a device connected to an HP-IB configures the computer as "listener" and the device as "talker". The read operation terminates when the number of bytes requested has been transferred, a pattern termination character is matched, or the device asserts the EOI (end or identify) line. A write operation configures the computer as "talker" and the device as "listener". The write operation terminates when the number of bytes specified has been transferred and it has asserted EOI.

Devices connected to an HP-IB are addressed using three values. The first value, called the *major value*, is used to select the appropriate device driver. The second value is called the *select code.* The select code refers to the I/O interface card or slot number to which the device is connected. The third value is called the *HP-IB address.* The HP-IB address is usually set by an in-line or rotary switch on the device itself. Refer to the device reference manual for information on setting the HP-IB address.

This driver is also used to access HP-IB plotters, digitizers and printers in *raw* mode. A printer in raw mode is used as a graphics device.

The *HP-IB address* is in the range 0 through 30, unless you want to use the device-independent library (DIL) to talk to an I/O interface card on a raw bus (no automatic addressing). In this case, use HP-IB address 31. For more details on DIL, refer to the *hpib**(3D), *io**(3D), and *gpio**(3D) manual entries.

Terminating a write operation does not depend on an asserted EOI.

HARDWARE DEPENDENCIES

Series 200:

The *major value* for HP-IB raw mode printers, plotters, and digitizers is always 21 (RAW HP-IB).

Series 500:

The *major value* for HP-IB printers, plotters, and digitizers is always 12 for HP 27110 cards and 37 for the Model 550 internal HP-IB.

SEE ALSO

intro(4), mknod(1M), documentation for the specific device.

Status: R

NAME

iomap -- physical address mapping

HP-UX COMPATIBILITY

Level:	HP-UX/NON-STANDARD
Origin:	HP
Remarks:	This information is valid for Series 200 and 300 only.

DESCRIPTION

The iomap mechanism allows the mapping (thus direct access) of physical addresses into the user process address space. For Series 200/300 computers, the physical address space begins at 0x000000 and extends to 0xffffff.

The special (device) files for iomap devices are character special files with major number 10.

The minor number for iomap devices is of the form:

0xAAAANN

where AAAA is a two-byte address, and NN is a one-byte field.

The address portion of the minor number is formed by dividing the physical address by 65536. NN*65536 is the size of the region to be mapped. For example, the minor number for a device at 0x720000 and 128k in size is 0x007202.

Access to the iomap devices is controlled by the file permissions set on the character special file.

Multiple processes may concurrently have iomap devices opened and mapped. It is the responsibility of the processes to synchronize their accesses.

Read and *write* system calls are not supported.

Ioctl is used to control the iomap device. The valid ioctl commands (see **<iomap.h>**) are:

IOMAPMAP
: map the iomap device into the user address space at the address specified in the ioctl argument. If the argument is 0 then the system selects the first available address. Multiple processes may concurrently have the iomap device mapped.

IOMAPUNMAP
: unmap the iomap device from the user address space.

Close shuts down the file descriptor associated with the iomap device. If the close is for the last system wide open on the device then the iomap device is also unmapped from the user address space; otherwise it is left mapped into the user address space (see IOMAPUNMAP above).

One shared memory descriptor (see *shmget*(2)) is used for each iomap device. Shared memory descriptors are accessible only through the iomap interface. Consequently, attempts to access them through *shmat*(2), *shmctl*(2), *shmdt*(2), etc. result in EACCESS errors.

WARNING

Iomap devices should be created and used with extreme caution. Inappropriate accesses to io devices or ram may result in a system crash.

ERRORS

[ENINVAL]	address field out of range, ioctl command invalid.
[ENOMEM]	cannot allocate required memory for mapping.
[ENODEV]	read/write unsupported.
[ENXIO]	no such address.

[ENOSPC]	cannot allocate required resources for mapping.
[ENOTTY]	bad ioctl command, or an ioctl was attempted on an open file.

SEE ALSO

mknod(8).

NAME

lp - line printer

HP-UX COMPATIBILITY

Level:	HP-UX/NUCLEUS
Origin:	System V

DESCRIPTION

All file names in **/dev** containing the mnemonic *lp* are special files providing the interface to a particular line printer. A line printer is a character special device which may optionally have an interpretation applied to the data.

If the *lp* mnemonic is preceded by the character **r**, then data is sent to the printer in *raw mode.* (This could assume, for example, a graphic printer operation.) In raw mode, no interpretation is done on the data to be printed, and no page formatting is performed. The bytes are simply sent to the printer and printed as is.

If the *lp* mnemonic is **not** preceded by the character **r**, then the data is interpreted according to rules discussed below. The driver understands the concept of a printer page in that it has a page length (in lines), line length (in characters), and indent from the left margin (in characters). The default line length, indent, lines per page, open and close page eject, and handling of backspace are set to defaults determined when the printer is opened and recognized by the system the first time. If the printer is not recognized, the default line length is 132 characters, indent is 4 characters, lines per page is 66, one page is ejected on close and none on open, and backspace is handled for a character printer.

The following rules describe the interpretation of the data stream.

> A form feed causes a page eject and resets the line counter to zero.
>
> Multiple consecutive form–feeds are treated as a single form–feed.
>
> The new–line character is mapped into a carriage–return/line–feed sequence, and if an offset is specified a number of blanks are inserted after the carriage–return/line–feed sequence.
>
> A new–line that extends over the end of a page is turned into a form–feed.
>
> Tab characters are expanded into the appropriate number of blanks (tab stops are assumed to occur every eight character positions).
>
> Backspaces are interpreted to yield the appropriate overstrike either for a character printer or a line printer.
>
> Lines longer than the line length minus the indent (i.e., 128 characters, using the above defaults) are truncated.
>
> Carriage–return characters cause the line to be overstruck.

Two *ioctl*(2) system calls are available to control the lines per page, characters per line, and indent. At either open or close time, if no page eject is requested, the paper will not be moved.

```
#include <sys/lprio.h>
ioctl (fildes, command, arg)
struct lprio *arg;
```

The *commands* are:

LPRGET	Get the current printer status information and store in the *lprio* structure referenced by **arg**.
LPRSET	Set the current printer status information from the structure referenced by **arg**.

Thus, indent, page width and page length can be set with an external program. If the columns field is set to zero, the defaults are restored at the next open.

FILES

/dev/lp	default or standard printer used by some HP–UX commands;
/dev/[r]lp*	special files for printers

HARDWARE DEPENDENCIES

Series 500:

The number of characters per line (80 or 132) and wrap–around can be selected/enabled via the *minor* number in the *mknod*(1M) command. See the *System Administrator Manual* for details.

The LPRGET and LPRSET *ioctl* commands are not currently supported.

Series 200:

The uppercase–only flag, the no–overprint flag, the raw–mode flag, and no–page–eject–on–open–or–close flag can be selected (enabled) by appropriate use of the minor number in the *mknod*(1M) command. See the *HP–UX System Administrator Manual* for details.

Integral PC:

This version of *lp* is not supported on the Integral PC. Refer to the *Integral Personal Computer Programmer's Guide* for more information about the *lp* implementation on the Intregral PC.

SEE ALSO

lp(1), ioctl(2), intro(4).

NAME

mem, kmem - core memory

HP-UX COMPATIBILITY

Level:	HP-UX/Optional
Origin:	System III
Remarks:	Not all HP-UX systems provide the *mem* and *kmem* files. Programs which use them cannot expect to be portable from one HP-UX implementation to another.

DESCRIPTION

Mem is a special file that is an image of the core memory of the computer. It may be used, for example, to examine, and even to patch the system.

Byte addresses in *mem* are interpreted as memory addresses. References to non-existent locations cause errors to be returned.

Examining and patching device registers is likely to lead to unexpected results when read-only or write-only bits are present.

The file *kmem* is the same as *mem* except that kernel virtual memory rather than physical memory is accessed.

Mem and *kmem* should always be protected so that only the super-user can read and write them, othewise both privacy and system security are compromised.

FILES

/dev/mem, /dev/kmem

HARDWARE DEPENDENCIES

Series 500:

Mem and *kmem* are not provided.

Series 200:

Memory does not begin at physical address 0. Instead, it occupies the upper portion of the physical address space from 0x900000 through 0xffffff, beginning at address 0xffffff and moving downward. Thus, if one megabyte of RAM is installed, it occupies addresses 0xf00000 through 0xffffff.

BUGS

On some machines memory files are accessed one byte at a time, an inappropriate method for some device registers.

NAME

modem - asynchronous serial modem line control

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

DESCRIPTION

This section describes the two modes of modem line control and the three types of terminal port access. It also discusses the effect of several bits of the *termio* structure which affect modem line control. The modem related *ioctl*(2) system calls are discussed at the end of the document.

There are several terms that are used within the following discussion which will be defined here for reference. "Modem control lines" (CONTROL) are generally defined as those outgoing modem lines that are automatically controlled by the driver. "Modem status lines" (STATUS) are generally defined as those incoming modem lines that are automatically monitored by the driver. CONTROL and STATUS for a terminal file vary according to the modem line control mode of the file (see **Modem line control modes** below). An *open*(2) to a port will be considered to be blocked if it is waiting for another file on the same port to be closed. An *open* to a port will be considered to be pending if it is waiting for the STATUS to be raised. An *open* to a port will be considered to be successful if the *open* system call has returned to the calling process without error.

Open flag bits

The only *open* flag bit recognized by the driver is the O_NDELAY bit. When this bit is set, an *open* call to the driver will never become blocked. If possible, the *open* will be returned immediately as successful, and the driver will continue the process of opening the tty file. If it is not possible, then the *open* will be returned immediately with the appropriate error code as described in the appropriate section.

Termio bits

The CLOCAL bit in the *termio* structure (see *termio*(4)) is used to remove the driver's automatic monitoring of the modem lines. However, the user's ability to control the modem lines is determined only by the mode in effect and does not depend on the state of CLOCAL . Normally, the driver will monitor and require the STATUS to be raised. An *open* system call will raise the CONTROL and wait for the STATUS before completing unless the CLOCAL bit is set. (If the O_NDELAY bit is set, the *open* will be returned immediately, but the driver will otherwise continue to monitor the modem lines as normal based on the state of the CLOCAL bit.) If CLOCAL is set when the last *close*(2) is issued to the port, the driver will not attempt to break any modem connection which may exist unless the HUPCL bit is set (see below). Normally, loss of the STATUS will cause the driver to break the modem connection and lower the CONTROL ; however, if CLOCAL is set, any changes in the STATUS will be ignored. A connection is required before any data may be read or written, unless CLOCAL is set. Any timers that would normally be in effect (see **Modem line control modes** and **Modem timers** below) will be stopped while CLOCAL is set.

When the CLOCAL bit is changed from clear to set, the driver will assume the existence of an active device (such as a modem) on the port regardless of the STATUS . If any of the CONTROL are raised at that point in time, they will continue in that state. The STATUS will no longer be actively monitored. When the CLOCAL bit is changed from set to clear, the driver will resume actively monitoring the STATUS . If all of the CONTROL are raised at that point in time, the driver will attempt to begin or continue a modem connection. If any of the CONTROL are not raised, the driver will break the modem connection. If any of the STATUS are not raised, the driver will act as though those signals were lost as described in **Modem line control modes** below. If the device is a controlling terminal, a *hangup* signal will be sent to the process group.

The HUPCL bit in the *termio* structure determines the action of the driver regarding the CONTROL when the last *close* system call is issued to a terminal file. If the HUPCL bit is set, the driver will lower the CONTROL at *close* time and the modem connection will be broken. If HUPCL is not set and a modem connection exists, it will continue to exist, even after the *close* is issued.

Terminal port access types

There are three types of modem access: call-in connections, call-out connections, and direct (no modem control) connections. A given port may be accessed through all three types of connection by accessing different files. The modem access type of a terminal file is determined by the file's major and/or minor device numbers.

The call-in type of access is used when the connection is expected to be established by an incoming call. This is the type that would be used by *getty*(1M) to accept logins over a modem. When an *open* is issued to such a file, the driver may wait for an incoming call and will then raise the CONTROL based on the current mode (see below) of the port. When the port is closed, the driver may lower the CONTROL depending on the HUPCL bit.

The call-out type of access is used when the connection is expected to be established by an outgoing call. This would be used by programs such as *uucp*(1). When an *open* is issued to such a file, the driver will immediately raise the CONTROL and wait for a connection based on the mode currently in effect. When the port is closed, the driver may lower the CONTROL depending on the HUPCL bit.

The direct type of access is used when no driver modem control is desired. This could then be used for directly connected terminals that use a three-wire connection, or to talk to a modem before a connection has been established. The second case allows a program to give dialing instructions to the modem. Neither the CLOCAL nor the HUPCL bits have any effect on a port accessed through a direct file. (However, both bits may be inherited by other types of files; see **Terminal port access interlock** below.) An *open* to a direct file does not affect the CONTROL and does not depend on any particular state of the STATUS to succeed. When the file is closed, the driver will not affect the state of the CONTROL . If a modem connection has been established, it will continue to exist. Setting the speed of a direct file to B0 (see *termio*(4)) will be considered an impossible speed change and will be ignored. It will not affect the CONTROL .

Modem line control modes

There are two modes of modem line control: CCITT mode and simple mode. A given port may have only one of these two modes in effect at any given point in time. An attempt to *open* a port with a mode other than the one in effect (from a pending or successful *open* on a different file) will cause the *open* to be returned with an ENXIO error. The modem access type of a terminal file is determined by the file's major and/or minor device numbers.

CCITT mode is used for connections to switched line modems. The CONTROL for CCITT mode are Data Terminal Ready (DTR) and Request to Send (RTS). The STATUS are Data Set Ready (DSR), Data Carrier Detect (DCD), and Clear to Send (CTS). Additionally, the Ring Indicate (RI) signal indicates the presence of an incoming call. When a connection is begun (an incoming call for a call-in file or an *open* issued to a call-out file), the CONTROL are raised and a connection timer (see **Modem timers** below) is started. If the STATUS become raised before the time period has elapsed, a connection is established and the *open* request is returned successfully. If the time period expires, the CONTROL are lowered and the connection is aborted. For a call-in file, the driver will wait for another incoming call; for a call-out file, the *open* will be returned with an EIO error. Once a connection is established, loss of either DSR or CTS will cause the CONTROL to be lowered and a *hangup* signal to be generated if the device is a controlling terminal. If DCD is lost, a timer is started. If DCD resumes before the time period has expired, the connection will be maintained. However, no data transfer will occur during this time. The driver will stop transmitting characters, and any characters received will be discarded. If DCD is not restored within the allotted time, the connection will be broken as described above for DSR and CTS . If the modem connection is to be broken when the *close* system call is issued (i.e. HUPCL is set), then the CONTROL will be lowered and the *close* will be returned as successful. However, no further *opens* will be allowed until after both DSR and CTS have been lowered by the modem, and the hangup timer (see **Modem timers** below) has expired. The action taken in response to an *open* during this time will be the same as if the port were still open. (See **Terminal port access interlock** below.)

When a port is in CCITT mode, the driver has complete control of the modem lines and the user is not allowed to change the setting of the CONTROL or affect which STATUS are actively monitored by the driver (see **Modem ioctls** below). This is to provide strict adherence with the CCITT recommendations.

Simple mode is used for connections to devices which require only a simple method of modem line control. This can include devices such as black boxes, data switches, or for system-to-system connections. It can also be used with modems which can not operate under the CCITT recommendations. The CONTROL for simple mode consists of only DTR . The STATUS consists of only DCD . When an *open* is issued, the CONTROL is raised but no connection timer is started. When the STATUS becomes raised, a connection is established and the *open* request is returned successfully. Once a connection is established, loss of the STATUS will cause the CONTROL to be lowered and a *hangup* signal to be generated if the device is a controlling terminal.

When a port is in simple mode, the driver will normally control the modem lines. However, the user is allowed to change the setting of the CONTROL (see **Modem ioctls** below).

Terminal port access interlock

An interlock mechanism is provided between the three access types of terminal files. It prevents more than one file from being successfully opened at a time, but allows certain *opens* to succeed while others are pending so that a port can be opened through a call-out connection while *getty* has a pending *open* at a call-in connection. The three access types are given a priority that determines which *open* will succeed if more than one file has an *open* issued against it. The three access types are ordered from lowest priority to highest as follows: call-in, call-out, and direct.

If an *open* is issued to a port which already has a successful *open* on it of a lower priority type, the new *open* will be returned with an EBUSY error. (EBUSY will also be returned if a CCITT call-in file is not yet successful, but has received an incoming call indication.) If the lower priority *open* is pending, the new *open* will succeed if possible, or will be left pending if waiting for the STATUS . If a higher priority *open* has succeeded or is pending, the new *open* will be blocked, unless the new *open* has the O_NDELAY flag bit set, in which case the *open* will be returned with an EBUSY error. Once an *open* on one type of file is successful, any pending *opens* on lower priority files will become blocked.

When a file of one priority is closed, a blocked *open* on the next lower priority type file will become active. If all of the STATUS are raised, the *open* will succeed, otherwise the *open* will become pending waiting for the STATUS . If the lower priority *open* is successful (because the connection was maintained when the higher priority file was closed), the port characteristics (speed, parity, etc.) that were set by the higher priority file will be inherited by the lower priority file. If the connection is not maintained through the *close*, the port characteristics will be set to default values at the next successful *open*.

Modem timers

There are four timers currently defined for use with modem connections. The first three of the timers are applicable only to CCITT mode connections. In general, the effect of changing a timer value while the timer is running is system dependent. However, setting the timer value to zero is guaranteed to disable the timer even if it is running.

The connect timer is used to limit the amount of time to wait for a connection to be established once it has been begun. This timer is started when an incoming call has been received on a call-in file, or when an *open* has been issued on a call-out file for which no *opens* are already pending. If the connection is completed in time, the timer is aborted. If the time period expires, the connection is aborted. For a call-in file, the driver will again wait for an incoming call and the *open* will remain pending. For a call-out file, the *open* will be returned with an EIO error.

The carrier detect timer is used to limit the amount of time to wait before causing a disconnect if DCD drops. If carrier is not re-established in this time, a disconnect will occur. If carrier is re-established before the timeout, the timer will be aborted and the connection maintained. During the period when carrier is not raised, no data will be transferred across the line.

The no activity timer is used to limit the amount of time a connection will remain open with no data transfer across the line. When the data line becomes quiescent with no data transfer, this timer will be started. If data is again transferred over the line in either direction before the time limit, the timer will be aborted. If no activity occurs before the timeout has occurred, the driver will disconnect the line. This can be used to avoid long and costly telephone connections when data transfer has been stopped either normally or abnormally.

The last timer defined, the hangup timer, is used for both CCITT and simple modes. This timer controls the amount of time to wait after disconnecting a modem line before allowing another *open* to be allowed. This time period should be made long enough to guarantee that the connection has been terminated by the telephone switching equipment. If this period is not long enough, the telephone connection may not be broken and a succeeding *open* may complete with the old connection.

Modem ioctls

Several *ioctl* system calls apply to manipulation of modem lines. They use the following information defined in **<sys/modem.h>**.

```
#define NMTIMER      6
typedef unsigned long  mflag;
struct  mtimer  {
        unsigned short m_timers[NMTIMER];
        };
```

Individual modem lines are represented by bits in an unsigned long variable as follows:

MRTS	00010000000	Request to Send	outbound
MCTS	00004000000	Clear to Send	inbound
MDSR	00002000000	Data Set Ready	inbound
MDCD	00000400000	Data Carrier Detect	inbound
MDTR	00000000040	Data Terminal Ready	outbound
MRI	00000000010	Ring Indicator	inbound
MDRS	00000000004	Data Rate Select	outbound

The timer values are defined in the array m_timers. The relative position of the timer and default initial values and units for each timer are as follows:

0	MTCONNECT	25 s
1	MTCARRIER	400 ms
2	MTNOACTIVITY	0 min
3	MTHANGUP	250 ms
4	Reserved	
5	Reserved	

A value of zero for any timer will disable that timer.

The modem line *ioctl* system calls have the form:

```
ioctl (fildes, command, arg)
unsigned long *arg;
```

The commands using this form are:

MCGETA	Get the current state of both inbound and outbound modem lines and store in the unsigned long referenced by **arg**. A raised line will be indicated by a one bit in the appropriate position.
MCSETA	Set the outbound modem lines from the unsigned long referenced by **arg**. Setting an outbound bit to one causes that line to be raised and zero to be lowered. Setting bits for inbound lines has no effect. Setting any bits while in CCITT mode has no effect. The change to the modem lines is immediate and using this form while characters are still being output may cause unpredictable results.
MCSETAW	Wait for the output to drain and set the new parameters as described above.
MCSETAF	Wait for the output to drain, then flush the input queue and set the new parameters as described above.

The timer value *ioctl* system calls have the form:

```
ioctl (fildes, command, arg)
struct mtimer *arg;
```

The commands using this form are:

MCGETT	Get the current timer value settings and store in the *mtimer* structure referenced by **arg**.
MCSETT	Set the timer values from the structure referenced by **arg**.

For any timer, setting the timer value to its previous value has no effect.

FILES

/dev/tty*
/dev/ttyd*
/dev/cul*
/dev/cua*

HARDWARE DEPENDENCIES

Series 500:

For the HP27140A 6-port modem multiplexer, the ranges and resolutions of the timers are as follows:

MTCONNECT	0-255 sec, 1 sec resolution
MTCARRIER	0-2550 msec, 10 msec resolution
MTNOACTIVITY	0-1092 min, 1 min resolution
MTHANGUP	0-65535 msec, 10 msec resolution

If a timer is set out of its range, then the maximum value that timer can assume is used instead.

For the HP27128A Asynchronous Serial Interface, the *ioctl* requests described above are not supported. The timers have fixed values as follows:

MTCONNECT	25 sec
MTCARRIER	400 msec
MTNOACTIVITY	0 min
MTHANGUP	500 msec

This interface only supports the call-in and call-out port access types, and does not support the direct access type.

It is not possible to change the state of the CLOCAL bit when using CCITT mode.

Simultaneous call-in and call-out open atempts in CCITT mode are not allowed.

The default state of the CLOCAL bit upon first open is determined by the state of switch on the interface (See the *System Administrator Manual*).

SEE ALSO

stty(1), mknod(1M), ioctl(2), termio(4).
HP-UX System Administrator Manual

WARNING

It is occasionally possible that a process may open a call-out file at approximately the same time as an incoming call is received. In some cases, the call-out connection may be satisfied by the incoming call. In general, however, the results are indeterminate. If necessary, the situation can be avoided by the use of two modems and ports, one for call-out connections and the other for receiving incoming calls.

NAME

magtape - magnetic tape interface and controls

HP-UX COMPATIBILITY

Level: Magnetic Tape Support — HP-UX/RUN ONLY

Origin: UCB and HP

DESCRIPTION

The files **/dev/mt*** and **/dev/rmt*** refer to specific tape drives; the behavior of the specific unit is specified by several bits in the least significant digit of the *minor* number in the *mknod*(1M) command.

There are three bits controlling the operation of the tape drive. These bits are usually encoded into the *minor* number of *mknod*(8). Refer to the System Administrator Manual for your computer for details.

rewind	When this bit is cleared, the tape is automatically rewound upon close. This is normally done for units numbered 0–3 and 8–11.
mode	When this bit is set, the tape drive behaves like the Berkeley tape drivers; when clear the driver behaves like System III. The details are described below. The *ioctl* operations described below work in both modes on *raw* tapes only.
density	When cleared, the tape drive is run at 1600 bpi; when set it is run at 800 bpi. The 800 bpi drives are usually numbered 0–7, and 1600 bpi are usually numbered 8–15.

When opened for reading or writing, the tape is assumed to be positioned as desired.

When a file is opened for writing and then closed, a double end–of–file (double tape mark) is written. If the device has the rewind bit set, the tape is rewound; otherwise, the tape is positioned before the second EOF just written.

When a read–only file is closed and the rewind bit is set, the tape is rewound. If the rewind bit is not set, the behavior depends on the mode bit. For System III compatibility, the tape is positioned after the EOF following the data just read. For Berkeley compatibility, the tape is not repositioned in any way.

The EOF is returned as a zero–length read.

By judiciously choosing **mt** files, it is possible to read and write multi–file tapes.

A tape treated as a block device consists of several 512 byte records terminated by an EOF. To the extent possible, the system makes it possible to treat the tape like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time (although very inadvisable).

The **mt** files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the *raw* interface is appropriate. The raw interface is described below.

The special files associated with a *raw* tape interface are named **rmt***. Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given.

During a read, the record size is passed back as the number of bytes read, up to the buffer size specified. The number of bytes ignored is available in the *mt_resid* field of the *mtget* structure via the MTIOCGET call of *ioctl.* In raw tape I/O, the buffer and size may have implementation dependent alignment restrictions. Seeks are ignored, instead the *ioctl* operations described below are available. An EOF is returned as a zero–length read, with the tape positioned after the EOF, so that the next read will return the next record.

Using Ioctl With Magnetic Tape

The *ioctl* system call can be used to manipulate magnetic tapes; refer to the include file

/usr/include/sys/mtio.h for a description of the possible operations.

The following code fragment shows how an *ioctl* call might be used to perform several mag tape operations:

```
#include <sys/types.h>
#include <sys/mtio.h>
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
        int fd;
        struct mtop top;

        /* open mag tape device file */

        top.mt_count = 1;

        switch(*argv[1]) {
                case '1':               /* write an eof */
                                        top.mt_op = MTWEOF;
                                        break;

                case '2':               /* rewind the tape */
                                        top.mt_op = MTREW;
                                        break;

                case '3':               /* backspace record */
                                        top.mt_op = MTBSR;
                                        break;

                default:
                                        fprintf(stderr, "Unknown option: %s0, argv[1]);
                                        exit(1);
                                        break;
        }
        fd = ioctl(fd, MTIOCTOP, &top);
}
```

This program accepts one argument which selects the operation to perform. The structure template *mtop* is defined in **sys/mtio.h**, and contains two parameters defining the operation to perform (*mt_op*) and how many operations to perform (*mt_count*). All constants used above (plus many others not used) are defined in **sys/mtio.h**.

HARDWARE DEPENDENCIES

Series 200:

Block magnetic tape is not supported.

The density bit cannot select 800 bpi; 800 bpi is not supported.

The settings of the mode, rewind, and density bits are reflected in the minor numbers used to create the special file names (see *mkdev*(8)).

Series 500:

Block magnetic tape is not supported.

The density bit cannot select 800 bpi; 800 bpi is not supported.

FILES

/dev/mt*
/dev/rmt*

SEE ALSO

intro(4), mkdev(1M), mknod(1M), and the *HP-UX System Administrator Manual* included with your system.

NAME

null - null file ("bit bucket")

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System III

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

FILES

/dev/null

NAME

pty - pseudo terminal driver

SYNOPSIS

pseudo–device pty

HP–UX COMPATIBILITY

Level: HP–UX/STANDARD

Origin: Berkeley 4.2

DESCRIPTION

The *pty* driver provides a communication path between an HP–UX application process and a supporting server process, and behaves much like a terminal/computer communication path. It is structured so that output from either process acts as input to the other, thus the term *pseudo*terminal. The slave–side of *pty* interacts with the application process, and its behavior is defined by *termio*(4). The master–side of *pty* interacts with the server process which controls the application process through *pty* as if *pty* were a hardware terminal interface.

The following *ioctl* requests, defined in **<sys/ptyio.h>**, apply only to master side of pty:

TIOCBREAK

Causes a break operation to be done on the slave side of the pty. This action is the same as if a user had hit the break key on a real terminal. Takes no parameters.

TIOCSIGSEND

Causes a signal to be sent on the slave side of the pty to the current tty process group of the slave side. The value of the parameter is taken to be the signal number to be sent. An EINVAL error will be returned and no signal sent if the specified signal number does not refer to a legal signal (see signal(2)). Note that this request allows the server process to send signals to processes that are not owned by the same user id.

TIOCSTOP

Stops data flowing from the slave side of the pty to the master side (e.g. like typing ^S). Takes no parameters.

TIOCSTART

Restarts output (stopped by TIOCSTOP or by typing ^S). Takes no parameters.

TIOCPKT

Enable/disable *packet* mode. Packet mode is enabled by specifying (by reference) a nonzero int parameter and disabled by specifying (by reference) a zero int parameter. When applied to the master side of a pseudo terminal, each subsequent *read* from the master side will return data written on the slave part of the pseudo terminal preceded by a zero byte (symbolically defined as TIOCPKT_DATA), or a single byte reflecting control status information. In the latter case, the byte is an inclusive–or of zero or more of the bits:

TIOCPKT_FLUSHREAD

whenever the read queue for the slave side is flushed.

TIOCPKT_FLUSHWRITE

whenever the write queue for the slave side is flushed.

TIOCPKT_STOP

whenever data flowing from the slave side of the pty to the master side is stopped by means of ^S, TIOCSTOP, or TCXONC.

TIOCPKT_START

whenever data flowing from the slave side of the pty to the master side is restarted.

TIOCPKT_DOSTOP
: whenever the stop and start characters get set to ^S/^Q.

TIOCPKT_NOSTOP
: whenever the stop and start characters get set to something other than ^S/^Q.

TIOCREMOTE
: A mode for the master half of a pseudo terminal, independent of TIOCPKT. This mode causes input to the pseudo terminal to be flow controlled and not input edited (regardless of the terminal mode). Each write to the master side produces a record boundary for the process reading the slave side. In normal usage, a write of data is like the data typed as a line on the terminal; a write of 0 bytes is like typing an end-of-file character (the EOF character as defined in termio(4)). The data read by the slave side is identical to the data written on the master side. Data written on the slave side and read on the master side with TIOCREMOTE enabled is still subject to the normal termio(4) processing. TIOCREMOTE can be used when doing remote line editing in a window manager, or whenever flow controlled input is required. The request takes one int sized parameter, passed by value. When zero, it disables TIOCREMOTE; when one it enables TIOCREMOTE. TIOCREMOTE is only effective when TIOCTTY (explained below) is also enabled, and all data buffered in the pseudo terminal will be flushed when this request is made.

TIOCTTY
: Enable or disable all termio(4) processing by pty. When disabled, all data is passed through the pty with no modification. Termio(4) processing (of input and output such as tab expansion) is enabled by specifying (by reference) a nonzero int parameter and disabled by specifying (by reference) a zero int parameter. Default is to be enabled. When TIOCTTY is disabled, the following pty modes are also inoperable: TIOCBREAK, TIOCSTOP, TIOCSTART, TIOCPKT, TIOCREMOTE, and TIOCMONITOR. Issuing a TIOCTTY ioctl request will also flush all data buffered in the pseudo terminal, and release any processes currently blocked waiting for data.

  When TIOCTTY is enabled (the default case), all termio(4) ioctl requests are handled by the pty driver itself. When TIOCTTY is disabled, slave side termio(4) ioctl requests are either ignored completely or passed to the master side depending upon the state of TIOCTRAP below. Slave side non-termio(4) ioctl requests are not affected by the state of TIOCTTY. They are always ignored completely or passed to the master side depending upon the state of TIOCTRAP below.

  Data being written through a pseudo terminal with TIOCTTY disabled will be handled in a manner similar to the way data flows through a pipe. A write request will block in the pty until all of its data has been written into the pty. A read request will block if there is no data available unless the O_NDELAY flag is set (see fcntl(2)). When data is available to be read, the read request will return whatever is available, and will not wait for the number of bytes requested to be satisfied. The number of bytes a pty can contain in its internal memory is implementation dependent, but will always be at least 256 bytes in each direction. For example, a write on the slave side of a pty of 1024 bytes might be read on the master side by four read requests returning 256 bytes each. The size of the chunks of data that are read is not guaranteed to be consistent, but no data will be lost.

Opening and closing of the master side acts as a modem connection/disconnection on a real terminal as far as the slave side is concerned. Having no server on the master side will cause opens on the slave side to hang until there is a server. (termio(4) description of O_NDELAY interaction with pty is also supported.) Opens to the master side are exclusive. Attempts to open an already open master side of a pty will return errno(2) error EBUSY. (Attempts to open a non-existent pty will return errno(2) ENXIO.) Closing the master side of a pty sends a SIGHUP hangup signal to the tty process group number of the corresponding slave side and flushes pending input and output.

Any termio(4) ioctl request can also be applied to the master side of the pty, unless TIOCTTY has been disabled.

IOCTL/OPEN/CLOSE TRAPPING

The capabilities that follow give additional flexibility and control for servers connected to the master side.

When trapping of ioctl/open/close is enabled, ioctl(2), open(2), and close(2) requests made to the slave side will notify the server on the master side of each request. The close request will only notify the server and continue to completion, while the open and ioctl requests will not complete until the master side has had a chance to handle them. The master side acknowledges completion via an ioctl to the master side. If the pty is not enabled to pass ioctl(2), open(2), and close(2) from the slave to the master, then they will be ignored (except for termio(4) related processing).

The following ioctl calls apply only to the master side of a pty and pertain to trapping open, close, and ioctl. They are also defined in **<sys/ptyio.h>**:

TIOCTRAP
: Enable or disable trapping of ioctl, open, and close from the slave side. Trapping is enabled by specifying (by reference) a nonzero int parameter and disabled by specifying (by reference) a zero int parameter. Default is to be disabled. (termio(4) ioctl requests will not be trapped, unless TIOCTTY is also disabled or TIOCMONITOR is enabled.)

TIOCTRAPSTATUS
: Find out if any ioctl/open/close traps are pending. The argument points to an int, that will be set to one if anything is pending and zero if nothing is pending. This ioctl request is used when the preferred method of a select(2) "exceptional condition" is not available.

TIOCREQGET
: In response to a select(2) "exceptional condition" on the master side, this ioctl request will read the pending ioctl, open, or close information into memory pointed to by the argument in the form:

```
struct request_info {
        int request;
        int argget;
        int argset;
        short pgrp;
        short pid;
        int errno_error;
        int return_value;
};
```

All elements of request_info refer to the slave side of the pty. Enumerating the elements:

request	is the ioctl command received.
argget	is the ioctl request to apply to master side to receive the trapped ioctl structure if there is one to receive, (a zero value means there is none). (When nonzero, argget is a TIOCARGGET request with the size field precomputed.)
argset	is the ioctl request to apply to master side to send back the resulting ioctl structure if there is one to send back, (a zero value means there is none). (When nonzero, argset is a TIOCARGSET request with the size field precomputed.)

pgrp	is the process group number of the process doing the operation.
pid	is the process id of the process doing the operation.
errno_error	is the *errno*(2) error code (initialized to zero) to be returned by ioctl on the slave side.
return_value	(initialized to zero) is the success value to be returned by ioctl on the slave side when *errno_error* is not set.

For the case that the ioctl argument received on the slave side is not a pointer, its value is stored as four bytes that can be retrieved with an ioctl request to the master side equal to *argget.*

When an open or close is being passed, *request* will be set to TIOCOPEN or TIOCCLOSE, respectively. For TIOCOPEN and TIOCCLOSE, both *argget* and *argset* will be of zero because there is no ioctl structure. When TIOCTTY is enabled, the termio(4) definition of open/close will be executed first, before being passed to the master side. Note, while all opens are trapped, only the last close on a particular inode for a pty slave side is trapped by the pty.

If a TIOCREQGET is done before anything has been trapped, this master side ioctl will block until a slave side ioctl, open, or close is trapped.

TIOCREQSET

Done to complete the handshake started by a previous TIOCREQGET. The argument should point to the request_info structure as defined by the TIOCREQGET.

Before doing this ioctl, to complete the handshake, the server should set *errno_error* to an errno(2) error value to be passed back to the slave side. If there is no error, *errno_error* can be left alone because the pty will have initialized it to zero. Also, when there is no error, *return_value* should be set, if other than a zero result is desired. It should be noted that the ability to determine the return value and error code for a request to the slave side is only available for trapped ioctl requests. The server will not be able to set these values if the trapped request is an open or a close.

If the TIOCREQSET request is made and *request* in the passed request_info structure does not equal the trapped value, errno(2) EINVAL will be returned. (EINVAL is also returned if there is no trapped ioctl/open/close.)

If the trapped slave-side request has been interrupted by a signal between the time that the server has done the TIOCREQGET and the TIOCREQSET, an EINVAL error will be returned by the TIOCREQSET request.

TIOCMONITOR

Enable or disable read only trapping of termio ioctl requests when TIOCTTY is also enabled. (When TIOCTTY is disabled, TIOCMONITOR has no effect. Also TIOCMONITOR is independent of TIOCTRAP.) Trapping is enabled by specifying (by reference) a nonzero int parameter and disabled by specifying (by reference) a zero int parameter. Default is to be disabled.

This allows a server process attached to the master side of the pty to know when characteristics of the line discipline in the pty are changed by an application on the slave side. The mechanism for handshaking trapped termio(4) requests (when TIOCTTY is enabled) is the same as that for non-termio(4) ioctl requests; except that any changes or error conditions set by the server on the master side will have no effect. (It is recommended that termio(4) ioctl requests be used on the master side to interrogate the configured state of the line discipline in the pty. One reason for this is to compensate for the window of time before TIOCMONITOR is enabled, when termio(4) ioctls were not trapped.)

When using *select*(2) on the master side of a pty, the "exceptional condition" refers to an open, close, or ioctl pending on the slave side. Ready for reading or writing refers to a read, or write pending respectively, from the point of view of the master side.

Of the ioctls that are subject to being trapped, only one per pty may be handled at one time. This means that when an application does a non-termio(4) ioctl to the slave side, all other ioctls to the same pty slave side will be blocked until the first one is handshaked back by the master side. (Ioctls that are not trapped, such as termio(4) when TIOCTTY is enabled and TIOCMONITOR is disabled, will not be blocked.) This permits the implementation of indivisible operations by an ioctl call on the slave side that is passed to the server process.

In summary, handshaking of an ioctl/open/close on the master side is done using the following steps:

Slave Side open/close/ioctl Trapped.
: This is indicated via a select(2) exceptional condition or via the TIOCTRAPSTATUS ioctl request.

TIOCREQGET ioctl request.
: This is done to find out what slave open/close/ioctl is trapped.

argget ioctl request.
: This optional ioctl is done if *argget* is nonzero and the server wants to do more than just reject the trapped slave ioctl.

argset ioctl request.
: This optional ioctl is done if *argset* is nonzero and the server wants to pass back a modified ioctl structure. It is done after the trapped ioctl is processed via the server on the master side.

TIOCREQSET ioctl request.
: This is done to complete the trapped slave open/close/ioctl. In case the trapped request is an ioctl, *errno_error* should be set appropriately. *return_value* should be set for trapped slave ioctls if *errno_error* is set to zero.

While a process is waiting in the slave side of the pty for the server to complete a handshake, it is susceptible to receiving signals. The following master side ioctl allows the server process to control how the pty will respond when a signal attempts to interrupt a trapped open or ioctl request.

TIOCSIGMODE
: Sets the signal handling state of the pty to the mode specified as the argument. The mode can have three values, which are TIOCSIGBLOCK, TIOCSIGABORT, and TIOCSIGNORMAL.

  TIOCSIGBLOCK
  : Causes some signals that are destined for the process whose open/ioctl is trapped to be postponed. The signals that are blocked are those which would otherwise cause the process to jump to an installed signal handler. Signals that are currently being ignored or would cause the slave-side process to be aborted will not be held off. When the server process completes the handshake by means of the TIOCREQSET ioctl request, the slave-side process will return to the calling program, and any pending signals will then be acted upon. Any signals that the user has blocked by means of *sigblock*(2) will continue to be blocked.

  TIOCSIGABORT
  : Forces all signals that interrupt a trapped open/ioctl request to not be restartable. The server process will set this mode when it wants the interrupted requests to return to the calling program with an EINTR error.

  TIOCSIGNORMAL
  : This is the default mode of the pty. If a signal interrupts a trapped open/ioctl request, the user's signal handler routine has the option of specifying whether the request is to be restarted. If the request is to be restarted, it will be executed

again from the begining, and the server will have to do another TIOCREQGET to start the handshake over again. If the user's signal handler routine specifies that the interrupted request is not to be restarted, then the request will return to the calling program with EINTR upon completion of the signal handler. Note that it is not guaranteed that the restarted request will be the very next one to be trapped.

HARDWARE DEPENDENCIES

Series 200:

The largest ioctl argument passable between master and slave sides is currently limited to 128 bytes.

Series 500:

The largest ioctl argument passable between master and slave sides is currently limited to 128 bytes.

The TIOCREMOTE mode is not currently implemented.

FILES

/dev/ptym/pty[pqrstuvw]*	master pseudo terminals
/dev/pty/tty[pqrstuvw]*	slave pseudo terminals

DIAGNOSTICS

None.

BUGS

It is not possible for the slave side to indicate an end–of–fle condition to the master side.

When using TIOCREMOTE, a single write to the master side of greater than 256 bytes may result in multiple smaller records being read from the slave side instead of only one record.

SEE ALSO

termio(4), ioctl(2), select(2), signal(2).

NAME

stty - terminal interface for Version 6/PWB compatability

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD — Version 6 and PWB Compatability

Origin: System V

Remarks: These faclilities are included to aid in conversion of old programs, and should not be used in new code. Use the interface described in *termio*(4). Note that these conversions do **not** work for programs ported from Version 7 UNIX, since some Version 7 flags are defined differently.

DESCRIPTION

These routines attempt to map the Version 6 and PWB *stty* and *gtty* calls into the current ioctls that perform the same functions. The mapping cannot be perfect. The way the features are translated is described below. The reader should be familiar with *termio*(4) before studying this page.

The following data structure is defined in the include file **sgtty.h**:

```
struct sgttyb {
        char      sg_ispeed;        /* input speed */
        char      sg_ospeed;        /* output speed */
        char      sg_erase;         /* erase character */
        char      sg_kill;          /* kill character */
        int       sg_flags;         /* mode flags */
}
```

The flags, as defined in **sgtty.h**, are:

```
#define      HUPCL      01
#define      XTABS      02
#define      LCASE      04
#define      ECHO       010
#define      CRMOD      020
#define      RAW        040
#define      ODDP       0100
#define      EVENP      0200
#define      ANYP       0300
#define      NLDELAY    001400
#define      TBDELAY    002000
#define      CRDELAY    030000
#define      VTDELAY    040000
#define      BSDELAY    0100000

#define      CR0        0
#define      CR1        010000
#define      CR2        020000
#define      CR3        030000
#define      NL0        0
#define      NL1        000400
#define      NL2        001000
#define      NL3        001400
#define      TAB0       0
#define      TAB1       002000
#define      NOAL       004000
#define      FF0        0
#define      FF1        040000
```

```
#define    BS0    0
#define    BS1    0100000
```

When the *stty*(2) command (*ioctl* **TIOCSETP**) is executed, the flags in the old **sgttyb** structure are mapped into their new equivalents in the **termio** structure. Then the **TCSETA** command is executed.

The following table shows the mapping between the old **sgttyb** flags and the current **termio** flags. Note that flags contained in the **termio** structure that are not mentioned below are cleared.

HUPCL (if set)
: sets the **termio** HUPCL flag;

HUPCL (if clear)
: clears the **termio** HUPCL flag;

XTABS (if set)
: sets the **termio** TAB3 flag;

XTABS (if clear)
: clears the **termio** TAB3 flag;

TBDELAY (if set)
: sets the **termio** TAB1 flag;

TBDELAY (if clear)
: clears the **termio** TAB1 flag;

LCASE (if set)
: sets the **termio** IUCLC, OLCUC, and XCASE flags;

LCASE (if clear)
: clears the **termio** IUCLC, OLCUC, and XCASE flags;

ECHO (if set)
: sets the **termio** ECHO flag;

ECHO (if clear)
: clears the **termio** ECHO flag;

NOAL (if set)
: clears the **termio** ECHOK flag;

NOAL (if clear)
: sets the **termio** ECHOK flag;

CRMOD (if set)
: sets the **termio** ICRNL and ONLCR flags; also, if CR1 is set, the **termio** CR1 flag is set, and if CR2 is set, the **termio** ONOCR and CR2 flags are set;

CRMOD (if clear)
: sets the **termio** ONLRET flag; also, if NL1 is set, the **termio** CR1 flag is set, and if NL2 is set, the **termio** CR2 flag is set;

RAW (if set)
: sets the **termio** CS8 flag, and clears the **termio** ICRNL and IUCLC flags; also, default values of 6 characters and 0.1 seconds are assigned to MIN and TIME, respectively;

RAW (if clear)
: sets the **termio** BRKINT, IGNPAR, ISTRIP, IXON, IXANY, OPOST, CS7, PARENB, ICANON, and ISIG flags; also, the default values control–D and null are assigned to the control characters EOF and EOL, respectively;

ODDP (if set)
: if EVENP is also set, clears the **termio** INPCK flag; otherwise, sets the **termio** PARODD flag;

VTDELAY (if set)
: sets the **termio** FFDLY flag;

VTDELAY (if clear)
: clears the **termio** FFDLY flag;

BSDELAY (if set)
    sets the **termio** BSDLY flag;
BSDELAY (if clear)
    clears the **termio** BSDLY flag.

In addition, the **termio** CREAD bit is set, and, if the baud rate is 110, the CSTOPB bit is set.

When using **TIOCSETP**, the *ispeed* entry in the **sgttyb** structure is mapped into the appropriate speed in the **termio** CBAUD field. The *erase* and *kill* **sgttyb** entries are mapped into the **termio** erase and kill characters.

When the *gtty*(2) (*ioctl* **TIOCGETP**) command is executed, the *termio*(4) **TCGETA** command is first executed. The resulting **termio** structure is then mapped into the **sgttyb** structure, which is then returned to the user.

The following table shows how the **termio** flags are mapped into the old **sgttyb** structure. Note that all flags contained in the **sgttyb** structure that are not mentioned below are cleared.

HUPCL (if set)
    sets the **sgttyb** HUPCL flag;
HUPCL (if clear)
    clears the **sgttyb** HUPCL flag;
ICANON (if set)
    clears the **sgttyb** RAW flag;
ICANON (if clear)
    sets the **sgttyb** RAW flag;
XCASE (if set)
    sets the **sgttyb** LCASE flag;
XCASE (if clear)
    clears the **sgttyb** LCASE flag;
ECHO (if set)
    sets the **sgttyb** ECHO flag;
ECHO (if clear)
    clears the **sgttyb** ECHO flag;
ECHOK (if set)
    clears the **sgttyb** NOAL flag;
ECHOK (if clear)
    sets the **sgttyb** NOAL flag;
PARODD (if set)
    sets the **sgttyb** ODDP flag;
PARODD (if clear)
    clears the **sgttyb** ODDP flag;
INPCK (if set)
    sets the **sgttyb** EVENP flag;
PARODD, INPCK (if both clear)
    sets the **sgttyb** ODDP and EVENP flags;
ONLCR (if set)
    sets the **sgttyb** CRMOD flag; also, if CR1 is set, the **sgttyb** CR1 flag is set, and if CR2 is set, the **sgttyb** CR2 flag is set;
ONLCR (if clear)
    if CR1 is set, the **sgttyb** NL1 flag is set, and if CR2 is set, the **sgttyb** NL2 flag is set;
TAB3 (if set)
    sets the **sgttyb** XTABS flag;
TAB3 (if clear)
    clears the **sgttyb** XTABS flag;
TAB1 (if set)
    sets the **sgttyb** TBDELAY flag;

TAB1 (if clear)
: clears the **sgttyb** TBDELAY flag;

FFDLY (if set)
: sets the **sgttyb** VTDELAY flag;

FFDLY (if clear)
: clears the **sgttyb** VTDELAY flag;

BSDLY (if set)
: sets the **sgttyb** BSDELAY flag;

BSDLY (if clear)
: clears the **sgttyb** BSDELAY flag.

When using **TIOCGETP**, the **termio** CBAUD field is mapped into the *ispeed* and *ospeed* entries of the **sgttyb** structure. Also, the **termio** erase and kill characters are mapped into the *erase* and *kill* **sgttyb** entries.

Note that, since there is not a one–to–one mapping between the **sgttyb** and **termio** structures, unexpected results may occur when using the older **TIOCSETP** and **TIOCGETP** calls. Thus, the **TIOCSETP** and **TIOCGETP** calls should be replaced in all future code by the current equivalents, **TCSETA** and **TCGETA**, respectively.

SEE ALSO

termio(4), stty(2).

NAME

termio – general terminal interface

HP-UX COMPATABILITY

Level: HP-UX/RUN ONLY

Origin: System V

DESCRIPTION

All of the asynchronous communications ports use the same general interface, no matter what hardware is involved. The remainder of this section discusses the common features of this interface.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by *getty* and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the *control terminal* for that process group. The control terminal plays a special role in handling quit and interrupt signals, as discussed below. The control terminal is inherited by a child process during a *fork*(2). A process can break this association by changing its process group using *setpgrp*(2).

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely full, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. This limit is dependent on the particular implementation, but is at least 256. When the input limit is reached, all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a new-line (ASCII LF) character, an end-of-file (ASCII EOT) character, or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. By default, the character # erases the last character typed, except that it will not erase beyond the beginning of the line. By default, the character @ kills (deletes) the entire input line, and optionally outputs a new-line character. Both these characters operate on a key-stroke basis, independently of any backspacing or tabbing that may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (\). In this case the escape character is not read. The erase and kill characters may be changed.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR	(Rubout or ASCII DEL) generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see *signal*(2).
QUIT	(Control-| or ASCII FS) generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory if the implementation supports core files.
ERASE	(#) erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.
KILL	(@) deletes the entire line, as delimited by a NL, EOF, or EOL character.

EOF	(Control-d or ASCII EOT) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOF is discarded. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.
NL	(ASCII LF) is the normal line delimiter. It can not be changed or escaped.
EOL	(ASCII NUL) is an additional line delimiter, like NL. It is not normally used.
STOP	(Control-s or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.
START	(Control-q or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can not be changed or escaped.

The character values for INTR, QUIT, ERASE, KILL, EOF, and EOL may be changed to suit individual tastes. The ERASE, KILL, and EOF characters may be escaped by a preceding \ character, in which case no special function is done.

When a modem disconnect is detected, a *hang-up* signal is sent to all processes that have this terminal as the control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hang-up signal is ignored, any subsequent read returns with an end-of-file indication. Thus, programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. If a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed.

Several *ioctl*(2) system calls apply to terminal files. The primary calls use the following structure, defined in <**termio.h**>:

```
#define  NCC       8
struct   termio {
         unsigned  short  c_iflag;     /* input modes */
         unsigned  short  c_oflag;     /* output modes */
         unsigned  short  c_cflag;     /* control modes */
         unsigned  short  c_lflag;     /* local modes */
         char             c_line;      /* line discipline */
         unsigned  char   c_cc[NCC];   /* control chars */
};
```

The special control characters are defined by the array *c_cc*. The relative positions and initial values for each function are as follows:

0	VINTR	DEL
1	VQUIT	FS
2	VERASE	#
3	VKILL	@
4	VEOF	EOT
5	VEOL	NUL
6	reserved	
7	reserved	

The *c_iflag* field describes the basic terminal input control:

IGNBRK	0000001	Ignore break condition.
BRKINT	0000002	Signal interrupt on break.
IGNPAR	0000004	Ignore characters with parity errors.
PARMRK	0000010	Mark parity errors.
INPCK	0000020	Enable input parity check.
ISTRIP	0000040	Strip character.
INLCR	0000100	Map NL to CR on input.
IGNCR	0000200	Ignore CR.
ICRNL	0000400	Map CR to NL on input.
IUCLC	0001000	Map upper-case to lower-case on input.
IXON	0002000	Enable start/stop output control.
IXANY	0004000	Enable any character to restart output.
IXOFF	0010000	Enable start/stop input control.
IENQAK	0020000	Enable output pacing control.

If IGNBRK is set, the break condition (a character framing error with data all zeros) is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise if BRKINT is set, the break condition will generate an interrupt signal and flush both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored.

If PARMRK is set, a character with a framing or parity error which is not ignored is read as the three-character sequence: 0377, 0, X, where X is the data of the character received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of 0377 is read as 0377, 0377. If PARMRK is not set, a framing or parity error which is not ignored is read as the character NUL (0).

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

If ISTRIP is set, valid input characters are first stripped to 7-bits, otherwise all 8-bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). Otherwise if ICRNL is set, a received CR character is translated into a NL character.

If IUCLC is set, a received upper-case alphabetic character is translated into the corresponding lower-case character.

If IXON is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. All start/stop characters are ignored and not read. If IXANY is set, any input character, will restart output which has been suspended.

If IXOFF is set, the system will transmit START/STOP characters when the input queue is nearly empty/full.

If IENQAK is set, the system will transmit ASCII ENQ after every 80 characters sent and then wait until the terminal responds with ASCII ACK. The terminal will respond in this way when it has sufficiently emptied its buffer. If the terminal does not respond after 5 seconds, the system will resume transmission anyway. The ASCII ACK that the terminal sends will not get entered into the input queue if it was sent in response to ASCII ENQ.

The initial input control value is all-bits-clear.

The *c_oflag* field specifies the system treatment of output:

OPOST	0000001	Postprocess output.
OLCUC	0000002	Map lower case to upper on output.
ONLCR	0000004	Map NL to CR-NL on output.
OCRNL	0000010	Map CR to NL on output.
ONOCR	0000020	No CR output at column 0.

ONLRET	0000040	NL performs CR function.
OFILL	0000100	Use fill characters for delay.
OFDEL	0000200	Fill is DEL, else NUL.
NLDLY	0000400	Select new-line delays:
NL0	0	
NL1	0000400	
CRDLY	0003000	Select carriage-return delays:
CR0	0	
CR1	0001000	
CR2	0002000	
CR3	0003000	
TABDLY	0014000	Select horizontal-tab delays:
TAB0	0	
TAB1	0004000	
TAB2	0010000	
TAB3	0014000	Expand tabs to spaces.
BSDLY	0020000	Select backspace delays:
BS0	0	
BS1	0020000	
VTDLY	0040000	Select vertical-tab delays:
VT0	0	
VT1	0040000	
FFDLY	0100000	Select form-feed delays:
FF0	0	
FF1	0100000	

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If OLCUC is set, a lower-case alphabetic character is transmitted as the corresponding upper-case character. This function is often used in conjunction with IUCLC.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If OCRNL is set, the CR character is transmitted as the NL character. If ONOCR is set, no CR character is transmitted when at column 0 (first position). If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay. If OFILL is set, fill characters will be transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character is DEL, otherwise NUL.

If a form-feed or vertical-tab delay is specified, it lasts for about 2 seconds.

New-line delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the new-line delays. If OFILL is set, two fill characters will be transmitted.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds. If OFILL is set, delay type 1 transmits two fill characters, and type 2, four fill characters.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about 0.10 seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, two fill characters will be transmitted for any delay.

Backspace delay lasts about 0.05 seconds. If OFILL is set, one fill character will be transmitted.

The actual delays depend on line speed and system load.

The initial output control value is all bits clear.

The *c_cflag* field describes the hardware control of the terminal:

CBAUD	0000037	Baud rate:
B0	0	Hang up
B50	0000001	50 baud
B75	0000002	75 baud
B110	0000003	110 baud
B134	0000004	134.5 baud
B150	0000005	150 baud
B200	0000006	200 baud
B300	0000007	300 baud
B600	0000010	600 baud
B900	0000011	900 baud
B1200	0000012	1200 baud
B1800	0000013	1800 baud
B2400	0000014	2400 baud
B3600	0000015	3600 baud
B4800	0000016	4800 baud
B7200	0000017	7200 baud
B9600	0000020	9600 baud
B19200	0000021	19200 baud
B38400	0000022	38400 baud
EXTA	0000036	External A
EXTB	0000037	External B
CSIZE	0000140	Character size:
CS5	0	5 bits
CS6	0000040	6 bits
CS7	0000100	7 bits
CS8	0000140	8 bits
CSTOPB	0000200	Send two stop bits, else one.
CREAD	0000400	Enable receiver.
PARENB	0001000	Parity enable.
PARODD	0002000	Odd parity, else even.
HUPCL	0004000	Hang up on last close.
CLOCAL	0010000	Local line, else dial-up.

The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the modem control lines (see *modem*(4)) will cease to be asserted. Normally, this will disconnect the line. For any particular hardware, impossible speed changes are ignored.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stops bits are required.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the PARODD flag specifies odd parity if set, otherwise even parity is used.

If CREAD is set, the receiver is enabled. Otherwise no characters will be received.

The specific effects of the HUPCL and CLOCAL bits depend on the mode and type of the modem control in effect. See *modem*(4) for the details.

If HUPCL is set, the modem control lines for the port will be disconnected when the last process with the port open closes it or terminates.

If CLOCAL is set, a connection does not depend on the state of the modem status lines.

Under normal circumstances, an *open* will wait for the type of modem connection requested to complete. However, if the O_NDELAY bit is specified (see *open*(2)) or the CLOCAL bit has been set, the *open* will return immediately without waiting for the connection. For those files on which the connection has not been established or has been lost, and for which the CLOCAL bit is not set, both *read* and *write* will return a zero character count. For *read*, this is equivalent to an end-of-file condition.

The initial hardware control value after open is B300, CS8, CREAD, HUPCL.

The *c_lflag* field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG	0000001	Enable signals.
ICANON	0000002	Canonical input (erase and kill processing).
XCASE	0000004	Canonical upper/lower presentation.
ECHO	0000010	Enable echo.
ECHOE	0000020	Echo erase character as BS-SP-BS.
ECHOK	0000040	Echo NL after kill character.
ECHONL	0000100	Echo NL.
NOFLSH	0000200	Disable flush after interrupt or quit.

If ISIG is set, each input character is checked against the special control characters INTR and QUIT. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g., 0377).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, read requests are satisfied directly from the input queue. A read will not be satisfied until at least MIN characters have been received or the timeout value TIME has expired between characters. This allows fast bursts of input to be read efficiently while still allowing single character input. The MIN and TIME values are stored in the position for the EOF and EOL characters, respectively. The time value represents tenths of seconds.

If XCASE is set, and if ICANON is set, an upper-case letter is accepted on input by preceding it with a \ character, and is output preceded by a \ character. In this mode, the following escape sequences are generated on output and accepted on input:

for:	*use*:
`	\'
|	\!
~	\^
{	\(
}	\)
\	\\

For example, **A** is input as **\a**, **\n** as **\\n**, and **\N** as **\\\n**.

If ECHO is set, characters are echoed as received.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which will clear the last character from a CRT screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the NL character will be echoed after the kill character to emphasize that the line will be deleted. Note that an escape character preceding the erase or kill character removes any special function. If ECHONL is set, the NL character will be echoed even if ECHO is not set. This is useful for terminals set to local echo (so-called half duplex). Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

If NOFLSH is set, the normal flush of the input and output queues associated with the quit and interrupt characters will not be done.

The initial line-discipline control value is all bits clear.

The primary *ioctl*(2) system calls have the form:

```
ioctl (fildes, command, arg)
struct termio *arg;
```

The commands using this form are:

TCGETA	Get the parameters associated with the terminal and store in the *termio* structure referenced by **arg**.
TCSETA	Set the parameters associated with the terminal from the structure referenced by **arg**. The change is immediate.
TCSETAW	Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.
TCSETAF	Wait for the output to drain, then flush the input queue and set the new parameters.

Additional *ioctl*(2) calls have the form:

```
ioctl (fildes, command, arg)
int arg;
```

The commands using this form are:

TCSBRK	Wait for the output to drain. If *arg* is 0, then send a break (zero bits for at least 0.25 seconds).
TCXONC	Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output.
TCFLSH	If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

HARDWARE DEPENDENCIES

Series 200/300:

Data loss may occur with HP 98626/98644 serial interfaces if the effective combined data rate for all installed serial interfaces exceeds 2400 baud (for example, two interfaces running at 1200 baud and a third at 300 baud is equivalent to 2700 baud combined).

The *c_iflag* field parameter IXANY (enable any character to restart output) is not supported by the HP 98628B interface card.

The *c_iflag* field parameter IENQAK (enable output pacing control) is not supported.

Timed delays are not supported.

The HP 98628B interface does not support the following baud rates: 900, 7200, 38400.

The *c_lflag* field parameter XCASE is not supported.

Series 500:

38400 baud is not supported by the RS-232 interface.

European modems are not currently supported.

HP27140 Six-Channel Modem Multiplexer:

Timed output delays (as opposed to fill-character delays) are not supported.

The XCASE flag is not supported.

These baud rates are not supported: 200, 38400, EXTA, and EXTB.

HP27128 Asynchronous Serial Interface, HP27130 Eight-Channel Multiplexer:

These baud rates are not supported: 200, 38400, EXTA, and EXTB.

There is no support for tab expansion, case mapping, or output delays for control characters.

The line kill character is always echoes as <backslash><CR><LF>, so the ECHOK flag is not setable, and will always have the same state as the ECHO flag.

When type-ahead limit is reached, input is not flushed, but further input is simply ignored.

The PARMRK flag is not supported.

The echoing of carriage-return and new-line characters may not be quite as expected in the more obscure driver configurations.

The echoing of the EOF character is not suppressed.

The ONLRET, ONOCR, and OCRNL flags are not supported.

The VMIN and VTIME parameters for raw terminal input are not supported.

The ECHONL flag is not supported.

When ECHOE is set and ECHO is clear, a <SP><BS> is not echoes for the erase character.

(27130 only) The CLOCAL flag is permanently set.

(27128 only) The default setting of baud rate, bits per character, parity, and CLOCAL bit are determined by the switches on the interface.

(27128 only) The "direct connect" cable (female connector) does not contain a Data Carrier Detect line, so a hangup signal will be sent if the CLOCAL flag is cleared when this cable is being used.

Model 520 Console, HP98700 Terminal, Pseudo Terminal (pty):

Since these devices do not deal with real asynchronous serial data links, the following flags are meaningless: IGNPAR, PARMRK, INPCK, IXOFF, IENQAK, CBAUD, CSIZE, CSTOPB, PARENB, PARODD, HUPCL, and CLOCAL.

FILES

/dev/tty*
/dev/console

SEE ALSO

stty(1), fork(2), ioctl(2), stty(2), setpgrp(2), signal(2), tty(4), modem(4), mknod(8).

HARDWARE DEPENDENCIES

Series 200/300:

Data loss may occur with the 98626/98644 if the effective input of all 98626/98644 cards exceeds 2400 baud.

The *c_iflag* field parameter IXANY (enable any character to restart output) is not supported by the HP 98628B interface card.

The *c_iflag* field parameter IENQAK (enable output pacing control) is not supported.

Timed delays are not supported.

The HP 98628B interface does not support the following baud rates: 900, 7200, 38400.

The *c_lflag* field parameter XCASE is not supported.

NAME

tty - controlling terminal interface

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V

DESCRIPTION

The file **/dev/tty** is, in each process, a synonym for the control terminal associated with the process group of that process, if any. It is useful for programs or shell sequences that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

FILES

/dev/tty
/dev/tty*

SEE ALSO

termio(4).

NAME

intro - introduction to file formats

HP-UX COMPATIBILITY

Remarks: Header files are often used to hide hardware incompatibilities.

DESCRIPTION

This section outlines the formats of various files. The C **struct** declarations for the file formats are given where applicable. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys**.

NAME

a.out – assembler and link editor output

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System III
Remarks:	This manual page describes the *a.out* file format for Series 200 and 300 computers. Refer to other *a.out*(5) manual pages for descriptions of other valid implementations.

DESCRIPTION

A.out is the output file of the link editor *ld*. *Ld* will make **a.out** executable if there were no linking errors and no unresolved external references. The assembler *as* produces non-executable files with the same structure.

File *a.out* has seven sections: a header, the program text and data segments, a pascal interface section, a symbol table, information for debugger support, and text and data relocation information (in that order). The pascal interface text will only be present in those pascal code segments that have not been linked. The last three sections may be missing if the program was linked with the **–s** option of *ld*(1) or if the symbol table, debug information, and relocation bits were removed by *strip*(1). Also note that if there were no unresolved external references after linking, the relocation information will be removed.

The file section containing information for debugger support has three tables – the debug name table (DNTT), the source line table (SLT), and the value table (VT). These tables contain symbolic information used by the HP-UX debugger *cdb*(1). HP-UX compilers create this information under control of the –g option.

When an **a.out** file is loaded into memory for execution, three logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized, the latter actually being initialized to all 0's), and a stack. The text segment begins at location 0x0 in the core image; the header is not loaded. If the magic number (the first field in the header) is EXEC_MAGIC, it indicates that the text segment is not to be write-protected or shared, so the data segment will be contiguous with the text segment. If the magic number is SHARE_MAGIC or DEMAND_MAGIC, the data segment begins at the first 0 mod 0x1000 byte boundary following the text segment, and the text segment is not writable by the program; if other processes are executing the same **a.out** file, they will share a single text segment. If the magic number is DEMAND_MAGIC, the text and data segments are not read in from the file until they are referenced by the program.

The stack will occupy the highest possible locations in the core image and grow downward (the stack is automatically extended as required). The data segment is only extended as requested by the *brk*(2) system call.

The start of the text segment in the **a.out** file is given by the macro TEXT_OFFSET(hdr), where hdr is a copy of the file header. The macro DATA_OFFSET(hdr) provides the starting location of the data segment.

The value of a word in the text or data portions that is not a reference to an undefined external symbol is exactly the value that will appear in memory when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation information (discussed below) for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added to the word in the file.

Header

The format of the **a.out** header for the MC68000 is as follows (segment sizes are in bytes):

```
struct exec {
        MAGIC   a_magic;        /* magic number */
        short   a_stamp;        /* version stamp */
        short   a_unused;
        long    a_sparehp;
        long    a_text;         /* size of text segment */
        long    a_data;         /* size of data segment */
        long    a_bss;          /* size of bss segment */
        long    a_trsize;       /* size of text relocation info */
        long    a_drsize;       /* size of data relocation info */
        long    a_pasint;       /* size of interface text */
        long    a_lesyms;       /* size of symbol table */
        long    a_dnttsize;     /* debug name table size */
        long    a_entry;        /* entry point of program */
        long    a_sltsize;      /* source-line table size */
        long    a_vtsize;       /* value table size */
        long    a_spare3;
        long    a_spare4;
};
```

Pascal Interface Section

The Pascal interface section consists of the ascii representation of the interface text for that Pascal module.

The start of the Pascal interface section is given by the macro MODCAL_OFFSET(hdr).

Symbol Table

The symbol table consists of entries of the form:

```
        struct nlist {
                long            n_value;
                unsigned char   n_type;
                unsigned char   n_length;
                short           n_almod;
                short           n_unused;
        };
```

Following this structure is *n_length* ascii characters which compose the symbol name.

The *n_type* field indicates the type of the symbol; the following values are possible:

00	undefined symbol
01	absolute symbol
02	text segment symbol
03	data segment symbol
04	bss segment symbol

One of these values ANDed with 040 indicates an external symbol. One of these values ANDed with 020 indicates an aligned symbol.

The start of the symbol table is given by the macro LESYM_OFFSET(hdr).

Relocation

If relocation information is present, it amounts to eight bytes per relocatable datum.

The format of the relocation data is:

```
struct  r_info      {
        long        r_address;
        short       r_symbolnum;
        char        r_segment;
        char        r_length;
};
```

The *r_address* field indicates the position of the relocation within the segment.

The *r_segment* field indicates the segment referred to by the text or data word associated with the relocation word:

00	indicates the reference is to the text segment;
01	indicates the reference is to initialized data;
02	indicates the reference is to bss (uninitialized data);
03	indicates the reference is to an undefined external symbol.

The field *r_symbolnum* contains a symbol number in the case of external references, and is unused otherwise. The first symbol is numbered 0, the second 1, etc.

The field *r_length* indicates the length of the datum to be relocated.

00	indicates it is a byte
01	indicates it is a short
02	indicates it is a long
03	indicates it is a special align symbol

The start of the text relocation section is provided by the macro RTEXT_OFFSET(hdr).

The start of the data relocation section is provided by the macro RDATA_OFFSET(hdr).

SEE ALSO

as(1), ld(1), nm(1), strip(1), magic(5).

NAME

a.out - executable linker output file

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	HP
Remarks:	This manual entry describes the *a.out* file format for the Series 500. Refer to other *a.out* manual pages for information valid for other implementations.

DESCRIPTION

A.out is the output file of the linker *ld*(1). *Ld* will make *a.out* executable if there are no errors during compilation and linking, and no unresolved external references.

This file has five sections - a file header, a segment table, a segment information section, a symbol table(s) section, and a name pool(s) section. It looks as follows:

File Header
Segment Table
Segment Information – segment image (code/data) – fix-up information (loader) – relocation information (*ld*)
Symbol Tables: – linker symbol table – information for debugger support
Name Pool (strings)

Note that the above pictorial representation represents the logical order of the file, not necessarily the physical order. A description of each section of the file follows.

File Header

The *a.out* file header is conceptually divided into two pieces. The first is a section of "scalar" values, and the second is a "file map" containing data pertaining to the rest of the file. The entire file header is made up of 128 bytes of information, 32 of which make up the scalar section. The following is a pictorial representation of the scalar section:

Byte	
0	System ID : File Type
4	Reserved for Future Use
8	Flags
12	Program Entry Point
16	Version Stamp
20	Memory Offset
24	Working Set Guarantee
28	Reserved for Future Use

Each horizontal "slice" represents a word made up of four eight-bit bytes. The first word is called the "magic number", which is made up of two half-words called the system ID and the file type. The system ID identifies the target machine upon which the object code will run. The file type specifies whether or not the file is executable (hex 107), shareable (hex 108), or relocatable (hex 106).

The third word is used to specify the settings of three flags. The left-most three bits of this word are significant; the remainder of the word is ignored. Bit 1, the left-most of the flag bits, marks the program as using a single data segment, if set. You can override this with the **-T** or **-A** *ld* options, which force the program to reside in one or two data segments, respectively. Bit 2 marks the file as relinkable, if set (meaning that the file contains relocation records and a symbol table). Bit 3 marks the file as debuggable, if set.

The *Program Entry Point* word contains an external program pointer (EPP) referencing the starting code for the program. *Ld* normally assigns the starting address of the main program to this word. This can be changed with the **-e** linker option.

The *Version Stamp* is a user-supplied 32-bit integer which is used to distinguish one version of an application program from another. The user can specify this integer using the **-V** *ld*(1) option at link time.

The file map portion of the header looks as follows:

Byte	
32	Code Segment Tbl: offset
36	Code Segment Tbl: size
40	Code Seg Images: offset
44	Code Seg Images: size
48	Data Segment Tbl: offset
52	Data Segment Tbl: size
56	Data Seg Images: offset
60	Data Seg Images: size
64	Link Symbol Tbl: offset
68	Link Symbol Tbl: size
72	DNTT: offset
76	DNTT: Size
80	VT: offset
84	VT: Size
88	SLT: offset
92	SLT: size
96	Name Pool: offset
100	Name Pool: size
104	Interface Info: offset
108	Interface Infor: size
112	Reserved for Future Use
116	Reserved for Future Use
120	Reserved for Future Use
124	Reserved for Future Use

Each *offset* entry in the file map shows where the given section starts, relative to the beginning of the *a.out* file. Each *size* entry gives the size, in bytes, for that section.

Segment Table

The segment table collects, in one place, all information about the code and data segments making up the program. The segment table consists of an array of entries. Each entry describes one code or data segment of the program.

The following information is given for both code and data segment table entries:

a *segment name*, which consists of an offset into the name pool, relative to the beginning of the name pool. This is useful for symbolically referring to code or data segments (not currently implemented).

a *segment type*, which specifies one of three possible types of segments - code, direct data (in GDS), or indirect data (in GDS or EDS).

a list of *segment attributes.* The segments can be paged, virtual, demand loadable, writable, and privileged. The linker sets the attributes for executable files.

a *segment offset*, which references a particular code or data segment within the segment image area. The reference is given relative to the beginning of the segment image area.

a *segment size*, which is the size, in bytes, of the particular code or data segment being described in the entry.

a *segment fixup size*, which specifies the size, in bytes, of the loader fixup area in the particular segment being described.

a *segment relocation information size*, which specifies the number of bytes of relocation records for this segment.

The following information is given for data segment table entries only:

a *segment limit*, which specifies the maximum number of bytes that the indirect data segment can contain. Attempting to increase the size beyond this stated limit results in an error. The linker assigns a default value of 1.5 megabytes to this field, but it may be changed with the **-m** *chatr*(1) option.

a *segment zero-padding size*, which is a byte count of the uninitialized data area. The linker computes this value from the data relocation records.

The following information is given for code segment table entries only:

a *segment local procedures count*, which specifies the number of procedures defined in that segment, but only known locally within it.

a *segment external procedures count*, which specifies the number of procedures defined in that segment, but externally known.

Several words are left unused in each segment table entry to allow for future growth.

Segment Information

This section of the file contains the segment images for each segment included in the final, executable file. This section contains a subsection for each program segment. Each subsection is in turn made up of three parts - the contents of the segment (code or data), a list of pointers that the loader must "fix up" in that segment, and the relocation records for that segment. Each subsection looks as follows:

Code/Data Image
Loader Initialization. Information
Loader Fixup Information
Relocation Records

The code image contains the compiled machine code for each program segment. The data image contains an image of initialized data for the program. Contained in this code are pointers. The loader fixup information area contains offsets that reference these pointers (the offsets are given relative to the beginning of the code/data image area). These offsets must be "fixed up" at run time (i.e., the program loader *exec* must update the segment number fields with the correct values). The linker generates the loader fixup information.

Symbol Tables

The linker symbol table contains data on relocatable symbols relevant to the linker (e.g. name and type for each global symbol). Refer to *nm*(1) (Series 500 only) for a complete description of each symbol type and the parameters associated with them. The contents of the symbol table may be listed in several different ways with *nm.*

Name Pool

The name pool contains a list of null–terminated strings, which specify the names of the symbols in the program. The symbol table entries contain indexes into the name pool instead of the names themselves. This permits arbitrarily long names to be used instead of fixed–length names. The first string in the name pool is always a null string. This enables zero to be used as an index into the name pool for entities which have no names.

SEE ALSO

chatr(1), ld(1), nm(1), strip(1), magic(5).

NAME

acct – per-process accounting file format

HP-UX COMPATIBILITY

Level: HP-UX/EXTENDED

Origin: System V

SYNOPSIS

#include <sys/acct.h>

HP-UX COMPATIBILITY

Level: HP-UX/EXTENDED

Origin: System V

DESCRIPTION

Files produced as a result of calling *acct*(2) have records in the form defined by **<sys/acct.h>**, whose contents are:

```
typedef ushort comp_t;              /* "floating point" */
                                    /* 13-bit fraction, 3-bit exponent  */

struct  acct
{
        char      ac_flag;          /* Accounting flag */
        char      ac_stat;          /* Exit status */
        ushort    ac_uid;           /* Accounting user ID */
        ushort    ac_gid;           /* Accounting group ID */
        dev_t     ac_tty;           /* control typewriter */
        time_t    ac_btime;         /* Beginning time */
        comp_t    ac_utime;         /* acctng user time in clock ticks */
        comp_t    ac_stime;         /* acctng system time in clock ticks */
        comp_t    ac_etime;         /* acctng elapsed time in clock ticks */
        comp_t    ac_mem;           /* memory usage in clicks */
        comp_t    ac_io;            /* chars trnsfrd by read/write */
        comp_t    ac_rw;            /* number of block reads/writes */
        char      ac_comm[8];       /* command name */
};

#define AFORK     01                /* has executed fork, but no exec */
#define ASU       02                /* used super-user privileges */
#define ACCTF     0300              /* record type: 00 = acct */
```

In *ac_flag*, the AFORK flag is turned on by each *fork*(2) and turned off by an *exec*(2). The *ac_comm* field is inherited from the parent process and is reset by any *exec*. Each time the system charges the process with a clock tick, it also adds to *ac_mem* the current process size, computed as follows:

(data size) + (text size) / (number of in-core processes using text) +
sum of ((shared memory segment size) / (number of in-core processes attached to segment))

The value of *ac_mem / (ac_stime + ac_utime)* can be viewed as an approximation to the mean process size, as modified by text-sharing.

The structure **tacct.h**, which resides with the source files of the accounting commands, represents the total accounting format used by the various accounting commands:

```
/*
 * total accounting (for acct period), also for day
 */

struct  tacct {
        uid_t           ta_uid;         /* userid */
        char            ta_name[8];     /* login name */
        float           ta_cpu[2];      /* cum. cpu time, p/np (mins) */
        float           ta_kcore[2];    /* cum kcore-minutes, p/np */
        float           ta_con[2];      /* cum. connect time, p/np, mins */
        float           ta_du;          /* cum. disk usage */
        long            ta_pc;          /* count of processes */
        unsigned short  ta_sc;          /* count of login sessions */
        unsigned short  ta_dc;          /* count of disk samples */
        short           ta_fee;         /* fee for special services */
};
```

HARDWARE DEPENDENCIES

Series 500:

ac_mem includes only certain resident segments still held by a process when it terminates. Because *ac_mem* does not account for shared or virtual memory, or for changes in the amount of memory allocated dynamically, *ac_mem* / (*ac_stime* + *ac_utime*) may not always furnish a good approximation of memory usage.

SEE ALSO

acct(2), exec(2), fork(2).
acct(1M), and acctcom(1) in the *HP-UX Reference.*

BUGS

The *ac_mem* value for a short-lived command gives little information about the actual size of the command, because *ac_mem* may be incremented while a different command (e.g., the shell) is being executed by the process.

NAME
ar - common archive file format

SYNOPSIS
#include <ar.h>

HP-UX COMPATIBILITY
Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION
Ar(1) is used to concatenate several files into an archival file. Archives are used mainly as libraries to be searched by the link editor *ld*(1).

Each archive begins with the archive magic string.

```
#define ARMAG   "!<arch>\n"         /* magic string */
#define SARMAG  8                   /* length of magic string */
```

Each archive which contains object files (see *a.out*(5)) includes an archive symbol table. This symbol table is used by the link editor *ld*(1) to determine which archive members must be loaded during the link edit process. The archive symbol table (if it exists) is always the first file in the archive (but is never listed) and is automatically created and/or updated by *ar*.

Following the archive magic string are the archive file members. Each file member is preceded by a file member header which is of the following format:

```
#define ARFMAG  "`\n"           /* header trailer string */

struct ar_hdr                   /* file member header */
{
    char   ar_name[16];         /* '/' terminated file member name */
    char   ar_date[12];         /* file member date */
    char   ar_uid[6];           /* file member user identification */
    char   ar_gid[6];           /* file member group identification */
    char   ar_mode[8];          /* file member mode (octal) */
    char   ar_size[10];         /* file member size */
    char   ar_fmag[2];          /* header trailer string */
};
```

All information in the file member headers is in printable ASCII. The numeric information contained in the headers is stored as decimal numbers (except for *ar_mode* which is in octal). Thus, if the archive contains printable files, the archive itself is printable.

The *ar_name* field is blank-padded and slash (/) terminated. The *ar_date* field is the modification date of the file at the time of its insertion into the archive. Common format archives can be moved from system to system as long as the portable archive command *ar*(1) is used. Note that older versions or *ar*(1) did not use the common archive format, and those archives cannot be read or written by the common archiver. The conversion tool *arcv*(1) is provided for changing non-common format archives to this format.

Each archive file member begins on an even byte boundary; a newline is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

If the archive symbol table exists, the first file in the archive has a zero length name (i.e., **ar_name[0]** == '/'). The contents of this archive member are machine dependent. Further details can be found in the *ranlib*(5) manual page.

SEE ALSO

ar(1), arcv(1), ld(1), strip(1), ranlib(5).

WARNING

Strip(1) will remove all archive symbol entries from the header. The archive symbol entries must be restored via the **ts** option of the *ar*(1) command before the archive can be used with the link editor *ld*(1).

NAME

BIF - Bell Interchange Format utilities

HP–UX COMPATIBILITY

Level:	Bell System III – HP-UX/NUCLEUS
Origin:	HP

DESCRIPTION

BIF (Bell Interchange Format) is the name given to the format of mounted media used by HP–UX series 200 revisions 2.0 and 2.1. This format is based upon that used in System III Unix.

These utilities are provided for reading and writing data to and from BIF volumes. These utilities (referred to hereafter as *bif**(1)) may be used to retrieve and store information on a BIF volume.

The *bif**(1) utilities are the only utilities within HP–UX where the internal contents of a BIF volume are known. To the rest of HP–UX a BIF volume is simply a file/disk containing some unspecified data. You must not use mount(1) on a BIF volume, since the operating system does not recognize it.

BIF file names are specified to the *bif**(1) utilities by concatenating the HP–UX path name for the BIF volume with the BIF file name, separating the two with a colon (:). For example,

/dev/fd.0:/users/ivy	specifies BIF file /users/ivy within HP–UX device special file /dev/fd.0.

Note that this file naming convention is applicable only for use as arguments to the *bif**(1) utilities and does not constitute a legal path name for any other use within HP–UX. The shell sh(1) "meta" characters: * ? and [...] do not work for specifying an arbitrary pattern for file name matching when using the BIF utilities.

If the device name and a trailing colon are specified without a file or directory name following (e.g. /dev/rfd.0:), then the root (/) of the BIF file system is assumed by convention.

A primitive form of data protection is provided by a lockfile /tmp/BIF..LCK that only allows one process and it's immediate children to use the bif utilities at a time.

SEE ALSO

bifchmod(1), bifchown(1), bifcp(1), bifdf(1), biffind(1), biffsck(1), biffsdb(1), bifls(1), bifmkdir(1), bifmkfs(1), bifrm(1).

NAME

checklist - static information about the file systems.

SYNOPSIS

#include <checklist.h>

HP-UX COMPATIBILITY

Level:	Large Machine/HP Extension
Origin:	HP, System V and UCB

DESCRIPTION

Checklist is an ASCII file and resides in directory **/etc.** It is only read by programs, and not written; it is the duty of the system administrator to properly create and maintain this file. */etc/checklist* contains a list of mountable file system entries. The fields within each entry of a file system are separated by one or more blanks. Each file system entry is contained on a separate line. The order of entries in */etc/checklist* is important because *fsck*, *mount*, and *umount* sequentially iterate through */etc/checklist.*

Each file system entry must contain a **special file name** and may additionally contain all of the following fields, in order:

block special file name
directory
type
pass number on parallel fsck
backup frequency
comment

These additional fields are ignored in an HP-UX system if the set of system administration tools implemented on that system does not support them.

The *special file name* is either a character or block special file name. This field is used by the fsck(1M) command.

The *block special file name* is used by the mount(1M) and other commands.

The *directory* is the name of the root of the mounted file system which corresponds to the block special file name. The *directory* must already exist and must be given as an absolute path name.

Type can be "rw", "ro", "sw" or "xx". If *type* is "rw" or "ro" then the file system whose name is given in the *block special file* field is mounted read-write or read-only on the specified *directory* by **mount -a**. If *type* is "sw" then the *special file name* is made available as a piece of swap space by the *swapon*(1M) command. The fields *pass number* and *backup frequency* are ignored for "sw" entries. Entries marked "xx" are ignored by all commands and can be used to mark unused sections. If *type* is specified as either "xx" or "sw" the entry is ignored by the *mount*(1M) command.

The *pass number* field is used by the *fsck*(1M) command to determine the order in which file system checks are done when using the -p option of *fsck.* The root file system should be specified with a *pass number* of 1, and other file systems should have larger numbers. File systems within a drive should have distinct numbers, but file systems on different drives can be checked on the same pass to utilize possible parallelism available in the hardware. A file system with a *pass number* of zero will be ignored by the fsck(1M) command. If a pass number is not present, fsck will check each such file system sequentially after all eligible file systems with pass numbers have been checked.

The *backup frequency* field is reserved for possible use by future backup utilities.

The *comment* field is an optional field which starts with a pound sign (#) and ends with a newline. Space from the *backup frequency* up to the comment field, if present, or the newline is reserved for future use.

Examples of file system entries specified in /etc/checklist:

For system which supports only special file name field:

```
        /dev/rdsk/0s0
```

For system which supports multi-fields:

```
    /dev/rdsk/0s0  /dev/dsk/0s0  /  rw  1  0  #root disc
```

HARDWARE DEPENDENCIES

Series 500:

All of the optional fields in a file system entry will be ignored.

Series 200 and 300

There is no limit to the number of special file names in */etc/checklist.* However, the commands *mount −a* and *umount −a* give an error if the number of mountable file system entries in */etc/checklist* exceeds NMOUNT.

SEE ALSO

fsck(1M), getfsent(3X), mount(1M), swapon(1M).

NAME

col_seq_8 – Collating sequence table for languages with 8-bit character sets

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

Native Language Support:
8-bit data, customs

DESCRIPTION

There are four language dependent collation algorithms for European languages. These algorithms are:

2–to–1 Conversions: Some languages, like Spanish, require two adjacent characters to occupy one position in the collating sequence. Examples are "CH" (which follows "C") and "LL" (which follows "L").

1–to–2 Conversions: Some languages, like German, require one character (e.g. "sharp S") to occupy two adjacent positions in the collating sequence.

Don't care Characters: Some languages designate certain characters to be ignored in character comparisons. For example, if "–" is a "Don't Care" character, then the strings "REACT" and "RE–ACT" would equal each other when compared.

Case and Accent Priority: Many languages require a "two pass" collating algorithm: in pass one, the accents are stripped off the letters and the resulting two strings are compared; if they are equal, a second pass with the accents back in place is performed to break the tie. The case of letters may also be used in this fashion.

This table has four sections – a file header, a sequence table, a 2–to–1 mapping table and a 1–to–2 mapping table.

Header
Sequence Table
2–to–1 Mapping Table
1–to–2 Mapping Table

Length and pointers are in units of two bytes.

Header:

	Byte 0	Byte 1
0	Table Length	
2	Language Id Number	
4	Reserved	
6	Pointer to Sequence Table	
8	Length of Sequence Table	
10	Pointer to 2–to–1 Mapping Table	
12	Length of 2–to–1 Mapping Table	
14	Pointer to 1–to–2 Mapping Table	
16	Length of 1–to–2 Mapping Table	
18	Lowest Char	Highest Char
20	Reserved	

Sequence Table:

Sequence Entry 0
Sequence Entry 1
(other entries from 2–254)
Sequence Entry 255

The byte value of a character is used as an index into the sequence table.

Sequence Entry Format: Each entry in the sequence table above uses two bytes and has one of the following formats:

First Byte – Bits: 15–8	Second Byte – 7-6	– 5-4-3-2-1-0	Format Type
0	00	0	don't–care characters
sequence no.	00	priority	all 1–to–1 mapped characters w/o priority
sequence no.	01	index	2–to–1 mapped characters
seq # (1.ch)	10	index	1–to–2 mapped characters

The 6–bit index indexes into either the 2–to–1 or the 1–to–2 mapping table.

Mapping Table for 2–to–1 Mapped Characters

2–to–1 Mapping Table
Entry Pointer 1
Entry Pointer 2
(other entry pointers)
Entry Pointer n

Sequence Entry Format for Mapped Pairs		
Byte 0	Byte 1	
0	Legal Char 1	
Sequence Entry for this Pair		
(other mapped pair entries)		
0	Legal Char n	
Sequence Entry for This Pair		
Sentinel: –1		
0		priority

The "legal" 2–to–1 characters are listed for each particular character. "Legal" means that the combination of two characters is treated as a single character. If a match is found, then the corresponding sequence entry is used for the two. Whenever a legal successor is not found in the table, the character is treated according to 1–to–1 mapping, and the priority in the last entry, combined with sequence number of the character, creates the sequence entry.

Mapping Table for 1–to–2 Mapped Characters

1–to–2 Mapping Table
Sequence Entry
Sequence Entry
(other sequence entries)
Sequence Entry

Entries in the 1–to–2 mapping table have the same format as entries in the sequence table. The sequence number of the first character is known from the entry in the sequence table. The sequence number of the second character is found in the 1–to–2 mapping entry, and the priority is used for both characters.

SEE ALSO

sort(1), nl_string(3C).

NAME

core – format of core image file

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD – Assembly option

Origin: System III

DESCRIPTION

The HP-UX system writes out a core image of a terminated process when any of various errors occur. See *signal*(2) for the list of reasons; the most common are memory violations, illegal instructions, floating point exceptions, and user-generated quit signals. The core image is called **core** and is written in the process's working directory (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID will not produce a core image.

The first section of the core image is a header which contains information about the terminated process. The remainder represents the actual contents of the user's core area when the core image was written. This area contains the stack, user global data, and heap segments. The last object in the core image is the code segment fixup map which maps user code segments into real addresses.

The format of the information in the first section is described by the *user* structure of the system, defined in **<sys/user.h>**.

SEE ALSO

cdb(1), setuid(2), signal(2).

NAME

cpio - format of cpio archive

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

Remarks: Not supported on the Integral Personal Computer.

DESCRIPTION

The *header* structure, when the **-c** option of *cpio*(1) is not used, is:

```
struct {
        short   h_magic,
                h_dev;
        ushort  h_ino,
                h_mode,
                h_uid,
                h_gid;
        short   h_nlink,
                h_rdev,
                h_mtime[2],
                h_namesize,
                h_filesize[2];
        char    h_name[h_namesize rounded to word];
} Hdr;
```

When the **-c** option is used, the *header* information is described by:

```
        sscanf(Chdr,"%6ho%6ho%6ho%6ho%6ho%6ho%6ho%6ho%11lo%6ho%11lo",
            &Hdr.h_magic,&Hdr.h_dev,&Hdr.h_ino,&Hdr.h_mode,
            &Hdr.h_uid,&Hdr.h_gid,&Hdr.h_nlink,&Hdr.h_rdev,
            &Longtime,&Hdr.h_namesize,&Longfile);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file is recorded together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated path name *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Directories and the trailer are recorded with *h_filesize* equal to zero.

It will not always be the case that h_dev and h_ino correspond to the results of *stat(2),* but the values are always sufficient to tell whether two files in the archive are linked to each other.

When a device special file is archived by HP-UX *cpio* (using -x), h_rdev will contain a magic constant which is dependent upon the implementation which is doing the writing. *H_rdev* flags the device file as an HP-UX 32-bit device specifier, and h_filesize will contain the 32-bit device specifier (see *stat(2)*).

Special files are not restored, and cpio(1) generates a warning, if either h_filesize is zero, h_rdev is non-zero, or the identifying information is not that for the restoring system. If the -x option is not present, special files are not archived or restored. Non-HP-UX device special files are never restored.

SEE ALSO

cpio(1), find(1), stat(2).

NAME

dialups, d_passwd - dialup security control

HP-UX COMPATIBILITY

Level:	HP-UX/STANDARD
Origin:	Capability from System V, page by HP.

DESCRIPTION

Dialups and *d_passwd* are used to control the dialup security feature of *login*(1). If */etc/dialups* is present, the first word on each line is compared with the name of the line upon which the login is being performed. (Including the **/dev/**, as returned by *ttyname*(3)). If the login is occurring on a line found in *dialups*, dialup security is invoked. Anything after a space or tab is ignored.

When dialup security is invoked, *login*(1) will request an additional password, and check it against that found in */etc/d_passwd.* The command name found in the "program to use as Shell" field of */etc/passwd* is used to select the password to be used. Each entry in *d_passwd* consists of three fields, separated by colons. The first is the command name, matching an entry in *passwd.* The second is the encrypted password to be used for dialup security for those users logging in to use that program. The third is commentary, but the second colon is required to delimit the end of the password. A null password is designated with two adjacent colons. The entry for */bin/sh* is used if no other entry matches the command name taken from *passwd.*

FILES

/etc/dialups	Dial in tty lines
/etc/d_passwd	Passwords

SEE ALSO

login(1), passwd(5).

NAME

dir - format of directories

SYNOPSIS

#include <types.h>
#include <sys/dir.h>

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: UCB and HP

Remarks: This entry describes the directory format for the HFS file system. Refer to other *dir*(5) manual pages for information valid for other implementations.

DESCRIPTION

A directory behaves exactly like an ordinary file, except that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry (see *fs*(5)). The structure of a directory entry as given in the **dir.h** include file is:

```
#define DIRSIZ          14
#define DIR_PADSIZE     10

struct  direct {
        u_long          d_ino;               /* inode number of entry */
        u_short         d_reclen;            /* length of this record */
        u_short         d_namlen;            /* length of string in d_name */
        char            d_name[DIRSIZ];      /* name must be no longer than this */
        char            d_pad[DIR_PADSIZE];
};
```

By convention, the first two entries in each directory are for **.** and **..** ("dot" and "dot dot"). The first is an entry for the directory itself. The second is for the parent directory. The meaning of **..** is modified for the root directory of the master file system; there is no parent, so **..** and **.** have the same meaning.

The *direct* structure defined here is the actual directory format for the HFS file system and is not compatible with other HP-UX supported file systems. The *direct* structure defined in */usr/include/ndir.h* should be used in conjunction with the directory(3C) library routines for compatibility across all HP-UX supported file systems.

SEE ALSO

fs(5), directory(3C).

NAME

dir - format of directories

SYNOPSIS

#include <types.h>
#include <sys/dir.h>

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

Remarks: This entry describes the SDF directory format for Series 500. Refer to other *dir*(5) manual pages for information valid for other implementations.

DESCRIPTION

A directory behaves exactly like an ordinary file, except that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry (see *inode*(5)). The structure of a directory entry as given in **sys/dir.h** is:

```
#ifndef DIRSIZ
#define DIRSIZ          14
#endif
struct  direct
{
        char            d_name[DIRSIZ+2];   /* 16-char file name */
        short           d_object_type;      /* not referenced by HP-UX */
        short           d_file_code;        /* not referenced by HP-UX */
        ino_t           d_ino;              /* use fir # for i-node */
};
```

The SDF directory implementation eliminates entries for **.** and **..**. Instead, this information is available as part of the i-node.

File names are stored in directories in a special manner in two cases:

When a file name contains **embedded blanks**, the blanks are represented by null characters on the disc. This is apparent when accessing the disc in raw (character) mode.

When a file name is **blank padded**, all unspecified characters are set to blanks. Again, this is apparent only when reading from the disc in raw mode.

When a director has been opened vi *open*(2), file names appear as null-terminated, and contain embedded blanks where they belong.

The *direct* structure defined here is the actual directory format for the SDF file system, and is not compatible with other file systems supported on HP-UX. The *direct* structure defined in */usr/include/ndir.h* should be used in conjunction with the directory(3C) library routines for compatibility across all HP-UX supported file systems.

SEE ALSO

fs(5), inode(5), directory(3C).

NAME

dir - format of directories

SYNOPSIS

#include <types.h>
#include <sys/dir.h>

HP-UX COMPATIBILITY

Level:	HP-UX/STANDARD
Origin:	System III
Remarks:	This entry describes the directory format for the Bell file system. Refer to other *dir*(5) manual pages for information valid for other implementations.

DESCRIPTION

A directory behaves exactly like an ordinary file, except that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry (see *fs*(5)). The structure of a directory entry as given in the **dir.h** include file is:

```
#ifndef DIRSIZ
#define DIRSIZ          14
#endif
struct  direct
{
        ino_t           d_ino;
        char            d_name[DIRSIZ];
};
```

By convention, the first two entries in each directory are for . and .. ("dot" and "dot dot"). The first is an entry for the directory itself. The second is for the parent directory. The meaning of .. is modified for the root directory of the master file system; there is no parent, so .. and . have the same meaning.

The *direct* structure defined here is the actual directory format for the Bell file system and is not compatible with other HP-UX supported file systems. The *direct* structure defined in */usr/include/ndir.h* should be used in conjunction with the directory(3C) library routines for compatibility across all HP-UX supported file systems.

SEE ALSO

fs(5), directory(3C).

NAME

disktab - disc description file

SYNOPSIS

#include <disktab.h>

HP-UX COMPATIBILITY

Level:	Large Machine/HP Extension/HFS
Origin:	HP and UCB

DESCRIPTION

Disktab is a simple data base which describes disc geometries and disc section characteristics. Entries in *disktab* consist of a number of ':' separated fields. The first entry for each disc gives the names which are known for the disc, separated by '|' characters. The last name given should be a long name fully identifying the disc.

The following list indicates the normal values stored for each disc entry. Sectors are of size DEV_BSIZE, defined in <sys/param.h> on your system.

Name	**Type**	**Description**
ns	num	Number of sectors per track
nt	num	Number of tracks per cylinder
nc	num	Total number of cylinders on the disc
b0	num	Block size for section '0' (bytes)
b1	num	Block size for section '1' (bytes)
b<n>	num	Block size for section '<n>' (bytes)
f0	num	Fragment size for section '0' (bytes)
f1	num	Fragment size for section '1' (bytes)
f<n>	num	Fragment size for section '<n>' (bytes)
s0	num	Size of section '0' in sectors
s1	num	Size of section '1' in sectors
s<n>	num	Size of section '<n>' in sectors
rm	num	Revolution per minute
ty	str	Type of disc (e.g. removable, winchester)

Example:

```
hp7914:     :ty=winchester:ns#16:nt#7:nc#1061:s0#118832\
            :b0#8192:f0#1024:rm#3600:
```

HARDWARE DEPENDENCIES

Series 200:
The Series 200 SM HP-UX 5.0 release can have only one section per disc drive.

FILES

/etc/disktab

SEE ALSO

newfs(1M)

NAME

errfile - system error logging file

HP-UX COMPATIBILITY

Level:	HP-UX/STANDARD
Origin:	HP
Remarks:	This manual page describes *errfile* as implemented on the Series 500. Refer to other *errfile* manual pages for information valid for other implementations.

DESCRIPTION

Errfile is a logging file containing lines of ASCII text. Each line describes certain system errors that have occurred, or warnings about serious system conditions. Only those system error messages deemed serious enough to be of interest to the system administrator are logged. Urgent messages are also written to /dev/console.

HP-UX creates *errfile* if it does not exist.

The system administrator should check the contents of *errfile* periodically and note errors that need attention. Also, *errfile* tends to grow without bounds, so outdated information needs to be removed on a regular basis.

FILES

usr/adm/errfile

NAME

fs - format of file system volume

SYNOPSIS

#include <sys/types.h>
#include <sys/param.h>
#include <sys/fs.h>
#include <sys/ino.h>
#include <sys/inode.h>
#include <sys/sysmacros.h>

HP-UX COMPATIBILITY

Level: Large Machine/HP Extension/HFS
Origin: HP and UCB 4.2

DESCRIPTION

Every file system storage volume has a common format for certain vital information. The first 8 kbytes on a volume contain a volume header which identifies that volume as a LIF volume. Such volume may be divided into a number of sections.

Each section can contain a file system. The first 8 kbytes in each section is ignored, except where it coincides with the volume header discussed above. The actual file system begins next with the *super block.* The layout of the super block as defined by the include file *<sys/fs.h>* is:

```
#define FS_MAGIC    0x011954
#define FS_CLEAN    0x17
#define FS_OK            0x53
#define FS_NOTOK    0x31
struct  fs {
        struct  fs *fs_link;            /* linked list of file systems */
        struct  fs *fs_rlink;           /*     used for incore super blocks */
        daddr_t         fs_sblkno;          /* addr of super-block in filesys */
        daddr_t         fs_cblkno;          /* offset of cyl-block in filesys */
        daddr_t         fs_iblkno;          /* offset of inode-blocks in filesys */
        daddr_t         fs_dblkno;          /* offset of first data after cg */
        long    fs_cgoffset;            /* cylinder group offset in cylinder */
        long    fs_cgmask;              /* used to calc mod fs_ntrak */
        time_t fs_time;                 /* last time written */
        long    fs_size;        /* number of blocks in fs */
        long    fs_dsize;               /* number of data blocks in fs */
        long    fs_ncg;                 /* number of cylinder groups */
        long    fs_bsize;               /* size of basic blocks in fs */
        long    fs_fsize;               /* size of frag blocks in fs */
        long    fs_frag;        /* number of frags in a block in fs */
/* these are configuration parameters */
        long    fs_minfree;             /* minimum percentage of free blocks */
        long    fs_rotdelay;            /* num of ms for optimal next block */
        long    fs_rps;                 /* disk revolutions per second */
/* these fields can be computed from the others */
        long    fs_bmask;               /* "blkoff" calc of blk offsets */
        long    fs_fmask;               /* "fragoff" calc of frag offsets */
        long    fs_bshift;              /* "lblkno" calc of logical blkno */
        long    fs_fshift;              /* "numfrags" calc number of frags */
/* these are configuration parameters */
        long    fs_maxcontig;           /* max number of contiguous blks */
        long    fs_maxbpg;              /* max number of blks per cyl group */
/* these fields can be computed from the others */
```

```
        long    fs_fragshift;           /* block to frag shift */
        long    fs_fsbtodb;             /* fsbtodb and dbtofsb shift constant */
        long    fs_sbsize;              /* actual size of super block */
        long    fs_csmask;              /* csum block offset */
        long    fs_csshift;             /* csum block number */
        long    fs_nindir;              /* value of NINDIR */
        long    fs_inopb;               /* value of INOPB */
        long    fs_nspf;        /* value of NSPF */
        long    fs_sparecon[6];         /* reserved for future constants */
/* sizes determined by number of cylinder groups and their sizes */
        daddr_t fs_csaddr;              /* blk addr of cyl grp summary area */
        long    fs_cssize;              /* size of cyl grp summary area */
        long    fs_cgsize;              /* cylinder group size */
/* these fields should be derived from the hardware */
        long    fs_ntrak;               /* tracks per cylinder */
        long    fs_nsect;               /* sectors per track */
        long    fs_spc;                 /* sectors per cylinder */
/* this comes from the disk driver partitioning */
        long    fs_ncyl;                /* cylinders in file system */
/* these fields can be computed from the others */
        long    fs_cpg;                 /* cylinders per group */
        long    fs_ipg;                 /* i-nodes per group */
        long    fs_fpg;                 /* blocks per group * fs_frag */
/* this data must be re-computed after crashes */
        struct  csum fs_cstotal;        /* cylinder summary information */
/* these fields are cleared at mount time */
        char    fs_fmod;                /* super block modified flag */
        char    fs_clean;               /* file system is clean flag */
        char    fs_ronly;               /* mounted read-only flag */
        char    fs_flags;               /* currently unused flag */
        char    fs_fsmnt[MAXMNTLEN];            /* name mounted on */
/* these fields retain the current block allocation info */
        long    fs_cgrotor;             /* last cg searched */
        struct  csum *fs_csp[MAXCSBUFS];/* list of fs_cs info buffers */
        long    fs_cpc;                 /* cyl per cycle in postbl */
        short   fs_postbl[MAXCPG][NRPOS];/* head of blocks for each rotation */
        long    fs_magic;               /* magic number */
        char    fs_name[6];             /* name of file system */
        char    fs_fpack[6];            /* pack name of file system */
        u_char fs_rotbl[1];             /* list of blocks for each rotation */
/* actually longer */
};
```

A file system consists of a number of cylinder groups. Each cylinder group has i-nodes and data.

A file system is described by its super-block, which in turn describes the cylinder groups. The super-block is critical data and is replicated in each cylinder group to protect against catastrophic loss. This is done at file system creation time and the critical super-block data does not change, so the copies need not be referenced further unless disaster strikes.

Addresses stored in i-nodes are capable of addressing fragments of 'blocks'. File system blocks of at most size MAXBSIZE can be optionally broken into smaller pieces, each of which is addressable; these pieces may be DEV_BSIZE, or some multiple of a DEV_BSIZE unit.

Large files consist of exclusively large data blocks. To avoid undue wasted disk space, the last data block of a file is allocated only as many fragments of a large block as are necessary, if that file is small enough to not require indirect data blocks. The file system format retains only a single pointer to such a fragment, which is a piece of a single large block that has been divided. The size of such a fragment is determinable from information in the i-node, using the "blksize(fs, ip, lbn)" macro.

The file system records space availability at the fragment level; to determine block availability, aligned fragments are examined.

I-numbers begin at 0. I-nodes 0 and 1 are reserved. I-node 2 is used for the root directory of the file system. The *lost+found* directory is given the next available inode when it is initially created by *mkfs*.

fs_minfree gives the minimum acceptable percentage of file system blocks which may be free. If the freelist drops below this level only the super-user may continue to allocate blocks. This may be set to 0 if no reserve of free blocks is deemed necessary, however severe performance degradations will be observed if the file system is run at greater than 90% full; thus the default value of *fs_minfree* is 10%.

The best trade-off between block fragmentation and overall disk utilization and performance varies for each intended use of the file system. Suggested values can be found in the System Administrator's Manual for each implementation.

Cylinder group related limits: Each cylinder keeps track of the availability of blocks at different rotational positions, so that sequential blocks can be laid out with minimum rotational latency. NRPOS is the number of rotational positions which are distinguished. For example, with NRPOS 8 the resolution of the summary information is 2ms for a typical 3600 rpm drive.

fs_rotdelay gives the minimum number of milliseconds to initiate another disk transfer on the same cylinder. It is used in determining the rotationally optimal layout for disk blocks within a file; the default value for *fs_rotdelay* is 2ms. Suggested values of fs_rotdelay for different disks can be found in the System Administrator's Manual.

Each file system has a statically allocated number of i-nodes. An i-node is allocated for each NBPI bytes of disk space. The i-node allocation strategy is extremely conservative.

MAXIPG bounds the number of i-nodes per cylinder group, and is needed only to keep the structure simpler by having only a single variable size element (the free bit map).

N.B.: MAXIPG must be a multiple of INOPB(fs).

MINBSIZE is the smallest allowable block size. With a MINBSIZE of 4096 it is possible to create files of size 2^32 with only two levels of indirection. MINBSIZE must be big enough to hold a cylinder group block, thus MINBSIZE must always be greater than sizeof(struct cg). Note that super blocks are never more than size SBSIZE.

The path name on which the file system is mounted is maintained in *fs_fsmnt*. MAXMNTLEN defines the amount of space allocated in the super block for this name. The limit on the amount of summary information per file system is defined by MAXCSBUFS. It is currently parameterized for a maximum of two million cylinders.

Per cylinder group information is summarized in blocks allocated from the first cylinder group's data blocks. These blocks are read in from *fs_csaddr* (size *fs_cssize*) in addition to the super block.

N.B.: sizeof (struct csum) must be a power of two in order for the "fs_cs" macro to work.

Super block for a file system: MAXBPC bounds the size of the rotational layout tables and is limited by the fact that the super block is of size SBSIZE. The size of these tables is **inversely** proportional to the block size of the file system. The size of the tables is increased when sector sizes are not powers of two, as this increases the number of cylinders included before the rotational pattern repeats (*fs_cpc*). The size of the rotational layout tables is derived from the number of bytes remaining in (struct fs).

MAXBPG bounds the number of blocks of data per cylinder group, and is limited by the fact that cylinder groups are at most one block. The size of the free block table is derived from the size of blocks and the number of remaining bytes in the cylinder group structure (struct cg).

Inode: The i–node is the focus of all file activity in the HP–UX file system. There is a unique i–node allocated for each active file, each current directory, each mounted–on file, text file, and the root. An i–node is 'named' by its device/i–number pair. For the format of an i–node and its flags, see *inode(5)*

HARDWARE DEPENDENCIES

Series 200:
Series 200 HP–UX 5.0 release supports only one section per volume. Thus, there can only be one file system on each volume and the first 8 kbytes of a file system is the boot area. This area contains the LIF volume header, the directory that defines the contents of the volume and the bootstrapping program.

HFS file structure is not implemented on Series 500 or Integral PC.

SEE ALSO

lif(1), HP–UX System Administrator's Manual.

NAME
fs - format of system volume

SYNOPSIS
#include <sys/param.h>
#include <sys/filsys.h>

HP-UX COMPATIBILITY

Level:	HP-UX/STANDARD
Origin:	HP
Remarks:	This manual page describes the format of the system volume as implemented on the Series 500. Refer to other *fs* manual pages for information valid for other implementations.

DESCRIPTION
Every Structured Directory Format (SDF) volume is divided into logical blocks, the size of which is selected when *init* is executed. Block 0 is the superblock. It has the following format:

```
struct filsys {
        ushort      s_format;       /* disc fmt, should = 0x700 Unix */
        ushort      s_corrupt;      /* non-zero if directory corrupt */
        char        s_fname[16];    /* root dir name, blank padded */
        time_ios    s_init;         /* date initialized / unique id */
        int         s_blksz;        /* no. bytes per block */
        daddr_t     s_boot;         /* boot area starting block */
        int         s_bootsz;       /* size of boot area in blks */
        daddr_t     s_fa;           /* FA file starting block */
        int         s_version;      /* version no., 0 for Unix */
        daddr_t     s_maxblk;       /* largest addressable blk */
        char        s_passwd[16];   /* volume password, Unix unused */
        time_ios    s_bkup;         /* last backup date, Unix unused */
                                    /* rest of blk unused */
};
```

The file attributes file (FA file) begins at the block specified by *s_fa* in the superblock. It has five major sections:

Each entry consists of 128 bytes. Entry 0 is the i-node of the FA file itself (see *inode*(5) for a description of the i-node structure). Entry 1 is the i-node for the file system's root directory, /.

Entry 3 through entry n consists of the free map, which keeps track of every free (unused) block of memory on the device. The free map contains a bit for each block on the device. If a bit is set, the corresponding block of memory is free; otherwise, the corresponding block is being used. The free map is zero-padded to guarantee that it ends on a 128-byte boundary.

Entry n+1 through the end of the FA file contains an entry for every file in the system. Each entry is either an i-node, an extent map, or unused. An extent map contains 128 bytes of information, and looks as follows:

```
struct em_rec {
        ushort      e_type;      /* =2 for extent maps */
        ushort      e_exnum;     /* # extents in this rec. */
        int         e_res1;      /* unused */
        ino_t       e_next;      /* next map in list; none = neg */
        ino_t       e_last;      /* last map in list; none = neg */
        ino_t       e_inode;     /* owner i-node no. */
        daddr_t     e_boffset;   /* blk offset of 1st extent from start of file */
        struct {
                daddr_t     e_startblk;     /* extent start blk */
                int         e_numblk;       /* # blks in extent */
        }       e_extent[13];
};
```

FILES

/usr/include/sys/param.h
/usr/include/sys/filsys.h
/usr/include/sys/ino.h

SEE ALSO

inode(5), fsck(1M).

NAME

fspec - format specification in text files

HP-UX COMPATIBILITY

Level:	HP-UX/STANDARD
Origin:	System V
Remarks:	Not supported on the Integral Personal Computer.

DESCRIPTION

It is sometimes convenient to maintain text files on the HP-UX system with non-standard tabs, (i.e., tabs which are not set at every eighth column). Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by HP-UX system commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets **<:** and **:>**. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* The **t** parameter specifies the tab settings for the file. The value of *tabs* must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;
2. a - followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. a **-** followed by the name of a "canned" tab specification.

Standard tabs are specified by **t-8**, or equivalently, **t1,9,17,25,**etc. The canned tabs which are recognized are defined by the *tabs*(1) command.

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

m*margin* The **m** parameter specifies a number of spaces to be prepended to each line. The value of *margin* must be an integer.

d The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are **t-8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file. The following is an example of a line containing a format specification:

```
* <:t5,10,15 s72:> *
```

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

Several HP-UX system commands have been specifically structured so they can correctly interpret file format specifications.

HARDWARE DEPENDENCIES

Series 500:

Some earlier-design interface cards do not handle tab expansion correctly. This can cause unexpected results.

SEE ALSO

ed(1), newform(1), tabs(1).

BUGS

Does not work with *vi*(1) and *ex*(1).

NAME

gettydefs - speed and terminal settings used by getty

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

The **/etc/gettydefs** file contains information used by *getty*(1M) to set up the speed and terminal settings for a line. It supplies information on what the *login* prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a <*break*> character.

Each entry in **/etc/gettydefs** has the following format:

label# initial-flags # final-flags #login-prompt#next-label

The pound sign (#) is the field separator for lines in **gettydefs**. The spaces must appear as shown. Each entry is followed by a blank line. Lines that begin with # are ignored and may be used to comment the file. The various fields can contain quoted characters of the form **\b**, **\n**, **\c**, etc., as well as \nnn, where *nnn* is the octal value of the desired character. The various fields are:

label This is the string against which *getty* tries to match its second argument. It is often the speed, such as **1200**, at which the terminal is supposed to run, but it need not be (see below).

initial-flags These flags are the initial *ioctl*(2) settings to which the terminal is to be set. The flags that *getty* understands are the same as the ones listed in */usr/include/termio.h* (see *tty*(4)). Normally only the speed flag is required in the *initial-flags*. *Getty* automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until *getty* executes *login*(1).

final-flags These flags take the same values as the *initial-flags* and are set just prior to when *getty* executes *login*. The speed flag is again required. The composite flag **SANE** takes care of most of the other flags that need to be set so that the processor and terminal are communicating in a rational fashion. The other two commonly specified *final-flags* are **TAB3**, so that tabs are sent to the terminal as spaces, and **HUPCL**, so that the line is hung up on the final close.

login-prompt This entire field is printed as the *login-prompt*. Unlike the above fields where white space is ignored (a space, tab or new-line), they are included in the *login-prompt* field. Thus, it is important that *only* the characters making up the login prompt be included between the #'s in this field, with no extra white space.

next-label If this entry does not specify the desired speed, indicated by the user typing a <*break*> character, then *getty* searches for the entry with *next-label* as its *label* field and sets up the terminal for those settings. Usually, a series of speeds are linked together in this fashion, into a closed set: for instance, **2400** linked to **1200**, which in turn is linked to **300**, which finally is linked to **2400**.

If *getty* is called without a second argument, then the first entry of **/etc/gettydefs** is used, thus making the first entry of **/etc/gettydefs** the default entry. It is also used if *getty* cannot find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into the command which brings up a terminal at **300** baud.

It is strongly recommended that **/etc/gettydefs** be run through *getty* with the check option to be sure there are no errors.

EXAMPLE

The following two lines show an example of a 300/1200 baud toggle, which is useful for dial–up ports:

```
1200# B1200 HUPCL # B1200 SANE IXANY IXANY TAB3 #login: #300
300# B300 HUPCL # B300 SANE IXANY IXANY TAB3 #login: #1200
```

The following line shows a typical 9600 baud entry for a hard–wired connection:

```
9600# B9600 # B9600 SANE IXANY IXANY ECHOE TAB3 #login: #9600
```

FILES

/etc/gettydefs

SEE ALSO

login(1), ioctl(2), tty(4), getty(1M).

NAME

group - group file, grp.h

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: System III

DESCRIPTION

Group contains for each group the following information:

group name
encrypted password
numerical group ID
comma-separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; each group is separated from the next by a new-line. If the password field is null, no password is associated with the group.

There are two files of this form in the system, /etc/group and /etc/logingroup. /etc/group exists to supply names for each group, and to support changing groups via *newgrp*(1). /etc/logingroup provides a default group access list for each user via *login*(1) and *initgroups*(3c).

The real and effective group ID set up by *login* for each user is defined in /etc/passwd (see *passwd*(5). If /etc/logingroup is empty or non-existent, the default group access list is limited to this effective group ID. If /etc/logingroup and /etc/group are links to the same file, the default access list includes the entire set of groups associated with the user. The group name and password fields in /etc/logingroup are never used; they are included only to give the two files a uniform format, allowing them to be linked together.

All group ID's used in /etc/logingroup or /etc/passwd should be defined in /etc/group. No user should be associated with more than **NGROUPS** (see *setgroups*(2)) groups in /etc/logingroup.

These files reside in directory **/etc**. Because of the encrypted passwords, they can and do have general read permission and can be used, for example, to map numerical group ID's to names.

Grp.h describes the group structure returned by getgrent(3), etc:

```
struct      group {          /* see getgrent(3) */
            char        *gr_name;
            char        *gr_passwd;
            int         gr_gid;
            char        **gr_mem;
};
```

FILES

/etc/group /etc/logingroup

SEE ALSO

groups(1), newgrp(1), passwd(1), setgroups(2), crypt(3C), getgrent(3), initgroups(3c), passwd(5).

BUGS

There is no tool that helps you ensure that /etc/passwd, /etc/group, and /etc/logingroup are compatible.

There is no tool that helps you set group passwords in /etc/group.

NAME

inittab - script for the init process

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System V
Remarks:	Not supported on the Integral Personal Computer.

DESCRIPTION

The *inittab* file supplies the script to *init*'s role as a general process dispatcher. The process that constitutes the majority of *init*'s process dispatching activities is the line process **/etc/getty** that initiates individual terminal lines. Other processes typically dispatched by *init* are daemons and the shell.

The *inittab* file is composed of entries that are position dependent and have the following format:

id:rstate:action:process

Each entry is delimited by a newline, however, a backslash (\) preceding a newline indicates a continuation of the entry. Up to 512 characters per entry are permitted. Comments may be inserted in the *process* field using the *sh*(1) convention for comments. Comments for lines that spawn *getty*s are displayed by the *who*(1) command. It is expected that they will contain some information about the line such as the location. There are no limits (other than maximum entry size) imposed on the number of entries within the *inittab* file. The entry fields are:

id This is one or two characters used to uniquely identify an entry.

rstate This defines the *run-level* in which this entry is to be processed. *Run-levels* effectively correspond to a configuration of processes in the system. That is, each process spawned by *init* is assigned a *run-level* or *run-levels* in which it is allowed to exist. The *run-levels* are represented by a number ranging from **0** through **6**. As an example, if the system is in *run-level* **1**, only those entries having a **1** in the *rstate* field will be processed. When *init* is requested to change *run-levels,* all processes which do not have an entry in the *rstate* field for the target *run-level* will be sent the warning signal (**SIGTERM**) and allowed a 20-second grace period before being forcibly terminated by a kill signal (**SIGKILL**). The *rstate* field can define multiple *run-levels* for a process by selecting more than one *run-level* in any combination from **0-6**. If no *run-level* is specified, then the process is assumed to be valid at all *run-levels* **0-6**. There are three other values, **a**, **b** and **c**, which can appear in the *rstate* field, even though they are not true *run-levels*. Entries which have these characters in the *rstate* field are processed only when the *telinit* (see *init*(1M)) process requests them to be run (regardless of the current *run-level* of the system). They differ from *run-levels* in that *init* can never enter *run-level* **a**, **b** or **c**. Also, a request for the execution of any of these processes does not change the current *run-level*. Furthermore, a process started by an **a**, **b** or **c** command is not killed when *init* changes levels. They are only killed if their line in **/etc/inittab** is marked **off** in the *action* field, their line is deleted entirely from **/etc/inittab**, or *init* goes into the *SINGLE USER* state.

action Key words in this field tell *init* how to treat the process specified in the *process* field. The actions recognized by *init* are as follows:

respawn If the process does not exist then start the process, do not wait for its termination (continue scanning the *inittab* file), and when it dies restart the process. If the process currently exists then do nothing and continue scanning the *inittab* file.

wait Upon *init*'s entering the *run-level* that matches the entry's *rstate*, start the process and wait for its termination. All subsequent reads of the *inittab* file while *init* is in the same *run-level* will cause *init* to ignore this entry.

once Upon *init*'s entering a *run-level* that matches the entry's *rstate*, start the process, do not wait for its termination. When it dies, do not restart the process. If upon entering a new *run-level*, where the process is still running from a previous *run-level* change, the program will not be restarted.

boot The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *Init* is to start the process, not wait for its termination; and when it dies, not restart the process. In order for this instruction to be meaningful, the *rstate* should be the default or it must match *init*'s *run-level* at boot time. This action is useful for an initialization function following a hardware reboot of the system.

bootwait The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *Init* is to start the process, wait for its termination and, when it dies, not restart the process.

powerfail Execute the process associated with this entry only when *init* receives a power fail signal (**SIGPWR** see *signal*(2)).

powerwait Execute the process associated with this entry only when *init* receives a power fail signal (**SIGPWR**) and wait until it terminates before continuing any processing of *inittab*.

off If the process associated with this entry is currently running, send the warning signal (**SIGTERM**) and wait 20 seconds before forcibly terminating the process via the kill signal (**SIGKILL**). If the process is nonexistent, ignore the entry.

ondemand This instruction is really a synonym for the **respawn** action. It is functionally identical to **respawn** but is given a different keyword in order to divorce its association with *run-levels*. This is used only with the **a**, **b** or **c** values described in the *rstate* field.

initdefault An entry with this *action* is only scanned when *init* initially invoked. *Init* uses this entry, if it exists, to determine which *run-level* to enter initially. It does this by taking the highest *run-level* specified in the **rstate** field and using that as its initial state. If the *rstate* field is empty, this is interpreted as **0123456** and so *init* will enter *run-level* **6**. Additionally, if *init* does not find an **initdefault** entry in **/etc/inittab**, then it will request an initial *run-level* from the user at reboot time.

sysinit Entries of this type are executed before *init* tries to access the console. It is expected that this entry will be only used to initialize devices on which *init* might try to ask the *run-level* question. These entries are executed and waited for before continuing.

process This is a *sh* command to be executed. The entire **process** field is prefixed with *exec* and passed to a forked *sh* as **sh -c 'exec** *command'*. For this reason, any legal *sh* syntax can appear in the *process* field. Comments can be inserted with the **;** #*comment* syntax.

FILES

/etc/inittab

SEE ALSO

sh(1), who(1), getty(1M), init(1M), exec(2), open(2)), signal(2).

NAME

inode - format of an i-node

SYNOPSIS

#include <sys/types.h>
#include ,sys/inode.h>
#include <sys/ino.h>

HP-UX COMPATIBILITY

Level: Large Machine/HP Extension/HFS

Origin: HFS

Remarks: This entry describes the i-node structure for the HFS file system. Refer to other *inode*(5) manual pages for information valid for other implementations.

DESCRIPTION

An i-node for a plain file or directory in a file system has the following structure defined by **<sys/ino.h>** and **<sys/inode.h>**.

```
/* Inode structure as it appears on a disk block */

struct  dinode
{
        u_short    di_mode;           /* mode and type of file */
        short      di_nlink;          /* number of links to file */
        short      di_uid;            /* owner's user id */
        short      di_gid;            /* owner's group id */
        quad       di_size;           /* number of bytes in file */
        time_t     di_atime;          /* time last accessed */
        long       di_atspare;
        time_t     di_mtime;          /* time last modified */
        long       di_mtspare;
        time_t     di_ctime;          /* time of last file status change */
        long       di_ctspare;
        daddr_t    di_db[NDADDR];     /* disk block addresses */
        daddr_t    di_ib[NIADDR];     /* indirect blocks */
        long       di_flags;          /* status, currently unused */
        long       di_blocks;         /* blocks actually held */
        long       di_spare[5];       /* reserved, currently unused */
};
```

For the meaning of the defined types *u_short, quad, daddr_t* and *time_t* see *types*(7).

See */usr/include/sys/inode.h* for the definition of i-node structures for special files, pipes, or FIFO's.

FILES

/usr/include/sys/ino.h

SEE ALSO

stat(2), fs(5), types(7).

NAME

inode - format of an i–node

SYNOPSIS

#include <sys/types.h>
#include <sys/param.h>
#include <sys/ino.h>

HP–UX COMPATIBILITY

Level:	HP–UX/STANDARD
Origin:	HP
Remarks:	This entry describes the i–node structure for the Series 500. Refer to other *inode* manual pages for information valid for other implementations.

DESCRIPTION

An i–node for an ordinary file or directory in a file system has the following structure, as defined in **sys/ino.h**:

```
/*
* I–node structure as it appears on disc. This i–node is actually
* a file information record (FIR) in the HP SDF disc format.
*/

struct dinode {
        ushort      di_type         /* =1 for inodes */
        ushort      di_ftype;       /* file type */
        ushort      di_count;       /* reference count */
        short       di_uftype;      /* user file type (LIF) */
        time_ios    di_ctime;       /* time created */
        unsigned    di_other;       /* public capabilities */
        ino_t       di_protect;     /* file protect rec. none=–1 */
        ino_t       di_label;       /* file label rec. none=–1 */
        int         di_blksz;       /* file size in blocks */
        int         di_max;         * largest byte writable */
        ushort      di_exsz;        /* recom. extent size */
        ushort      di_exnum;       /* no. i–node extents (1–4) */
        struct {
                    daddr_t         di_startblk;/* extent start blk */
                    int             di_numblk;/* no. blks in extent */
        } di_extent[4];
        ino_t       di_exmap;       /* inode 1st extent map */
                                    /* none = –1 */
        int         di_size;        /* current size, bytes */

        /* Warning! Next 2 fields apply only to directories */

        ino_t       di_parent;      /* inode of parent */
        char        di_name[16];    /* name of this directory */

        /* The remaining fields defined only for local */
        /* implementation of structured directory format.    */

        time_t      di_atime;       /* time last accessed */
        time_ios    di_mtime;       /* time last mod. */
        int         di_recsz;       /* logical record size */
        ushort      di_uid;         /* owner's user id */
        ushort      di_gid;         /* owner's group id */
        ushort      di_mode;        /* mode, type of file */
```

```
                char        di_res2[2];      /* unused */
                /* The next field used only if file is */
                /* a device file; otherwise it is zero */
                dev_t       di_dev;          /* description of device */
        };
```

The meaning of the type declarations included above can be found in *types*(7).

FILES

/usr/include/sys/ino.h

SEE ALSO

dir(5), fs(5), types(7).

NAME

issue - issue identification file

HP-UX COMPATIBILITY

Level:	HP-UX/NUCLEUS
Origin:	System V

DESCRIPTION

The file **/etc/issue** contains the *issue* or project identification to be printed as a login prompt. This is an ASCII file which is read by program *getty* and then written to any terminal spawned or respawned from the *lines* file.

FILES

/etc/issue

SEE ALSO

getti(1m), login(1).

NAME

LIF - Logical Interchange Format description

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: HP

DESCRIPTION

LIF (Logical Interchange Format) is a Hewlett-Packard standard disc format that may be used for interchange of files among various HP computer systems. A LIF volume contains a header (identifying it as a LIF volume) and a directory that defines the contents (i.e. files) of the volume. The size of the directory is fixed when the volume is initialized (see *lifinit*(1)) and sets an upper bound on the number of files that may be created on the volume.

HP-UX contains a set of utilities (referred to hereafter as *lif**(1)) that may be used to initialize a LIF volume (i.e. create a header and an empty directory), copy files to and from LIF volumes, list the contents of LIF volumes, remove LIF files, and rename LIF files.

The *lif**(1) utilities are the only utilities within HP-UX where the internal structure of a LIF volume is known. To the rest of HP-UX a LIF volume is simply a file containing some unspecified data. The term 'LIF volume' should in no way be confused with the HP-UX notion of a file system volume or mountable volume.

The LIF utility on HP-UX currently supports three file types, ASCII(1), BINARY(-2) and BIN(-23951).

There are three copying modes associated with them.

ASCII If the copying mode is ASCII, and an HP-UX file is being copied to a LIF volume, the utility strips the trailing LF and prepends two bytes of record length to each record. These records are then written to a LIF formated media. When copying a LIF ASCII file to HP-UX the two byte record length are stripped and a trailing LF is appended. These records are then written to the destination. In this mode of copying the length of the file is preserved. The default file type for this mode of copying is ASCII(1).

BINARY
If the copying mode is BINARY, and an HP-UX file is being copied to a LIF volume, the utility simply appends two bytes for record length to each 1k byte record. A trailing fractional block will have a count reflecting the number of bytes in that block. No interpretation is placed on the content of the records. These records are then written to a LIF formated media. When copying a LIF file to an HP-UX file in BINARY copying mode, the record lengths are stripped and the content of records is directly written to the destination. In this mode of copying the length of the binary file is preserved. The default file type for this mode of copying is BINARY(-2).

RAW If the copying mode is RAW, and an HP-UX file is being copied to a LIF volume, the utility simply copies the raw data to the destination. File sizes which are not multiples of 256 bytes will be padded with nulls to the next higher multiple. Therefore, the file sizes are not preserved. When copying a LIF file to an HP-UX file in RAW mode, the information is directly copied without any interpretation placed on the content of the source. The default file type for this mode of copying is BIN(-23951).

A LIF volume may be created on any HP-UX file (either regular disc file or device special file) that supports random access via *lseek*(2). Note that you should **not** mount the special file before using the *lif**(1) routines. See *lifinit*(1) for details. Within a LIF volume, individual files are identified by 1 to 10 character file names. File names may consist of upper-case alphanumeric characters (A through Z, 0 through 9) and the underscore character (_). The first character of a LIF file name must be a letter. The *lif**(1) utilities will accept any file name, including illegal file names generated on other systems, but will only create legal names. For example, file names containing lower-case letters will be read but not created.

LIF file names are specified to the *lif**(1) utilities by concatenating the HP-UX path name for the LIF volume with the LIF file name, separating the two with a colon (:). For example,

/dev/fd.0:ABC	specifies LIF file ABC within HP-UX device special file /dev/fd.0.
myfile:ABC	specifies LIF file ABC within HP-UX disc file 'myfile'.

Note that this file naming convention is applicable only for use as arguments to the *lif**(1) utilities and do not constitute legal path names for any other use within HP-UX.

HARDWARE DEPENDENCIES

Series 500:

You **must** use a character special file to access the media.

SEE ALSO

lifcp(1), lifinit(1), lifls(1), lifrename(1), lifrm(1).

NAME

magic - magic numbers for HP–UX implementations

SYNOPSIS

#include <magic.h>

HP–UX COMPATIBILITY

Level:	Use: HP–UX/RUN ONLY Header: HP–UX/DEVELOPMENT
Origin:	HP

DESCRIPTION

Magic.h localizes all information about HP–UX "magic numbers" in one file, and thus facilitates uniform treatment of magic numbers. This file specifies the location of the magic number in a file (always the start of the file) and the structure of the magic number:

```
struct magic_number
  {
   unsigned short system_id;
   unsigned short file_type;
  };
typedef struct magic_number MAGIC;
```

Magic.h includes definitions for the system IDs of all HP machines running HP-UX, and file types that are common to all implementations. There may be additional implementation–dependent file types. The predefined file types are:

```
/* for object code files */
#define RELOC_MAGIC   0x106   /* relocatable only */
#define EXEC_MAGIC    0x107   /* normal executable */
#define SHARE_MAGIC   0x108   /* shared executable */
```

HARDWARE DEPENDENCIES

Series 200:

The following additional file type is defined:

```
#define DEMAND_MAGIC      0x10B
```

SEE ALSO

ar(1), chatr(1), ld(1), a.out(5), ar(5), model(5).

BUGS

Cpio files use a different form of magic number that is incompatible with *magic*(5).

NAME

master - master device information table

HP–UX COMPATIBILITY

Level:	Config(1M) Support — HP–UX/RUN ONLY
Origin:	System V and HP

DESCRIPTION

This file is used by *config*(1M) to obtain device information that enables it to generate the configuration file. *Master* contains lines of various *Master* contains lines of various forms rame– ters.

Software drivers are defined as follows:

Field 1: device name, used in the user–specified dfile (8 chars maximum)

Field 2: handler name, used by the kernel to prefix routines such as cs80_read, lp_write, ...

Field 3: element characteristics: 5 bits make up the mask

Bit 1 – card
Bit 2 – specified only once
Bit 3 – required driver
Bit 4 – block device
Bit 5 – character device

Field 4: functions for the device: 10 bits make up the mask

Bit 1 – size handler
Bit 2 – link routine
Bit 3 – open handler
Bit 4 – close handler
Bit 5 – read handler
Bit 6 – write handler
Bit 7 – ioctl handler
Bit 8 – select handler
Bit 9 – seltru handler
Bit 10 – C_ALLCLOSES flag

Field 5: major device number, if a block–type device

Field 6: major device number, if a character–type device

Aliases for names are defined as follows:

Field 1: alias name => product number
Field 2: device name

Parameters are defined as follows:

Field 1: parameter name, as used in the user–specified dfile
Field 2: parameter name, as used in the #define statement in conf.c
Field 3: parameter value

SEE ALSO

config(1M)

NAME

mknod - create a special file entry

SYNOPSIS

#include <mknod.h>

HP-UX COMPATIBILITY

Level:	HP-UX/STANDARD
Origin:	HP

DESCRIPTION

Mknod.h provides utilities to pack and unpack device names as used by *mknod*(2). It contains the macro **dev = makedev(major, minor)** which packs the major and minor fields into a form suitable for *mknod*(2). It also contains **major(dev)** and **minor(dev)** which extract the corresponding fields.

The macro MINOR_FORMAT is a *printf* specification that prints the minor field in the format best suited to the particular implementation. The specification given by MINOR_FORMAT must cause the resulting string to indicate the base of the number in the same format as that used for C: no leading zero for decimal, leading zero for octal, and leading zero and 'x' for hexadecimal.

When a minor field is printed in the format specified by MINOR_FORMAT, each sub-field contained in the minor will be wholly contained in the mininum possible number of digits of the resulting string. (Splitting a field across unnecessary digits for the sake of packing is not done.)

SEE ALSO

mknod(1M), mknod(2), section 4.

WARNING

All of the macros defined in <mknod.h> are also defined in <sys/sysmacros.h> for Bell System V compatibility. Mknod.h only exists for compatibility with previous releases of HP-UX, and should not be used for new development.

NAME

mnttab - mounted file system table

SYNOPSIS

#include <sys/types.h>
#include <mnttab.h>

HP-UX COMPATIBILITY

Level: Large Machine/SVID

Origin: System V

DESCRIPTION

Mnttab resides in directory **/etc** and contains a table of devices, mounted by the *mount*(1M) command, in the following structure as defined by **<mnttab.h>**:

```
struct  mnttab {
        char    mt_dev[MNTLEN];
        char    mt_filsys[MNTLEN];
        short   mt_ro_flg;
        time_t  mt_time;
};
```

Each entry is (2 x MNTLEN + 6) bytes in length (MNTLEN is defined in /usr/include/mnttab.h). The first MNTLEN bytes are the null-padded name of the place where the *special file* is mounted; the next MNTLEN bytes represent the null-padded root name of the mounted special file; the remaining 6 bytes contain the mounted *special file*'s read/write permissions and the date on which it was mounted. The minimum value for MNTLEN is 32.

The maximum number of entries in *mnttab* is based on the system parameter **NMOUNT** located in /usr/include/mnttab.h, which defines the number of allowable mounted special files.

WARNING

The table is present only for programs to return information about the mounted file systems. It does not matter to *mount* if there are duplicated entries nor to *umount* if a name cannot be found.

SEE ALSO

mount(1M), setmnt(1M).

NAME

model - HP–UX machine identification

HP–UX COMPATIBILITY

Level: HP–UX/RUN ONLY

Origin: HP

SYNOPSIS

#include <model.h>

DESCRIPTION

There are some distinctions between the implementations of HP-UX due to hardware differences. Where such distinctions exist, conditional compilation or other definitions can be used to isolate the differences. Flags and typedefs to resolve these distinctions are collected in *model.h.* This file contains constants indentifying various HP-UX implementations.

For example, the header file *model.h* contains the constants **HP_S_200** for Series 200 and **HP_S_500** for Series 500. Other such constants will be added as HP–UX extends to other machines.

Model.h also has a statement defining the preprocessor constant *MYSYS* to represent the specific implementation for which compilation is desired. *MYSYS* will be equal to one of the constants above (**HP_S_200** or **HP_S_500**).

Conditional compilation may be used to adapt one file for execution on more than one HP–UX implementation, if it contains implementation– or architecture–dependent features. For instance,

```
#if MYSYS==HP_S_200
        <statements>
#endif
```

will cause the statements following the if statement to be compiled only for the HP 9000 Series 200.

Model.h also contains typedefs for several predefined types to enhance portability of certain types of code and of files.

int8
u_int8
Signed and unsigned 8–bit integers.

int16
u_int16
Signed and unsigned 16–bit integers.

int32
u_int32
Signed and unsigned 32–bit integers.

machptr
u_machptr
Signed and unsigned integers large enough to hold a pointer.

HARDWARE DEPENDENCIES

Series 200:
A conditional compilation variable, **hp9000s200**, is implemented. It is predefined to the C preprocessor.

Series 500:
A conditional compilation variable, **hp9000s500**, is implemented. It is predefined to the C preprocessor.

SEE ALSO

cc(1), cpp(1), magic(5).

NAME

nlist - nlist structure format

SYNOPSIS

#include <nlist.h>

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: System III

Remarks: The exact content of the structure defined below can be best found by examining */usr/include/nlist.h*. It varies somewhat between the various implementations of HP-UX.

Nlist is currently implemented on the Series 200 and Integral PC only.

DESCRIPTION

Nlist(3) can be used to extract information from a the symbol table in an object file. Because symbol tables are machine dependent (as defined in each implementation's copy of **<a.out.h>**) a header file, *nlist.h* is defined to encapsulate the differences.

The *nlist* function, when used with the *nlist* structure can be used to extract certain information about selected symbols in the symbol table. The data associated with each symbol is machine specific, thus only the name and position of the *n_name* field in the *nlist* structure is standardized by HP-UX. The rest of the structure includes at least the value and type of the symbol. The names and meanings of all fields not standardized will change no more than necessary.

The structure, as defined for the Series 200, is:

```
struct nlist {
                  char              *n_name;
                  long              n_value;
                  unsigned char     n_type;
                  unsigned char     n_length;
                  short             n_almod;
                  short             n_unused;
        };
```

SEE ALSO

nlist (3), a.out (5)

NAME

passwd - password file, pwd.h

HP-UX COMPATIBILITY

Level: Multi-user – HP-UX/STANDARD

Origin: System V

DESCRIPTION

Passwd contains for each user the following information:

login name
encrypted password
numerical user ID
numerical group ID
reserved field which will be used for identification
initial working directory
program to use as shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. Each user is separated from the next by a new–line. If the password field is null, no password is demanded. If the shell field is null, /bin/sh is used.

This file resides in directory **/etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user IDs to names.

The encrypted password consists of 13 characters chosen from a 64–character set of "digits" described below, except when the password is null, in which case the encrypted password is also null. Login can be prevented by entering in the password field a character that is not part of the set of digits(e.g. *).

The characters used to represent "digits" are **.** for 0, **/** for 1, **0** through **9** for 2-11, **A** through **Z** for 12-37, and **a** through **z** for 38-63.

The super–user can set up "password aging" for a user by inserting a comma and a string of characters after the user's encrypted password. The first character in the string is the maximum number of weeks a password can remain valid; after that number of weeks elapses, the user will be required to choose a new password upon logging in. The second character is the minimum number of weeks that must elapse before the user can change passwords again. The rest of the characters tell in which week the password was last changed (weeks are counted from the beginning of 1970). A null string is the same as zero. If both the first and second characters are zero (i.e. both "."), the user must change passwords upon his or her next login; the "age" will then disappear from the password file altogether. For example, if the super–user inserts ",.." after a user's encrypted password, the user will have to change passwords upon logging in; when that happens, the ",.." will disappear. **The comma is required.**

If the second character has a greater decimal equivalent than the first (such as ",./"), only the super–user will be able to change the password.

Pwd.h designates the broken out password file as obtained by getpwent(3C):

```
struct passwd {
        char            *pw_name;
        char            *pw_passwd;
        unsigned int    *pw_uid;
        unsigned int    *pw_gid;
        char            *pw_age;
        char            *pw_comment;
        char            *pw_gecos;
        char            *pw_dir;
        char            *pw_shell;
```

```
    };
```

It is suggested that the range 0–99 not be used for user and group ID's (*pw_uid* and *pw_gid* in the above structure) so that IDs which may be assigned for system software do not conflict.

HARDWARE DEPENDENCIES

Series 200/500:

The following fields have character limitations as noted:

the login name field can be no longer than 8 characters;

the initial working directory field can be no longer than 63 characters;

the program field can be no longer than 44 characters.

The results are unpredictable if these fields are longer than the limits specified above.

The reserved field, called *pw_gcos* in the data structures used by *getpwent*(3C), is reserved for future use. It currently may be used to contain any information the system manager desires, but such use may conflict with the use of future HP features. The correct operation of the system will never depend on this field, but some optional feature may specify its format and content.

FILES

/etc/passwd

SEE ALSO

login(1), passwd(1), a64l(3C), crypt(3C), getpwent(3C), group(5).

NAME

privgrp – format of privileged values

SYNOPSIS

#include <sys/privgrp.h>

HP–UX COMPATIBILITY

Level: HP–UX/STANDARD

Origin: HP

Remarks:

Implemented on Series 200/300 only.

DESCRIPTION

Setprivgrp(2) sets a mask of privileges, and *Getprivgrp*(2) returns an array of structures giving privileged group assignments on a per group–id basis. *Privgrp.h* contains the constants and structures needed to deal with these system calls, and contains:

```
/*
 * Privileged group definitions --
 * the numeric values may vary between implementations.
 */
#define PRIV_RTPRIO     1
#define PRIV_MLOCK      2
#define PRIV_CHOWN      3

/* Maximum number of privileged groups in system */
#define PRIV_MAXGRPS  32

/*
 * Size of the privilege mask,
 * based on largest numbered privilege
 */

#define PRIV_MASKSIZ  1

/*
 * Structure defining the privilege mask
 */

struct privgrp_map {
    int    priv_groupno;
    unsigned int priv_mask[PRIV_MASKSIZ];
};

/*
 * Structure returned to user on getprivgrp system call.
 */

struct privgrp_list {
    struct privgrp_map[PRIV_MAXGRPS+1];
};
```

PRIV_RTPRIO allows access to the *rtprio*(2) system call.
PRIV_MLOCK allows access to the *plock*(2) system call.
PRIV_CHOWN allows users to give files away.

Privileges are described in a multi–word mask. The value of the **#define** for each privilege is interpreted as a bit index (counting from 1). Thus a group–id may have several different privileges assocaited with it by having different bits **or**'ed into the mask.

The system is configured with a maximun number of groups with special privileges. PRIV_MAXGRPS defines this maximum.

PRIV_MASKSIZ defines the size of the multi–word mask used defining privileges associated with a group–id.

Privileges are returned to the user from the *getprivgrp*(2) system call in array of structures of type **struct privgrp_mask**. The structure associates a multi–word mask with a group–id.

SEE ALSO

getprivgrp(2)

NAME

profile - set up user's environment at login time

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

If the file /etc/profile exists, it is executed by the shell for every user who logs in. /etc/profile should be set up to do only those things that are desirable for *every* user on the system, or to set reasonable defaults. If your login (home) directory contains a file named **.profile**, it will be executed before your session begins. Profile files are useful for setting various environment parameters, setting terminal modes, or overriding some or all of the results of executing /etc/profile.

The following example is typical (except for the comments):

```
# Make some environment variables global
export MAIL PATH TERM
# Set file creation mask
umask 22
# Tell me when new mail comes in
MAIL=/usr/mail/myname
# Add my /bin directory to the shell search sequence
PATH=$PATH:$HOME/bin
```

FILES

$HOME/.profile
/etc/profile

SEE ALSO

env(1), login(1), mail(1), sh(1), stty(1), su(1), environ(7), term(7).

NAME

ranlib - archive symbol table format for object libraries

SYNOPSIS

#include <ranlib.h>

HP-UX COMPATIBILITY

Level:	HP-UX/STANDARD
Origin:	HP

DESCRIPTION

Any archive containing object files also includes an archive symbol table, thus allowing the linker *ld* to scan libraries in random (rather than sequential) order.

The archive symbol table (if it exists) is always the first file in the archive, but it is never listed. It is automatically created and/or updated by *ar*.

The archive symbol table lists each externally known name in the archive, together with the offset of the archive element that defines that name. This offset is useful as an input argument to *lseek*(2) or *fseek*(3).

HARDWARE DEPENDENCIES

Series 500:

The archive symbol table file contains the symbol table and a name pool of strings (the names of external symbols). This allows for symbols with arbitrarily long names. The **rl_hdr** structure defines the layout of the file, and the **rl_ref** structure defines the contents of an archive symbol table entry. These structures have the following format:

```
struct rl_hdr {
        long int rl_tcbas;              /* offset of table */
        long int rl_tclen;              /* length of table */
        long int rl_nmbas;              /* offset of name pool */
        long int rl_nmlen;              /* length of name pool */
};

struct rl_ref {
        long int name_pos;              /* index into name pool */
        long int lib_pos;               /* offset of defining file */
};
```

Series 200:

The archive symbol table file contains a header, a name pool of strings (the names of external symbols), and the archive symbol table. This allows for symbols with arbitrarily long names. The header contains a short integer which specifies the number of entries, and a long integer which specifies the size of the string table. Following this is the name pool. The last section of the file contains the archive symbol table entries. The structure of these entries is defined below:

```
typedef long off_t;

struct      ranlib {
            union {
                    off_t ran_strx;         /* string table index */
                    char *ran_name;
            } ran_un;
            off_t   ran_off;                /* lib member offset */
};
```

SEE ALSO
ar(1), ld(1), ar(5).

NAME

sccsfile - format of SCCS file

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the **ASCII SOH** (start of heading) character (octal 001). This character is hereafter referred to as the *control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form **DDDDD** represent a five digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

```
@hDDDDD
```

The value of the checksum is the sum of all characters, except those of the first line. The @**h** provides a *magic number* of (octal) 000550 (0168 hex).

Delta table

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
 .
 .
 .
@c <comments> ...
 .
 .
 .
@e
```

The first line (@**s**) contains the number of lines inserted/deleted/unchanged respectively. The second line (@**d**) contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The @**i**, @**x**, and @**g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The @**m** lines (optional) each contain one **MR** number associated with the delta; the @**c** lines contain comments associated with the delta.

The @**e** line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines @**u** and @**U**. An empty list allows anyone to make a delta.

Flags~~~~~

Keywords used internally (see *admin*(1) for more information on their use). Each flag line takes the form:

```
@f <flag>        <optional text>
```

The following flags are defined:

```
@f t    <type of program>
@f v    <program name>
@f i
@f b
@f m    <module name>
@f f    <floor>
@f c    <ceiling>
@f d    <default-sid>
@f n
@f j
@f l    <lock-releases>
@f q    <user defined>
```

The **t** flag defines the replacement for the **%Y%** identification keyword. The **v** flag controls prompting for **MR** numbers in addition to comments; if the optional text is present it defines an **MR** number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the **b** flag is present the **-b** keyletter may be used on the *get* command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the **%M%** identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added. The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a *get* command. The **n** flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes *get* to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing (*get*(1) with the **-e** keyletter). The **q** flag defines the replacement for the **%Q%** identification keyword.

Comments

Arbitrary text surrounded by the bracketing lines @**t** and @**T**. The comments section typically will contain a description of the file's purpose.

Body~~~~~

The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: *insert*,~*delete*, and *end*, represented by:

@I DDDDD
@D DDDDD
@E DDDDD

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).

SCCS User's Guide in *HP-UX Concepts and Tutorials.*

NAME
term - format of compiled term file.

SYNOPSIS
term

HP-UX COMPATIBILITY
Level:	HP-UX/STANDARD
Origin:	System V

DESCRIPTION
Compiled terminfo descriptions are placed under the directory **/usr/lib/terminfo**. In order to avoid a linear search of a huge HP-UX system directory, a two-level scheme is used: **/usr/lib/terminfo/c/name** where *name* is the name of the terminal, and *c* is the first character of *name*. Thus, *act4* can be found in the file **/usr/lib/terminfo/a/act4**. Synonyms for the same terminal are implemented by multiple links to the same compiled file.

The format has been chosen so that it will be the same on all hardware. An 8 or more bit byte is assumed, but no assumptions about byte ordering or sign extension are made.

The compiled file is created with the *tic*(1M) program, and read by the routine *setupterm*. Both of these pieces of software are part of *curses*(3X). The file is divided into six parts: the header, terminal names, boolean flags, numbers, strings, and string table.

The header section begins the file. This section contains six short integers in the format described below. These integers are (1) the magic number (octal 0432); (2) the size, in bytes, of the names section; (3) the number of bytes in the boolean section; (4) the number of short integers in the numbers section; (5) the number of offsets (short integers) in the strings section; (6) the size, in bytes, of the string table.

Short integers are stored in two 8-bit bytes. The first byte contains the least significant 8 bits of the value, and the second byte contains the most significant 8 bits. (Thus, the value represented is 256*second+first.) The value -1 is represented by 0377, 0377, other negative value are illegal. The -1 generally means that a capability is missing from this terminal. Machines where this does not correspond to the hardware read the integers as two bytes and compute the result.

The terminal names section comes next. It contains the first line of the terminfo description, listing the various names for the terminal, separated by the '|' character. The section is terminated with an ASCII NUL character.

The boolean flags have one byte for each flag. This byte is either 0 or 1 as the flag is present or absent. The capabilities are in the same order as the file <term.h>.

Between the boolean section and the number section, a null byte will be inserted, if necessary, to ensure that the number section begins on an even byte. All short integers are aligned on a short word boundary.

The numbers section is similar to the flags section. Each capability takes up two bytes, and is stored as a short integer. If the value represented is -1, the capability is taken to be missing.

The strings section is also similar. Each capability is stored as a short integer, in the format above. A value of -1 means the capability is missing. Otherwise, the value is taken as an offset from the beginning of the string table. Special characters in ^X or \c notation are stored in their interpreted form, not the printing representation. Padding information $<nn> and parameter information %x are stored intact in uninterpreted form.

The final section is the string table. It contains all the values of string capabilities referenced in the string section. Each string is null terminated.

Note that it is possible for *setupterm* to expect a different set of capabilities than are actually present in the file. Either the database may have been updated since *setupterm* has been recompiled (resulting in extra unrecognized entries in the file) or the program may have been recompiled more recently than the database was updated (resulting in missing entries). The routine *setupterm* must be prepared for both possibilities - this is why the numbers and sizes are included. Also, new capabilities must always be added at the end of the lists of boolean, number, and string capabilities.

As an example, an octal dump of the description for the Microterm ACT 4 is included:

```
microterm|act4|microterm act iv,
   cr=^M, cud1=^J, ind=^J, bel=^G, am, cub1=^H,
   ed=^_, el=^^, clear=^L, cup=^T%p1%c%p2%c,
   cols#80, lines#24, cuf1=^X, cuu1=^Z, home=^],
```

```
000 032 001      \0 025  \0  \b  \0 212  \0   "  \0   m   i   c   r
020   o   t   e   r   m   |   a   c   t   4   |   m   i   c   r   o
040   t   e   r   m       a   c   t       i   v  \0  \0 001  \0  \0
060  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
100  \0  \0   P  \0 377 377 030  \0 377 377 377 377 377 377 377 377
120 377 377 377 377  \0  \0 002  \0 377 377 377 377 004  \0 006  \0
140  \b  \0 377 377 377 377  \n  \0 026  \0 030  \0 377 377 032  \0
160 377 377 377 377 034  \0 377 377 036  \0 377 377 377 377 377 377
200 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
520 377 377 377 377      \0 377 377 377 377 377 377 377 377 377 377
540 377 377 377 377 377 377 007  \0  \r  \0  \f  \0 036  \0 037  \0
560 024   %   p   1   %   c   %   p   2   %   c  \0  \n  \0 035  \0
600  \b  \0 030  \0 032  \0  \n  \0
```

Some limitations: total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes.

FILES

/usr/lib/terminfo/?/*   compiled terminal capability data base

SEE ALSO

tic(1M), curses(3X), terminfo(5).

NAME

terminfo - terminal capability data base

SYNOPSIS

/usr/lib/terminfo/?/*

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

Terminfo is a data base describing terminals that is used by programs and subroutines such as *vi*(1) and *curses*(3X). Terminals are described in *terminfo* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *terminfo.*

Entries in *terminfo* consist of a number of ',' separated fields. White space after each ',' is ignored. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name given is the most common abbreviation for the terminal, the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should be in lower case and contain no blanks; the last name may well contain upper case and blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, thus "hp2621". This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. Thus, a vt100 in 132 column mode would be vt100–w. The following suffixes should be used where possible:

Suffix	**Meaning**	**Example**
–w	Wide mode (more than 80 columns)	vt100–w
–am	With auto. margins (usually default)	vt100–am
–nam	Without automatic margins	vt100–nam
–*n*	Number of lines on the screen	aaa–60
–na	No arrow keys (leave them in local)	c100–na
–*n*p	Number of pages of memory	c100–4p
–rv	Reverse video	c100–rv

CAPABILITIES

The variable is the name by which the programmer (at the terminfo level) accesses the capability. The capname is the short name used in the text of the database, and is used by a person updating the database. The i.code is the two letter internal code used in the compiled database, and always corresponds to the old **termcap** capability name.

Capability names have no hard length limit, but an informal limit of 5 characters has been adopted to keep them short and to allow the tabs in the source file **caps** to line up nicely. Whenever possible, names are chosen to be the same as or similar to the ANSI X3.64–1979 standard. Semantics are also intended to match those of the specification.

(P)	indicates that padding may be specified
(G)	indicates that the string is passed through tparm withparms as given (#*i*).
(*)	indicates that padding may be based on the number of lines affected
(#$_i$)	indicates the i^{th} parameter.

Variable Booleans	Cap–name	I. Code	Description
auto_left_margin,	bw	bw	cub1 wraps from column 0 to last column
auto_right_margin,	am	am	Terminal has automatic margins
beehive_glitch,	xsb	xb	Beehive (f1=escape, f2=ctrl C)
ceol_standout_glitch,	xhp	xs	Standout not erased by overwriting (hp)
eat_newline_glitch,	xenl	xn	newline ignored after 80 cols (Concept)
erase_overstrike,	eo	eo	Can erase overstrikes with a blank
generic_type,	gn	gn	Generic line type (e.g.,, dialup, switch).
hard_copy,	hc	hc	Hardcopy terminal
has_meta_key,	km	km	Has a meta key (shift, sets parity bit)
has_status_line,	hs	hs	Has extra "status line"
insert_null_glitch,	in	in	Insert mode distinguishes nulls
memory_above,	da	da	Display may be retained above the screen
memory_below,	db	db	Display may be retained below the screen
move_insert_mode,	mir	mi	Safe to move while in insert mode
move_standout_mode,	msgr	ms	Safe to move in standout modes
over_strike,	os	os	Terminal overstrikes
status_line_esc_ok,	eslok	es	Escape can be used on the status line
teleray_glitch,	xt	xt	Tabs ruin, magic so char (Teleray 1061)
tilde_glitch,	hz	hz	Hazeltine; can not print ~'s
transparent_underline,	ul	ul	underline character overstrikes
xon_xoff,	xon	xo	Terminal uses xon/xoff handshaking
Numbers:			
columns,	cols	co	Number of columns in a line
init_tabs,	it	it	Tabs initially every # spaces
lines,	lines	li	Number of lines on screen or page
lines_of_memory,	lm	lm	Lines of memory if > lines. 0 means varies
magic_cookie_glitch,	xmc	sg	Number of blank chars left by smso or rmso
padding_baud_rate,	pb	pb	Lowest baud where cr/nl padding is needed
virtual_terminal,	vt	vt	Virtual terminal number (UNIX system)
width_status_line,	wsl	ws	No. columns in status line
Strings:			
back_tab,	cbt	bt	Back tab (P)
bell,	bel	bl	Audible signal (bell) (P)
carriage_return,	cr	cr	Carriage return (P*)
change_scroll_region,	csr	cs	change to lines #1 through #2 (vt100) (PG)
clear_all_tabs,	tbc	ct	Clear all tab stops (P)
clear_screen,	clear	cl	Clear screen and home cursor (P*)
clr_eol,	el	ce	Clear to end of line (P)

clr_eos,	ed	cd	Clear to end of display (P*)
column_address,	hpa	ch	Set cursor column (PG)
command_character,	cmdch	CC	Term. settable cmd char in prototype
cursor_address,	cup	cm	Screen rel. cursor motion row #1 col #2 (PG)
cursor_down,	cud1	do	Down one line
cursor_home,	home	ho	Home cursor (if no cup)
cursor_invisible,	civis	vi	Make cursor invisible
cursor_left,	cub1	le	Move cursor left one space
cursor_mem_address,	mrcup	CM	Memory relative cursor addressing
cursor_normal,	cnorm	ve	Make cursor appear normal (undo vs/vi)
cursor_right,	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll,	ll	ll	Last line, first column (if no cup)
cursor_up,	cuu1	up	Upline (cursor up)
cursor_visible,	cvvis	vs	Make cursor very visible
delete_character,	dch1	dc	Delete character (P*)
delete_line,	dl1	dl	Delete line (P*)
dis_status_line,	dsl	ds	Disable status line
down_half_line,	hd	hd	Half-line down (forward 1/2 linefeed)
enter_alt_charset_mode,	smacs	as	Start alternate character set (P)
enter_blink_mode,	blink	mb	Turn on blinking
enter_bold_mode,	bold	md	Turn on bold (extra bright) mode
enter_ca_mode,	smcup	ti	String to begin programs that use cup
enter_delete_mode,	smdc	dm	Delete mode (enter)
enter_dim_mode,	dim	mh	Turn on half-bright mode
enter_insert_mode,	smir	im	Insert mode (enter);
enter_protected_mode,	prot	mp	Turn on protected mode
enter_reverse_mode,	rev	mr	Turn on reverse video mode
enter_secure_mode,	invis	mk	Turn on blank mode (chars invisible)
enter_standout_mode,	smso	so	Begin stand out mode
enter_underline_mode,	smul	us	Start underscore mode
erase_chars	ech	ec	Erase #1 characters (PG)
exit_alt_charset_mode,	rmacs	ae	End alternate character set (P)
exit_attribute_mode,	sgr0	me	Turn off all attributes
exit_ca_mode,	rmcup	te	String to end programs that use cup
exit_delete_mode,	rmdc	ed	End delete mode
exit_insert_mode,	rmir	ei	End insert mode
exit_standout_mode,	rmso	se	End stand out mode
exit_underline_mode,	rmul	ue	End underscore mode
flash_screen,	flash	vb	Visible bell (may not move cursor)
form_feed,	ff	ff	Hardcopy terminal page eject (P*)
from_status_line,	fsl	fs	Return from status line
init_1string,	is1	i1	Terminal initialization string
init_2string,	is2	i2	Terminal initialization string
init_3string,	is3	i3	Terminal initialization string
init_file,	if	if	Name of file containing is
insert_character,	ich1	ic	Insert character (P)
insert_line,	il1	al	Add new blank line (P*)
insert_padding,	ip	ip	Insert pad after character inserted (p*)
key_backspace,	kbs	kb	Sent by backspace key
key_catab,	ktbc	ka	Sent by clear-all-tabs key
key_clear,	kclr	kC	Sent by clear screen or erase key
key_ctab,	kctab	kt	Sent by clear-tab key

key_dc,	kdch1	kD	Sent by delete character key
key_dl,	kdl1	kL	Sent by delete line key
key_down,	kcud1	kd	Sent by terminal down arrow key
key_eic,	krmir	kM	Sent by rmir or smir in insert mode
key_eol,	kel	kE	Sent by clear-to-end-of-line key
key_eos,	ked	kS	Sent by clear-to-end-of-screen key
key_f0,	kf0	k0	Sent by function key f0
key_f1,	kf1	k1	Sent by function key f1
key_f10,	kf10	k;	Sent by function key f10
key_f2,	kf2	k2	Sent by function key f2
key_f3,	kf3	k3	Sent by function key f3
key_f4,	kf4	k4	Sent by function key f4
key_f5,	kf5	k5	Sent by function key f5
key_f6,	kf6	k6	Sent by function key f6
key_f7,	kf7	k7	Sent by function key f7
key_f8,	kf8	k8	Sent by function key f8
key_f9,	kf9	k9	Sent by function key f9
key_home,	khome	kh	Sent by home key
key_ic,	kich1	kI	Sent by ins char/enter ins mode key
key_il,	kil1	kA	Sent by insert line
key_left,	kcub1	kl	Sent by terminal left arrow key
key_ll,	kll	kH	Sent by home-down key
key_npage,	knp	kN	Sent by next-page key
key_ppage,	kpp	kP	Sent by previous-page key
key_right,	kcuf1	kr	Sent by terminal right arrow key
key_sf,	kind	kF	Sent by scroll-forward/down key
key_sr,	kri	kR	Sent by scroll-backward/up key
key_stab,	khts	kT	Sent by set-tab key
key_up,	kcuu1	ku	Sent by terminal up arrow key
keypad_local,	rmkx	ke	Out of "keypad transmit" mode
keypad_xmit,	smkx	ks	Put terminal in "keypad transmit" mode
lab_f0,	lf0	l0	Labels on function key f0 if not f0
lab_f1,	lf1	l1	Labels on function key f1 if not f1
lab_f10,	lf10	la	Labels on function key f10 if not f10
lab_f2,	lf2	l2	Labels on function key f2 if not f2
lab_f3,	lf3	l3	Labels on function key f3 if not f3
lab_f4,	lf4	l4	Labels on function key f4 if not f4
lab_f5,	lf5	l5	Labels on function key f5 if not f5
lab_f6,	lf6	l6	Labels on function key f6 if not f6
lab_f7,	lf7	l7	Labels on function key f7 if not f7
lab_f8,	lf8	l8	Labels on function key f8 if not f8
lab_f9,	lf9	l9	Labels on function key f9 if not f9
memory_lock	meml	ml	Enable memory lock
memory_unlock	memu	mu	Disable memory lock
meta_on,	smm	mm	Turn on "meta mode" (8th bit)
meta_off,	rmm	mo	Turn off "meta mode"
newline,	nel	nw	Newline (behaves like cr followed by lf)
pad_char,	pad	pc	Pad character (rather than null)
parm_dch,	dch	DC	Delete #1 chars (PG*)
parm_delete_line,	dl	DL	Delete #1 lines (PG*)
parm_down_cursor,	cud	DO	Move cursor down #1 lines (PG*)
parm_ich,	ich	IC	Insert #1 blank chars (PG*)
parm_index,	indn	SF	Scroll forward #1 lines (PG)

parm_insert_line,	il	AL	Add #1 new blank lines (PG*)
parm_left_cursor,	cub	LE	Move cursor left #1 spaces (PG)
parm_right_cursor,	cuf	RI	Move cursor right #1 spaces (PG*)
parm_rindex,	rin	SR	Scroll backward #1 lines (PG)
parm_up_cursor,	cuu	UP	Move cursor up #1 lines (PG*)
pkey_key,	pfkey	pk	Prog funct key #1 to type string #2
pkey_local,	pfloc	pl	Prog funct key #1 to execute string #2
pkey_xmit,	pfx	px	Prog funct key #1 to xmit string #2
print_screen,	mc0	ps	Print contents of the screen
prtr_off,	mc4	pf	Turn off the printer
prtr_on,	mc5	po	Turn on the printer
repeat_char,	rep	rp	Repeat char #1 #2 times. (PG*)
reset_1string,	rs1	r1	Reset terminal completely to sane modes.
reset_2string,	rs2	r2	Reset terminal completely to sane modes.
reset_3string,	rs3	r3	Reset terminal completely to sane modes.
reset_file,	rf	rf	Name of file containing reset string
restore_cursor,	rc	rc	Restore cursor to position of last sc
row_address,	vpa	cv	Vertical position absolute (set row) (PG)
save_cursor,	sc	sc	Save cursor position (P)
scroll_forward,	ind	sf	Scroll text up (P)
scroll_reverse,	ri	sr	Scroll text down (P)
set_attributes,	sgr	sa	Define the video attributes (PG9)
set_tab,	hts	st	Set a tab in all rows, current column
set_window,	wind	wi	Current window is lines #1-#2 cols #3-#4
tab,	ht	ta	Tab to next 8 space hardware tab stop
to_status_line,	tsl	ts	Go to status line, column #1
underline_char,	uc	uc	Underscore one char and move past it
up_half_line,	hu	hu	Half-line up (reverse 1/2 linefeed)
init_prog,	iprog	iP	Path name of program for init
key_a1,	ka1	K1	Upper left of keypad
key_a3,	ka3	K3	Upper right of keypad
key_b2,	kb2	K2	Center of keypad
key_c1,	kc1	K4	Lower left of keypad
key_c3,	kc3	K5	Lower right of keypad
prtr_non,	mc5p	pO	Turn on the printer for #1 bytes

A Sample Entry

The following entry, which describes the Concept-100, is among the more complex entries in the *terminfo* file as of this writing.

```
concept100  | c100| concept | c104 | c100-4p | concept  100,
     am, bel=^G, blank=\EH, blink=\EC, clear=^L$<2*>, cnorm=\Ew,
     cols#80, cr=^M$<9>, cub1=^H, cud1=^J, cuf1=\E=,
     cup=\Ea%p1%'  '%+%c%p2%'  '%+%c,
     cuu1=\E;, cvvis=\EW, db, dch1=\E^A$<16*>, dim=\EE, dl1=\E^B$<3*>,
     ed=\E^C$<16*>, el=\E^U$<16>, eo, flash=\Ek$<20>\EK, ht=\t$<8>,
     il1=\E^R$<3*>, in, ind=^J, .ind=^J$<9>, ip=$<16*>,
     is2=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200\Eo\47\E,
     kbs=^h, kcub1=\E>, kcud1=\E<, kcuf1=\E=, kcuu1=\E;,
     kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
     lines#24, mir, pb#9600, prot=\EI, rep=\Er%p1%c%p2%'   '%+%c$<.2*>,
     rev=\ED, rmcup=\Ev     $<6>\Ep\r\n, rmir=\E\200, rmkx=\Ex,
     rmso=\Ed\Ee, rmul=\Eg, rmul=\Eg, sgr0=\EN\200,
smcup=\EU\Ev    8p\Ep\r,  smir=\E^P,  smkx=\EX,  smso=\EE\ED,
smul=\EG,  tabs,  ul,  vt#8,  xenl,
```

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Comments may be included on lines beginning with "#". Capabilities in *terminfo* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have names. For instance, the fact that the Concept has *automatic margins* (i.e., an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **cols**, which indicates the number of columns the terminal has, gives the value '80' for the Concept.

Finally, string valued capabilities, such as **el** (clear to end of line sequence) are given by the two-character code, an '=', and then a string ending at the next following ','. A delay in milliseconds may appear anywhere in such a capability, enclosed in $<..> brackets, as in **el**=\EK$<3>, and padding characters are supplied by *tputs* to provide this delay. The delay can be either a number, e.g., '20', or a number followed by an '*', i.e., '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert character, the factor is still the number of *lines* affected. This is always one unless the terminal has **xenl** and the software uses it.) When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.)

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. Both **\E** and **\e** map to an ESCAPE character, **^x** maps to a control-x for any appropriate x, and the sequences **\n \l \r \t \b \f \s** give a newline, linefeed, return, tab, backspace, formfeed, and space. Other escapes include \^ for ^, \\ for \, \, for comma, \: for :, and \0 for null. (\0 will produce \200, which does not terminate a string but behaves as a null character on most terminals.) Finally, characters may be given as three octal digits after a ****.

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example above.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *terminfo* and to build up a description gradually, using partial descriptions with *vi* to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *terminfo* file to describe it or bugs in *vi.* To easily test a new terminal description you can set the environment variable TERMINFO to a pathname of a directory containing the compiled description you are working on and programs will look there rather than in */usr/lib/terminfo*. To get the padding for insert line right (if the terminal manufacturer did not document it) a severe test is to edit /etc/passwd at 9600 baud, delete 16 or so lines from the middle of the screen, then hit the 'u' key several times quickly. If the terminal messes up, more padding is usually needed. A similar test can be used for insert character.

Basic Capabilities

The number of columns on each line for the terminal is given by the **cols** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **lines** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the **clear** string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os**. (**os** applies to storage scope terminals, such as TEKTRONIX 4010 series, as well as hard copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr**. (Normally this will be carriage return, control M.) If there is a code to produce an audible signal (bell, beep, etc) give this as **bel**.

If there is a code to move the cursor one position to the left (such as backspace) that capability should be given as **cub1**. Similarly, codes to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**. These local cursor motions should not alter the text they pass over, for example, you would not normally use '**cuf1**= ' because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a CRT terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and never attempt to go up locally off the top. In order to scroll text up, a program will go to the bottom left corner of the screen and send the **ind** (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin** which have the same semantics as **ind** and **ri** except that they take one parameter, and scroll that many lines. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. The only local motion which is defined from the left edge is if **bw** is given, then a **cub1** from the left edge will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch selectable automatic margins, the *terminfo* file usually assumes that this is on; i.e., **am**. If the terminal has a command which moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the terminal has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe hardcopy and glass-tty terminals. Thus the model 33 teletype is described as

```
33 | tty33 | tty | model 33 teletype,
bel=^G, cols#72, cr=^M, cud1=^J, hc, ind=^J, os,
```

while the Lear Siegler ADM-3 is described as

```
adm3 | 3 | lsi adm3,
am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H, cud1=^J,
ind=^J, lines#24,
```

Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability, with *printf*(3S) like escapes **%x** in it. For example, to address the cursor, the **cup** capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup**.

The parameter mechanism uses a stack and special **%** codes to manipulate it. Typically a sequence will push one of the parameters onto the stack and then print it in some format. Often more complex operations are necessary.

The % encodings have the following meanings:

%%	outputs '%'
%d	print pop() as in printf
%2d	print pop() like %2d
%3d	print pop() like %3d
%02d	
%03d	as in printf
%c	print pop() gives %c
%s	print pop() gives %s
%p[1-9]	push ith parm
%P[a-z]	set variable [a-z] to pop()
%g[a-z]	get variable [a-z] and push it
%'c'	char constant c
%{nn}	integer constant nn
%+ %- %* %/ %m	arithmetic (%m is mod): push(pop() op pop())
%& %| %^	bit operations: push(pop() op pop())
%= %> %<	logical operations: push(pop() op pop())
%! %~	unary operations push(op pop())
%i	add 1 to first two parms (for ANSI terminals)
%? expr %t thenpart %e elsepart %;	if-then-else, %e elsepart is optional. else-if's are possible ala Algol 68: %? c_1 %t b_1 %e c_2 %t b_2 %e c_3 %t b_3 %e c_4 %t b_4 %e %; c_i are conditions, b_i are bodies.

Binary operations are in postfix form with the operands in the usual order. That is, to get x–5 one would use "%gx%{5}%–".

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cup** capability is cup=6\E&%p2%2dc%p1%2dY.

The Microterm ACT–IV needs the current row and column sent preceded by a **^T**, with the row and column simply encoded in binary, cup=^T%p1%c%p2%c. Terminals which use %c need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit **\n ^D** and **\r**, as the system may change or discard them. (The library routines dealing with terminfo set tty modes so that tabs are never expanded, so \t is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM–3a, which uses row and column offset by a blank character, thus cup=\E=%p1%' '%+%c%p2%' '%+%c. After sending '\E=', this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values) and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

If the terminal has row or column absolute cursor addressing, these can be given as single parameter capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two parameter sequence (as with the hp2645) and can be used in preference to **cup** . If there are parameterized local motions (e.g., move *n* spaces to the right) these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not have **cup**, such as the

TEKTRONIX 4025.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the \EH sequence on HP terminals cannot be used for **home**.)

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el**. If the terminal can clear from the current position to the end of the display, then this should be given as **ed**. **Ed** is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single parameter and insert or delete that many lines can be given as **il** and **dl**. If the terminal has a settable scrolling region (like the vt100) the command to set this can be described with the **csr** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command - the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string **wind**. The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *terminfo.* The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type abc def using local cursor motions (not spaces) between the abc and the def. Then position the cursor before the abc and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the abc shifts over to the def which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for insert null. While these are two logically separate attributes (one line vs. multiline insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

Terminfo can describe both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals which send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal actually requires both to be used in combination.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If your terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n*, will repeat the effects of **ich1** *n* times.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mir** to speed up inserting in this case. Omitting **mir** will affect only speed. Some terminals (notably Datamedia's) must not have **mir** because of the way their insert mode works.

Finally, you can specify **dch1** to delete a single character, **dch** with one parameter, *n*, to delete *n characters,* and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one parameter.

Highlighting, Underlining, and Visible Bells

If your terminal has one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode*, representing a good, high contrast, easy–on–the–eyes, format for highlighting error messages and other attention getters. (If you have a choice, reverse video plus half–bright is good, or reverse video alone.) The sequences to enter and exit standout mode are given as **smso** and **rmso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **smul** and **rmul** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **blink** (blinking) **bold** (bold or extra bright) **dim** (dim or half–bright) **invis** (blanking or invisible text) **prot** (protected) **rev** (reverse video) **sgr0** (turn off *all* attribute modes) **smacs** (enter alternate character set mode) and **rmacs** (exit alternate character set mode). Turning on any of these modes singly may or may not turn off other modes.

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking 9 parameters. Each parameter is either 0 or 1, as the corresponding attribute is on or off. The 9 parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need be supported by **sgr**, only those for which corresponding separate attribute commands exist.

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode–setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the HP 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit stan–dout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **flash**; it must not move the cursor.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis**. If there is a way to make the cursor completely invisible, give that as **civis**. The capability **cnorm** should be given which undoes the effects of both of these modes.

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly. This is also used for the TEKTRONIX 4025, where **smcup** sets the command character to be the one used by terminfo.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **smkx** and **rmkx**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1, kcuf1, kcuu1, kcud1,** and **khome** respectively. If there are function keys such as f0, f1, ..., f10, the codes they send can be given as **kf0, kf1, ..., kf10**. If these keys have labels other than the default f0 through f10, the labels can be given as **lf0, lf1, ..., lf10**. The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as **ka1**, **ka3**, **kb2**, **kc1**, and **kc3**. These keys are useful when the effects of a 3 by 3 directional pad are needed.

Tabs and Initialization

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control I). A "backtab" command which moves leftward to the next tab stop can be given as **cbt**. By convention, if the teletype modes indicate that tabs are being expanded by the computer rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs which are initially set every *n* spaces when the terminal is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used by the *tset* command to determine whether to set the mode for hardware tab expansion, and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the terminfo description can assume that they are properly set.

Other capabilities include **is1**, **is2**, and **is3**, initialization strings for the terminal, **iprog**, the path name of a program to be run to initialize the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the terminfo description. They are normally sent to the terminal, by the *tset* program, each time the user logs in. They will be printed in the following order: **is1**; **is2**; setting tabs using **tbc** and **hts**; **if**; running the program **iprog**; and finally **is3**. Most initialization is done with **is2**. Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. A pair of sequences that does a harder reset from a totally unknown state can be analogously given as **rs1**, **rs2**, **rf**, and **rs3**, analogous to **is2** and **if**. These strings are output by the *reset* program, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs2** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set the vt100 into 80–column mode would normally be part of **is2**, but it causes an annoying glitch of the screen and is not normally needed since the terminal is usually already in 80 column mode.

If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row). If a more complex sequence is needed to set the tabs than can be described by this, the sequence can be placed in **is2** or **if**.

Delays

Certain capabilities control padding in the teletype driver. These are primarily needed by hard copy terminals, and are used by the *tset* program to set teletype modes appropriately. Delays embedded in the capabilities **cr**, **ind**, **cub1**, **ff**, and **tab** will cause the appropriate delay bits to be set in the teletype driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used.

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit h19's 25th line, or the 24th line of a vt100 which is set to a 23–line scrolling region), the capability **hs** should be given. Special strings to go to the beginning of the status line and to return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The parameter **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to. If escape sequences and other special commands, such as tab, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**. The status line is normally assumed to be the same width as the rest of the screen, e.g., **cols**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half–line up) and **hd** (half–line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string **rep**. The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, tparm(repeat_char, 'x', 10) is the same as 'xxxxxxxxxx'.

If the terminal has a settable command character, such as the TEKTRONIX 4025, this can be indicated with **cmdch**. A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some UNIX systems: The environment is to be searched for a **CC** variable, and if found, all occurrences of the prototype character are replaced with the character in the environment variable.

Terminal descriptions that do not represent a specific kind of known terminal, such as *switch*, *dialup*, *patch*, and *network*, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to *virtual* terminal descriptions for which the escape sequences are known.)

If the terminal uses xon/xoff handshaking for flow control, give **xon**. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **smm** and **rmm**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm**#0 indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

If the terminal is one of those supported by the UNIX virtual terminal protocol, the terminal number can be given as **vt**.

Media copy strings which control an auxiliary printer connected to the terminal can be given as **mc0**: print the contents of the screen, **mc4**: turn off the printer, and **mc5**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. It is undefined whether the text is also displayed on the terminal screen when the printer is on. A variation **mc5p** takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Strings to program function keys can be given as **pfkey**, **pfloc**, and **pfx**. Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal dependent manner. The difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local; and **pfx** causes the string to be transmitted to the computer.

Glitches and Braindamage

Hazeltine terminals, which do not allow '~' characters to be displayed should indicate **hz**.

Terminals which ignore a linefeed immediately after an **am** wrap, such as the Concept and vt100, should indicate **xenl**.

If **el** is required to get rid of standout (instead of merely writing normal text on top of it), **xhp** should be given.

Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This glitch is also taken to mean that it is not possible to position the cursor on top of a "magic cookie", that to erase standout mode it is instead necessary to use delete and insert line.

The Beehive Superbee, which is unable to correctly transmit the escape or control C characters, has **xsb**, indicating that the f1 key is used for escape and f2 for control C. (Only certain Superbees have this problem, depending on the ROM.)

Other specific terminal problems may be corrected by adding more capabilities of the form **x***x*.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be cancelled by placing **xx@** to the left of the capability definition, where xx is the capability.

For example, the entry

```
2621–nl, smkx@, rmkx@, use=2621,
```

defines a 2621–nl that does not have the **smkx** or **rmkx** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/usr/lib/terminfo/?/* files containing terminal descriptions

SEE ALSO

tic(1M), curses(3X), printf(3S), term(5).

NAME

ttytype - data base of terminal types by port

SYNOPSIS

/etc/ttytype

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	UCB
Remarks:	Not supported on the Integral Personal Computer.

DESCRIPTION

Ttytype is a database containing, for each tty port on the system, the kind of terminal that is attached to that port. There is one line per port, containing the terminal kind (as a name listed in *terminfo*(5)), a space, and the name of the tty, less the initial "/dev/". For example, for an HP 2622 terminal on tty02:

```
2622 tty02
```

This information is read by *tset*(1) and by *login*(1) to initialize the TERM variable at login time.

SEE ALSO

login(1), tset(1).

BUGS

Some lines are only known as "dialup" or "plugboard".

NAME

utmp, wtmp, btmp - utmp, wtmp, btmp entry format

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V and UCB

SYNOPSIS

#include <sys/types.h>
#include <utmp.h>

DESCRIPTION

These files, which hold user and accounting information for such commands as *last(1), who*(1), *write*(1), and *login*(1), have the following structure as defined by **<utmp.h>**:

```
#define UTMP_FILE               "/etc/utmp"
#define WTMP_FILE               "/etc/wtmp"

#define ut_name ut_user

struct  utmp
{
        char    ut_user[8];        /* User login name */
        char    ut_id[4];          /* /etc/inittab id (usually line #) */
        char    ut_line[12];       /* device name (console, lnxx) */
        short   ut_pid;            /* process id */
        short   ut_type;           /* type of entry */
        struct  exit_status {
            short     e_termination;     /* Process termination status */
            short     e_exit;            /* Process exit status */
        } ut_exit;                 /* The exit status of a process
                                    * marked as DEAD_PROCESS. */
        time_t  ut_time;           /* time entry was made */
};

/*  Definitions for ut_type  */
#define EMPTY                   0
#define RUN_LVL                 1
#define BOOT_TIME               2
#define OLD_TIME                3
#define NEW_TIME                4
#define INIT_PROCESS            5  /* Process spawned by "init" */
#define LOGIN_PROCESS           6  /* A "getty" process waiting for login */
#define USER_PROCESS            7  /* A user process */
#define DEAD_PROCESS            8
#define ACCOUNTING              9
#define UTMAXTYPE ACCOUNTING       /* Largest legal value of ut_type */
```

```
/* Special strings or formats used in the "ut_line" field when */
/* accounting for something other than a process */
/* No string for the ut_line field can be more than 11 chars + */
/* a NULL in length */

#define RUNLVL_MSG          "run-level %c"
#define BOOT_MSG            "system boot"
#define OTIME_MSG           "old time"
#define NTIME_MSG           "new time"
```

Ut_name is valid for login entries only; otherwise the first character is null. There are logout entries in both **utmp** and **wtmp**. In **utmp**, these entries refer to terminals that are not currently logged in; in **wtmp**, they record history. File **btmp** contains bad login entries for each invalid logon attempt.

Note that **wtmp** and **btmp** tend to grow without bound, and should be checked regularly. Information that is no longer useful should be removed periodically to prevent it from becoming too large.

FILES

/etc/utmp
/etc/wtmp
/etc/btmp

SEE ALSO

acctcon(1M), fwtmp(1m), last(1), lastb(1), login(1), who(1), write(1), getut(3C).

NAME

intro - introduction to miscellany

DESCRIPTION

This section describes miscellaneous facilities such as macro packages, character set tables, etc.

NAME

ascii - map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System III

DESCRIPTION

Ascii is a map of the ASCII character set, giving both octal and hexadecimal equivalents of each character, to be printed as needed. It contains:

```
|000 nul|001 soh|002 stx|003 etx|004 eot|005 enq|006 ack|007 bel|
|010 bs |011 ht |012 nl |013 vt |014 np |015 cr |016 so |017 si |
|020 dle|021 dc1|022 dc2|023 dc3|024 dc4|025 nak|026 syn|027 etb|
|030 can|031 em |032 sub|033 esc|034 fs |035 gs |036 rs |037 us |
|040 sp |041 !  |042 "  |043 #  |044 $  |045 %  |046 &  |047 '  |
|050 (  |051 )  |052 *  |053 +  |054 ,  |055 -  |056 .  |057 /  |
|060 0  |061 1  |062 2  |063 3  |064 4  |065 5  |066 6  |067 7  |
|070 8  |071 9  |072 :  |073 ;  |074 <  |075 =  |076 >  |077 ?  |
|100 @  |101 A  |102 B  |103 C  |104 D  |105 E  |106 F  |107 G  |
|110 H  |111 I  |112 J  |113 K  |114 L  |115 M  |116 N  |117 O  |
|120 P  |121 Q  |122 R  |123 S  |124 T  |125 U  |126 V  |127 W  |
|130 X  |131 Y  |132 Z  |133 [  |134 \  |135 ]  |136 ^  |137 _  |
|140 `  |141 a  |142 b  |143 c  |144 d  |145 e  |146 f  |147 g  |
|150 h  |151 i  |152 j  |153 k  |154 l  |155 m  |156 n  |157 o  |
|160 p  |161 q  |162 r  |163 s  |164 t  |165 u  |166 v  |167 w  |
|170 x  |171 y  |172 z  |173 {  |174 |  |175 }  |176 ~  |177 del|

| 00 nul| 01 soh| 02 stx| 03 etx| 04 eot| 05 enq| 06 ack| 07 bel|
| 08 bs | 09 ht | 0a nl | 0b vt | 0c np | 0d cr | 0e so | 0f si |
| 10 dle| 11 dc1| 12 dc2| 13 dc3| 14 dc4| 15 nak| 16 syn| 17 etb|
| 18 can| 19 em | 1a sub| 1b esc| 1c fs | 1d gs | 1e rs | 1f us |
| 20 sp | 21 !  | 22 "  | 23 #  | 24 $  | 25 %  | 26 &  | 27 '  |
| 28 (  | 29 )  | 2a *  | 2b +  | 2c ,  | 2d -  | 2e .  | 2f /  |
| 30 0  | 31 1  | 32 2  | 33 3  | 34 4  | 35 5  | 36 6  | 37 7  |
| 38 8  | 39 9  | 3a :  | 3b ;  | 3c <  | 3d =  | 3e >  | 3f ?  |
| 40 @  | 41 A  | 42 B  | 43 C  | 44 D  | 45 E  | 46 F  | 47 G  |
| 48 H  | 49 I  | 4a J  | 4b K  | 4c L  | 4d M  | 4e N  | 4f O  |
| 50 P  | 51 Q  | 52 R  | 53 S  | 54 T  | 55 U  | 56 V  | 57 W  |
| 58 X  | 59 Y  | 5a Z  | 5b [  | 5c \  | 5d ]  | 5e ^  | 5f _  |
| 60 `  | 61 a  | 62 b  | 63 c  | 64 d  | 65 e  | 66 f  | 67 g  |
| 68 h  | 69 i  | 6a j  | 6b k  | 6c l  | 6d m  | 6e n  | 6f o  |
| 70 p  | 71 q  | 72 r  | 73 s  | 74 t  | 75 u  | 76 v  | 77 w  |
| 78 x  | 79 y  | 7a z  | 7b {  | 7c |  | 7d }  | 7e ~  | 7f del|
```

FILES

/usr/pub/ascii

NAME

environ - user environment

HP-UX COMPATIBILITY

Level: HP-UX/NUCLEUS

Origin: System III

DESCRIPTION

An array of strings called the "environment" is made available by *exec*(2) when a process begins. By convention, these strings have the form "name=value". The following names are used by various commands:

PATH The sequence of directory prefixes that *sh*(1), *time*(1), *nice*(1), *nohup*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (**:**). *Login*(1) sets **PATH=:/bin:/usr/bin**.

HOME Name of the user's login directory, set by *login*(1) from the password file *passwd*(5).

TERM The kind of terminal for which output is to be prepared. This information is used by commands (such as *mm*(1) or *vi*(1)) that are able to exploit special capabilities of that terminal.

TZ Time zone information. The minimum format is **tzname***diff* where **tzname** is an "alphabetic" string giving the time zone name or abbreviation, and *diff* is the (positive or negative, and possibly fractional) difference in hours from GMT. NOTE: west is positive, east is negative. If a summer time zone adjustment (such as Daylight Savings in the US) is to be applied the format is **tzname***diff***dstzname** where **dstzname** is the name of the "Daylight Savings" time zone.

LANG Language selection. This is one of the names listed in *langid*(7). It is used to select the character set, lexical order, up and down shift tables, and other information which varies from one area to another.

Further names may be placed in the environment by the *export* command and "name=value" arguments in *sh*(1), by the *setenv* command in *csh*(1), by the *env*(1) command, or by *exec*(2). It is unwise to conflict with certain shell variables that are frequently exported by **.profile** files: **MAIL**, **PS1**, **PS2**, **IFS**.

SEE ALSO

env(1), login(1), sh(1), exec(2), ctime(3C), getenv(3C), profile(5), tztab(5), hpnls(7), term(7).

NAME

fcntl - file control options

SYNOPSIS

#include <fcntl.h>

HP-UX COMPATIBILITY

Level:

Basic calls: HP-UX/RUN ONLY

Real time extensions: HP-UX/STANDARD - Real Time

Origin: System III, System V, UCB, and HP

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */

#define O_RDONLY   0
#define O_WRONLY   1
#define O_RDWR     2
#define O_NDELAY   04          /* Non-blocking I/O */
#define O_APPEND   010         /* append (writes guaranteed at the end) */
#define O_SYNCIO   0100000     /* Do write through caching */

/* Flag values accessible only to open(2) */
#define O_CREAT    00400       /* Open with file create (uses third open arg)*/
#define O_TRUNC    01000       /* Open with truncation */
#define O_EXCL     02000       /* Exclusive open */

/* fcntl(2) requests */
#define F_DUPFD    0           /* Duplicate fildes */
#define F_GETFD    1           /* Get fildes flags */
#define F_SETFD    2           /* Set fildes flags */
#define F_GETFL    3           /* Get file flags */
#define F_SETFL    4           /* Set file flags */
```

SEE ALSO

fcntl(2), open(2).

NAME

hier - file system hierarchy

DESCRIPTION

The following outline gives a quick tour through a representative HP–UX directory hierarchy. Some of the directories listed only appear with HP–UX versions which support certain optional commands or packages which use those directories. Some HP–UX versions add special directories not shown here.

/ Root directory.

/bin Frequently–used commands and those required to boot, restore, recover, and/or repair the system.

/dev Special files (device files); see *mknod*(8).

/etc System administrative commands and configuration files.

/etc/newconfig
New (updated) versions of customizable (localizable) configuration files and shell scripts. Shipped here so as not to overwrite current versions. Copied to regular locations for newly installed systems. Administrators may wish to keep them around for later reference.

/lib Frequently–used object code libraries and related utilities.

/lost+found
For connecting detached files; for use by *fsck*(8).

/tmp Place to put temporary files (those normally with short lifetimes and which may be removed without notice).

/users User home directories; sometimes immediate, sometimes at lower levels.

/users/guest
Default home directory for user "guest"; see *passwd*(5). Directory exists for novice users; you may wish to remove it.

/usr Less–frequently–used commands and other miscellaneous things; historically, often a separate, mounted volume.

/usr/adm System–administrative data files.

/usr/bin Less–frequently–used commands and those not required to boot, restore, recover, and/or repair the system.

/usr/contrib
User–contributed (unsupported, internal) commands, files, etc. Files under this directory come from outside the local site or organization, e.g. from users groups, HP service engineers, etc. See */usr/local* for local–site commands and files.

/usr/contrib/bin
User–contributed commands.

/usr/contrib/games
User–contributed games.

/usr/contrib/include
User–contributed include files. To include them, you must (in C) give a complete pathname, for example, #include "/usr/contrib/include/symtab.h".

/usr/contrib/lib
User–contributed libraries.

/usr/contrib/man/cat[1–8]
User–contributed manual entries, post–nroff form.

/usr/contrib/man/man[1–8]
: User–contributed manual entries, pre–nroff form.

/usr/contrib/man/$LANG/cat[1–8]
: User–contributed manual entries, formatted form for installed native languages. The LANG environment variable may take on values given in the **/usr/lib/nls/config** table.

/usr/contrib/man/$LANG/man[1–8]
: User–contributed manual entries, unformatted form for installed native languages.

/usr/include
: High–level C–language header files (shared definitions).

/usr/include/sys
: Low–level (kernel–related) C–language header files.

/usr/lib
: Less–frequently–used object code libraries, related utilities, miscellaneous data files, etc.

/usr/lib/acct
: Certain system–administrative commands.

/usr/lib/cron
: For *cron*(1) and *at*(1) scheduling information.

/usr/lib/graphics/c
: Device–independent Graphics Library (DGL) special C–language include files. Optional on some systems.

/usr/lib/graphics/demos
: DGL demonstration software.

/usr/lib/graphics/fortran
: DGL special FORTRAN–language include files.

/usr/lib/graphics/pascal
: DGL special Pascal–language include files.

/usr/lib/help
: Data files for *help*(1).

/usr/lib/lex
: Data files for *lex*(1).

/usr/lib/macros
: Macro definition packages for *nroff*(1).

/usr/lib/nls
: native language support

/usr/lib/nls/config
: correspondence between integer language id and name

/usr/lib/nls/$LANG
: Language definition (Character Set Support, Local Customs, and Messages) for installed native languages. The LANG environment variable may take on values given in the **/usr/lib/nls/config** table.

/usr/lib/spell
: Data files for *spell*(1).

/usr/lib/tabset
: Data files to set tabstops.

/usr/lib/term
: Terminal initialization files.

/usr/lib/tmac
: Macro definition packages for *nroff*(1).

/usr/lib/uucp[/*]
: Commands, configuration files, and working directories for *uucp*(1).

/usr/local
: Site–local commands, files, etc. Files under this directory come from inside the local site or organization. See */usr/contrib* for non–local unsupported commands and files.

/usr/local/bin
: Site–local commands.

/usr/local/games
: Site–local games.

/usr/local/include
: Site–local include files. To include them, you must (in C) give a complete pathname, for example, #include "/usr/local/include/symtab.h".

/usr/local/lib
: Site–local libraries.

/usr/local/man/cat[1–8]
: Site–local manual entries, post–nroff form.

/usr/local/man/man[1–8]
: Site–local manual entries, pre–nroff form.

/usr/local/man/$LANG/cat[1–8]
: Site–local manual entries, unformatted form for installed native languages. The LANG environment variable may take on values given in the **/usr/lib/nls/config** table.

/usr/local/man/$LANG/man[1–8]
: Site–local manual entries, formatted form for installed native languages.

/usr/mail
: User mailboxes.

/usr/man
: On–line documentation.

/usr/man/cat[1–8]
: Optional formatted (post–nroff) versions of on–line documentation for use by *man*(1).

/usr/man/man[1–8]
: Unformatted (pre–nroff) versions of on–line documentation for use by *man*(1).

/usr/man/$LANG
: On–line documentation for installed native languages. The LANG environment variable may take on values given in the **/usr/lib/nls/config** table.

/usr/man/$LANG/cat[1–8]
: Formatted native language versions of on–line documentation for use by *man*(1).

/usr/man/$LANG/man[1–8]
: Unformatted native language versions of on–line documentation for use by *man*(1).

/usr/news
: Local–system news articles for *news*(1).

/usr/preserve
: Place where *ex*(1) and *vi*(1) save lost edit sessions until recovered.

/usr/spool
: Spooled (queued) files for various programs.

/usr/spool/cron
: Spooled jobs for *cron*(1) and *at*(1).

/usr/spool/cron/atjobs
: Spooled jobs for *at*(1).

/usr/spool/lp
: Control and working files for *lp*(1).

/usr/spool/lp/class
: Printer class definition files.

/usr/spool/lp/interface
: Printer interface shell scripts.

/usr/spool/lp/member
: Printer class member definition files.

/usr/spool/lp/request
: Spool directories for each logical destination.

/usr/spool/uucp
: Queued work, lockfiles, logfiles, status files, and other files for *uucp*(1).

/usr/spool/uucppublic[/*]
: Publicly–accessible directory for use with *uucp*(1).

/usr/src Source files. Only present on HP–UX implementations which support source.

/usr/src/cmd/*
: Source for commands. Simple command sources reside at the top level. Subdirectories are named after specific commands, e.g. */usr/src/cmd/cc*, and contain the source for multi–file or otherwise complicated commands. Directory structure below here depends on the individual command; see the associated makefiles.

/usr/src/games/*
: Source for games. Simple game sources reside at the top level. Subdirectories are named after specific games, e.g. */usr/src/games/master*, and contain the source for multi–file or otherwise complicated games. Directory structure below here depends on the individual game; see the associated makefiles.

/usr/src/head
: Include files which are copied into */usr/include/**.

/usr/src/lib
: Source for libraries, in many subdirectories.

/usr/src/lib/libF77
: Source for FORTRAN–77 miscellaneous (mostly math) libraries.

/usr/src/lib/libI77
: Source for FORTRAN–77 I/O libraries.

/usr/src/lib/libPW
: Source for Programmer's Workbench libraries.

/usr/src/lib/libc
: Source for standard C libraries.

/usr/src/lib/libcurses/*
: Source for curses (cursor control) libraries.

/usr/src/lib/libl
: Source for lex(1) libraries.

/usr/src/lib/libm
: Source for C math libraries.

/usr/src/lib/liby
: Source for yacc(1) libraries.

/usr/tmp
: Alternate place to put temporary files; usually used when there may be very many of them or if they will be large.

SEE ALSO

ls(1), find(1), grep(1), whereis(1), hpnls(7).

HARDWARE DEPENDENCIES

Series 500 systems support shared libraries loaded by the kernel at powerup time. They reside in the directory */etc/sslibs.*

Some directories include commands or files not supported on all HP–UX implementations.

NAME

hpnls - HP Native Language Support (NLS) Model

SYNOPSIS

ls /usr/lib/nls/*

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

DESCRIPTION

The HP Native Language Support (NLS) model includes several capabilities that reduce or eliminate the barriers that would otherwise make HP-UX difficult to use in a non-English language. The three main categories, Character Set Support, Local Customs, and Messages, are subdivided into smaller categories in order to adequately reflect the extent of the Native Language Support.

CHARACTER SET SUPPORT -
A major NLS objective is to provide capabilities for adapting character sequences to local language needs.

CHARACTER CODE SIZE -
The length of the character code governs the number of distinct characters that can be included in the character set.

7-BIT - The ASCII character set consists of 33 control characters including DEL, space, and 94 printable characters. (See ascii(7).) This is sufficient to span the Latin alphabet, upper and lowercase, plus punctuation and special symbols. Seven bits of information is sufficient to distinguish the characters in such a set.

8-BIT - The use of an 8 bit character code allows 67 control codes, space, and 188 printable characters. In the case of European characters, this provides sufficient space for accented vowels, consonants with special forms, and other special symbols. (See roman8(7)). This is also sufficient to hold the phonetic Japanese character set Katakana. (See kana8(7).)

16-BIT -
A number of languages have very large character sets that require more than the 188 printable characters provided by the 8-bit character codes. Sixteen-bit character codes are available for these languages. To simplify processing, 16-bit printable characters are formed from pairs of 8-bit printable characters (neither byte may contain a control code or a space). This allows representation of up to 35344 characters.

CHARACTER TYPING -
Character processing which depends on character type must take into account the character type changes that vary with the character set being used. For example, an alphabetic character in the ROMAN8 character set may align with a punctuation character in the KANA8 set.

SHIFTING -
While the ROMAN8 character set has uppercase and lowercase for most alphabetic characters, some languages discard accents when characters are shifted to uppercase. Other alphabetic characters may not be shifted at all, when there is no notion of "case" in the underlying language.

COLLATING -
The ASCII collation order, while generally tolerated, is not adequate for American dictionary usage. Different languages sort characters from the ROMAN8 set in different orders. Some languages require that character pairs, such as "ch" and "ll" in Spanish, be sorted as single characters. Ideographic character sets may have multiple orderings. For example,

Japanese kanjis may be sorted in phonetic order; in a different 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.

DIRECTIONALITY -

The assumption that displayed text goes from left to right does not hold for all languages. Some Middle Eastern languages go from right to left. Far Eastern languages usually use vertical columns, starting from the right.

CODING SCHEME CONSIDERATIONS -

Although most HP supported 8-bit character sets preserve the ASCII codes in the range of 0 to 127, 16-bit character sets may use these byte values in 2-byte characters. Software that assigns special meaning to bytes (metacharacters) in this range must distinguish between 1-byte and 2-byte characters. In multilingual environments, standard escape code sequences are used to indicate change to alternate character sets. Since these sequences are not usually printed or displayed, the number of characters output is usually less than the number of bytes in the sequence. Any software that must locate a character within a sequence must accommodate this.

LOCAL CUSTOMS -

Some aspects of Native Language Support relate more to local customs of a particular geographic location than to the characters used to write the language.

REPRESENTATION OF NUMBERS -

The character used to denote the radix of a decimal number varies for different regions. Similarly the use of a "thousands" indicator or grouping of (usually three) digits may vary with local custom.

CURRENCY REPRESENTATION -

The symbol for currency varies from country to country. The symbol may either precede or follow the numeric value. Some currencies allow decimal fractions while others use alternate methods of representing smaller monetary values.

DATE AND TIME REPRESENTATION -

Month and weekday names vary with language (if they are not omitted entirely). Abbreviations may be other than three characters, or may not be allowed at all. Even when a strictly numeric representation is used, the order of year, month, and day as well as the delimiters which separate them is not universal.

DATE AND TIME ADJUSTMENTS -

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. The Gregorian calendar is most common, but some locales use different methods for determining meridian day and year; usually based on seasonal, astronomical, or historical events.

MESSAGES -

The need for messages to be readable by users is perhaps the most significant justification for implementing Native Language Support.

MESSAGE CONTENT -

Error messages, prompts, expected responses, and mnemonic command names should be based on the user's native language.

MESSAGE STRUCTURE -

Messages must often be built from segments. To accommodate grammatical differences, it may be necessary to change the order in which the fragments are connected.

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.

FILES

/usr/lib/nls/*

SEE ALSO

date(1), sort(1), ctime(3C), ecvt(3C), getmsg(3C), langinfo(3C), nl_conv(3C), nl_ctype(3C), nl_string(3C), printmsg(3C), strtod(3C), ascii(7), kana8(7), roman8(7).

NAME

kana8 - map of KANA8 character set used by NLS

SYNOPSIS

ls /usr/lib/nls/*

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

DESCRIPTION

Kana8 is a map of the KANA8 character set, giving the octal, decimal, and hexadecimal equivalents of each character, to be printed as needed. It contains:

000	0	00	nul	001	1	01	soh
002	2	02	stx	003	3	03	etx
004	4	04	eot	005	5	05	enq
006	6	06	ack	007	7	07	bel
010	8	08	bs	011	9	09	ht
012	10	0a	nl	013	11	0b	vt
014	12	0c	np	015	13	0d	cr
016	14	0e	so	017	15	0f	si
020	16	10	dle	021	17	11	dc1
022	18	12	dc2	023	19	13	dc3
024	20	14	dc4	025	21	15	nak
026	22	16	syn	027	23	17	etb
030	24	18	can	031	25	19	em
032	26	1a	sub	033	27	1b	esc
034	28	1c	fs	035	29	1d	gs
036	30	1e	rs	037	31	1f	us
040	32	20	sp	041	33	21	!
042	34	22	"	043	35	23	#
044	36	24	$	045	37	25	%
046	38	26	&	047	39	27	'
050	40	28	(	051	41	29	)
052	42	2a	*	053	43	2b	+
054	44	2c	,	055	45	2d	-
056	46	2e	.	057	47	2f	/
060	48	30	0	061	49	31	1
062	50	32	2	063	51	33	3
064	52	34	4	065	53	35	5
066	54	36	6	067	55	37	7
070	56	38	8	071	57	39	9
072	58	3a	:	073	59	3b	;
074	60	3c	<	075	61	3d	=
076	62	3e	>	077	63	3f	?
100	64	40	@	101	65	41	A
102	66	42	B	103	67	43	C
104	68	44	D	105	69	45	E
106	70	46	F	107	71	47	G

110	72	48	H	111	73	49	I
112	74	4a	J	113	75	4b	K
114	76	4c	L	115	77	4d	M
116	78	4e	N	117	79	4f	O
120	80	50	P	121	81	51	Q
122	82	52	R	123	83	53	S
124	84	54	T	125	85	55	U
126	86	56	V	127	87	57	W
130	88	58	X	131	89	59	Y
132	90	5a	Z	133	91	5b	[
134	92	5c	¥ yen	135	93	5d	]
136	94	5e	^	137	95	5f	_
140	96	60	`	141	97	61	a
142	98	62	b	143	99	63	c
144	100	64	d	145	101	65	e
146	102	66	f	147	103	67	g
150	104	68	h	151	105	69	i
152	106	6a	j	153	107	6b	k
154	108	6c	l	155	109	6d	m
156	110	6e	n	157	111	6f	o
160	112	70	p	161	113	71	q
162	114	72	r	163	115	73	s
164	116	74	t	165	117	75	u
166	118	76	v	167	119	77	w
170	120	78	x	171	121	79	y
172	122	7a	z	173	123	7b	{
174	124	7c	|	175	125	7d	}
176	126	7e	~	177	127	7f	del
200	128	80		201	129	81	
202	130	82		203	131	83	
204	132	84		205	133	85	
206	134	86		207	135	87	
210	136	88		211	137	89	
212	138	8a		213	139	8b	
214	140	8c		215	141	8d	
216	142	8e	ss2	217	143	8f	ss3
220	144	90		221	145	91	
222	146	92		223	147	93	
224	148	94		225	149	95	
226	150	96		227	151	97	
230	152	98		231	153	99	
232	154	9a		233	155	9b	
234	156	9c		235	157	9d	
236	158	9e		237	159	9f	
240	160	a0		241	161	a1	｡ ku-ten
242	162	a2	｢ hook	243	163	a3	｣ unhook
244	164	a4	､ to-ten	245	165	a5	･ dot
246	166	a6	ｦ wo	247	167	a7	ｧ small a
250	168	a8	ｨ small i	251	169	a9	ｩ small u
252	170	aa	ｪ small e	253	171	ab	ｫ small o
254	172	ac	ｬ small ya	255	173	ad	ｭ small yu
256	174	ae	ｮ small yo	257	175	af	ｯ small tsu

260	176	b0	-	dash	261	177	b1	ｱ	a
262	178	b2	ｲ	i	263	179	b3	ｳ	u
264	180	b4	ｴ	e	265	181	b5	ｵ	o
266	182	b6	ｶ	ka	267	183	b7	ｷ	ki
270	184	b8	ｸ	ku	271	185	b9	ｹ	ke
272	186	ba	ｺ	ko	273	187	bb	ｻ	sa
274	188	bc	ｼ	shi	275	189	bd	ｽ	su
276	190	be	ｾ	se	277	191	bf	ｿ	so
300	192	c0	ﾀ	ta	301	193	c1	ﾁ	chi
302	194	c2	ﾂ	tsu	303	195	c3	ﾃ	te
304	196	c4	ﾄ	to	305	197	c5	ﾅ	na
306	198	c6	ﾆ	ni	307	199	c7	ﾇ	nu
310	200	c8	ﾈ	ne	311	201	c9	ﾉ	no
312	202	ca	ﾊ	ha	313	203	cb	ﾋ	hi
314	204	cc	ﾌ	fu	315	205	cd	ﾍ	he
316	206	ce	ﾎ	ho	317	207	cf	ﾏ	ma
320	208	d0	ﾐ	mi	321	209	d1	ﾑ	mu
322	210	d2	ﾒ	me	323	211	d3	ﾓ	mo
324	212	d4	ﾔ	ya	325	213	d5	ﾕ	yu
326	214	d6	ﾖ	yo	327	215	d7	ﾗ	ra
330	216	d8	ﾘ	ri	331	217	d9	ﾙ	ru
332	218	da	ﾚ	re	333	219	db	ﾛ	ro
334	220	dc	ﾜ	wa	335	221	dd	ﾝ	n
336	222	de	"	voiced	337	223	df	°	degree
340	224	e0			341	225	e1		
342	226	e2			343	227	e3		
344	228	e4			345	229	e5		
346	230	e6			347	231	e7		
350	232	e8			351	233	e9		
352	234	ea			353	235	eb		
354	236	ec			355	237	ed		
356	238	ee			357	239	ef		
360	240	f0			361	241	f1		
362	242	f2			363	243	f3		
364	244	f4			365	245	f5		
366	246	f6			367	247	f7		
370	248	f8			371	249	f9		
372	250	fa			373	251	fb		
374	252	fc			375	253	fd		
376	254	fe			377	255	ff		

FILES

/usr/lib/nls/*

SEE ALSO

ascii(7), hpnls(7), roman8(7).

WARNINGS

Peripheral or software limitations may garble this manual page. Many printers and terminals do not support the KANA8 character set.

NAME

langid - language identification variable used with NLS

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

DESCRIPTION

This page defines integer values corresponding to values of the variable *LANG* in the user's environment. These are the values returned by *currlangid(3C)* , and are passed as parameters into native language support library routines.

LANGUAGE NAMES

The following languages are currently supported by HP-UX.

Language Num	Abbreviation	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	

SEE ALSO

langinfo(3C), environ(7), hier(7), hpnls(7).

BUGS

Currently only supported in 'sh'.

NAME

man - macros for formatting entries in this manual

SYNOPSIS

nroff -man files

HP-UX COMPATIBILITY

Level: Text Processing - HP-UX/STANDARD

Origin: System V

DESCRIPTION

These *nroff*(1) macros are used to lay out the format of the entries of this manual. These macros are used by the *man*(1) command.

The default page size is 8.5″×11″, with a 6.5″×10″ text area. The **-rV2** option may be used to set certain parameters to values appropriate for certain Versatec printers: it sets the line length to 82 characters, the page length to 84 lines, and it inhibits underlining.

Any *text* argument below may be one to six "words". Double quotes ("") may be used to include blanks in a "word". If *text* is empty, the special treatment is applied to the next line that contains text to be printed. For example, **.I** may be used to italicize a whole line, or **.SM** followed by **.B** to make small bold text. By default, hyphenation is turned off.

Type font and size are reset to default values before each paragraph and after processing font- and size-setting macros, e.g., **.I**, **.RB**, **.SM.** Tab stops are neither used nor set by any macro except **.DT** and **.TH.**

Default units for indents *in* are ens. When *in* is omitted, the previous indent is used. This remembered indent is set to its default value (5 ens in *nroff*-this corresponds to 0.5″ in the default page size) by **.TH**, **.P**, and **.RS**, and restored by **.RE.**

.TH *t s c n*	Set the title and entry heading; *t* is the title, *s* is the section number, *c* is extra commentary, e.g., "local", *n* is new manual name. Invokes **.DT** (see below).
.SH *text*	Place subhead *text*, e.g., **SYNOPSIS**, here.
.SS *text*	Place sub-subhead *text*, e.g., **Options**, here.
.B *text*	Make *text* bold.
.I *text*	Make *text* italic.
.SM *text*	Make *text* 1 point smaller than default point size.
.RI *a b*	Concatenate roman *a* with italic *b*, and alternate these two fonts for up to six arguments. Similar macros alternate between any two of roman, italic, and bold: **.IR .RB .BR .IB .BI**
.P	Begin a paragraph with normal font, point size, and indent. **.PP** is a synonym for **.P**.
.HP *in*	Begin paragraph with hanging indent.
.TP *in*	Begin indented paragraph with hanging tag. The next line that contains text to be printed is taken as the tag. If the tag does not fit, it is printed on a separate line.
.IP *t in*	Same as **.TP** *in* with tag *t*; often used to get an indented paragraph without a tag.
.RS *in*	Increase relative indent (initially zero). Indent all output an extra *in* units from the current left margin.
.RE *k*	Return to the *k*th relative indent level (initially, *k*=1; *k*=0 is equivalent to *k*=1); if *k* is omitted, return to the most recent lower indent level.
.PM *m*	Produces proprietary markings; where *m* may be **P** for **PRIVATE**, or **N** for **NOTICE.**
.DT	Restore default tab settings (every 5 ens in *nroff*).
.PD *v*	Set the interparagraph distance to *v* vertical spaces. If *v* is omitted, set the interparagraph distance to the default value (1v in *nroff*).

The following *strings* are defined:

\*R	"(Reg.)" in *nroff*.
\*S	Change to default type size.
\*(Tm	Trademark indicator.

The following *number registers* are given default values by **.TH**:

IN	Left margin indent relative to subheads (default is 5 ens in *nroff*).
LL	Line length including **IN**.
PD	Current interparagraph distance.

CAVEATS

In addition to the macros, strings, and number registers mentioned above, there are defined a number of *internal* macros, strings, and number registers. Except for names predefined by *nroff* and number registers **d**, **m**, and **y**, all such internal names are of the form *XA*, where *X* is one of),], and }, and *A* stands for any alphanumeric character.

If a manual entry needs to be preprocessed by *tbl*(1), it must begin with a special line (described in *man*(1)), causing the *man* command to invoke the appropriate preprocessor(s).

The programs that prepare the Table of Contents and the Permuted Index for this Manual assume the *NAME* section of each entry consists of a single line of input that has the following format:

```
name[, name, name ...] \- explanatory text
```

The macro package increases the inter–word spaces (to eliminate ambiguity) in the *SYNOPSIS* section of each entry.

The macro package itself uses only the roman font (so that one can replace, for example, the bold font with a different font). Of course, if the input text of an entry contains requests for other fonts (e.g., **.I**, **.RB**, **\fI**), the corresponding fonts must be mounted.

FILES

/usr/lib/tmac/tmac.an
/usr/lib/macros/cmp.n.[dt].an
/usr/lib/macros/ucmp.n.an

SEE ALSO

man(1), nroff(1).

BUGS

If the argument to **.TH** contains *any* blanks and is *not* enclosed by double quotes (""), the output can be incorrectly formatted.

NAME

math - math functions and constants

SYNOPSIS

#include <math.h>

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: System V

DESCRIPTION

This file contains declarations of all the functions in the Math Library (described in Section 3M), as well as various functions in the C Library (Section 3C) that return floating-point values.

It defines the structure and constants used by the *matherr*(3M) error-handling mechanisms, including the following constant used as an error-return value:

HUGE	The maximum value of a single-precision floating-point number.
MAXFLOAT	The maximum value of a single-precision floating-point number.

For the definitions of various machine-dependent "constants," see the description of the <*values.h*> header file.

FILES

/usr/include/math.h

SEE ALSO

intro(3), matherr(3M), values(7).

NAME

mm - the MM macro package for formatting documents

SYNOPSIS

mm [options] [files]

nroff -mm [options] [files]

nroff -cm [options] [files]

HP-UX COMPATIBILITY

Level:	Text Processing - HP-UX/STANDARD
Origin:	System V

DESCRIPTION

This package provides a formatting capability for a very wide variety of documents. The manner in which a document is typed in and edited is essentially independent of whether the document is to be eventually formatted at a terminal or is to be phototypeset. See the references below for further details.

The **-mm** option causes *nroff* and *troff*(1) to use the non-compacted version of the macro package, while the **-cm** option results in the use of the compacted version, thus speeding up the process of loading the macro package.

FILES

/usr/lib/tmac/tmac.m	pointer to the non-compacted version of the package
/usr/lib/macros/mmn	non-compacted version of the package
/usr/lib/macros/cmp.n.[dt].m	compacted version of the package
/usr/lib/macros/ucmp.[nt].m	initializers for the compacted version of the package

SEE ALSO

mm(1), nroff(1).
MM-Memorandum Macros in *HP-UX Concepts and Tutorials.*

HARDWARE DEPENDENCIES

Compacted macros are not suppoprted on Series 500 implementations.

NAME

INIT, GETC, PEEKC, UNGETC, RETURN, ERROR, compile, step, advance - regular expression compile and match routines

SYNOPSIS

#define INIT <declarations>
#define GETC( ) <getc code>
#define PEEKC( ) <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *compile (instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;
int eof;

int step (string, expbuf)
char *string, *expbuf;

int advance (string, expbuf)
char *string, *expbuf;

extern char *loc1, *loc2, *locs;

extern int circf, sed, nbra;

HP-UX COMPATIBILITY

Level:	HP-UX/RUN ONLY
Origin:	System III

DESCRIPTION

This page describes general-purpose regular expression matching routines in the form of *ed*(1), defined in **/usr/include/regexp.h.** Programs such as *ed*(1), *sed*(1), *grep*(1), *bs*(1), *expr*(1), etc., which perform regular expression matching use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to this file is complex. Programs that include this file must have the following five macros declared before the "#include <regexp.h>" statement. These macros are used by the *compile* routine.

GETC( )	Return the value of the next character in the regular expression pattern. Successive calls to GETC( ) should return successive characters of the regular expression.
PEEKC( )	Return the next character in the regular expression. Successive calls to PEEKC( ) should return the same character (which should also be the next character returned by GETC( )).
UNGETC(*c*)	Cause the argument *c* to be returned by the next call to GETC( ) (and PEEKC( )). No more than one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC( ). The value of the macro UNGETC(*c*) is always ignored.
RETURN(*pointer*)	This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.
ERROR(*val*)	This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never

return.

ERROR	MEANING
11	Range endpoint too large.
16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the *compile* routine is as follows:

```
compile(instring, expbuf, endbuf, eof)
```

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of ((char *) 0) for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address that the compiled regular expression may occupy. If the compiled expression cannot fit in (*endbuf-expbuf*) bytes, a call to ERROR(50) is made.

The parameter *eof* is the character which marks the end of the regular expression. For example, in *ed*(1), this character is usually a /.

Each program that includes this file must have a **#define** statement for INIT. This definition will be placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for GETC( ), PEEKC( ) and UNGETC( ). Otherwise it can be used to declare external variables that might be used by GETC( ), PEEKC( ) and UNGETC( ). See the example below of the declarations taken from *grep*(1).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. The call to *step* is as follows:

```
step(string, expbuf)
```

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter *expbuf* is the compiled regular expression which was obtained by a call of the function *compile*.

The function *step* returns non-zero if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, *loc1* will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

Step uses the external variable *circf* which is set by *compile* if the regular expression begins with ^. If this is set then *step* will try to match the regular expression to the beginning of the string only. If more than one regular expression is to be compiled before the first is executed the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance* until *advance* returns non–zero indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called; simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* will back up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string where the match first occurred at sometime during the backing up process, *advance* will break out of the loop that backs up and will return zero. This is used by *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like **s/y*//g** do not loop forever.

The additional external variables *sed* and *nbra* are used for special purposes.

EXAMPLES

The following is an example of how the regular expression macros and calls look from an old version of *grep*(1):

```
#define INIT            register char *sp = instring;
#define GETC()          (*sp++)
#define PEEKC()         (*sp)
#define UNGETC(c)       (--sp)
#define RETURN(c)       return;
#define ERROR(c)        regerr()

#include <regexp.h>
...
        (void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
        if (step(linebuf, expbuf))
                succeed();
```

FILES

/usr/include/regexp.h

SEE ALSO

bs(1), ed(1), expr(1), grep(1), sed(1).

BUGS

The handling of *circf* is poor.
The actual code is probably easier to understand than this manual page.

NAME

roman8 - map of ROMAN8 character set used by NLS

SYNOPSIS

ls /usr/lib/nls/*

HP-UX COMPATIBILITY

Level: HP-UX/STANDARD

Origin: HP

DESCRIPTION

Roman8 is a map of the ROMAN8 character set, giving the octal, decimal, and hexadecimal equivalents of each character, to be printed as needed. It contains:

000	0	00	nul	001	1	01	soh
002	2	02	stx	003	3	03	etx
004	4	04	eot	005	5	05	enq
006	6	06	ack	007	7	07	bel
010	8	08	bs	011	9	09	ht
012	10	0a	nl	013	11	0b	vt
014	12	0c	np	015	13	0d	cr
016	14	0e	so	017	15	0f	si
020	16	10	dle	021	17	11	dc1
022	18	12	dc2	023	19	13	dc3
024	20	14	dc4	025	21	15	nak
026	22	16	syn	027	23	17	etb
030	24	18	can	031	25	19	em
032	26	1a	sub	033	27	1b	esc
034	28	1c	fs	035	29	1d	gs
036	30	1e	rs	037	31	1f	us
040	32	20	sp	041	33	21	!
042	34	22	"	043	35	23	#
044	36	24	$	045	37	25	%
046	38	26	&	047	39	27	'
050	40	28	(	051	41	29	)
052	42	2a	*	053	43	2b	+
054	44	2c	,	055	45	2d	-
056	46	2e	.	057	47	2f	/
060	48	30	0	061	49	31	1
062	50	32	2	063	51	33	3
064	52	34	4	065	53	35	5
066	54	36	6	067	55	37	7
070	56	38	8	071	57	39	9
072	58	3a	:	073	59	3b	;
074	60	3c	<	075	61	3d	=
076	62	3e	>	077	63	3f	?
100	64	40	@	101	65	41	A
102	66	42	B	103	67	43	C
104	68	44	D	105	69	45	E
106	70	46	F	107	71	47	G

110	72	48	H	111	73	49	I
112	74	4a	J	113	75	4b	K
114	76	4c	L	115	77	4d	M
116	78	4e	N	117	79	4f	O
120	80	50	P	121	81	51	Q
122	82	52	R	123	83	53	S
124	84	54	T	125	85	55	U
126	86	56	V	127	87	57	W
130	88	58	X	131	89	59	Y
132	90	5a	Z	133	91	5b	[
134	92	5c	\	135	93	5d	]
136	94	5e	^	137	95	5f	_
140	96	60	`	141	97	61	a
142	98	62	b	143	99	63	c
144	100	64	d	145	101	65	e
146	102	66	f	147	103	67	g
150	104	68	h	151	105	69	i
152	106	6a	j	153	107	6b	k
154	108	6c	l	155	109	6d	m
156	110	6e	n	157	111	6f	o
160	112	70	p	161	113	71	q
162	114	72	r	163	115	73	s
164	116	74	t	165	117	75	u
166	118	76	v	167	119	77	w
170	120	78	x	171	121	79	y
172	122	7a	z	173	123	7b	{
174	124	7c	|	175	125	7d	}
176	126	7e	~	177	127	7f	del
200	128	80		201	129	81	
202	130	82		203	131	83	
204	132	84		205	133	85	
206	134	86		207	135	87	
210	136	88		211	137	89	
212	138	8a		213	139	8b	
214	140	8c		215	141	8d	
216	142	8e	ss2	217	143	8f	ss3
220	144	90		221	145	91	
222	146	92		223	147	93	
224	148	94		225	149	95	
226	150	96		227	151	97	
230	152	98		231	153	99	
232	154	9a		233	155	9b	
234	156	9c		235	157	9d	
236	158	9e		237	159	9f	
240	160	a0		241	161	a1	À A accent grave
242	162	a2	Â A circumflex	243	163	a3	È E accent grave
244	164	a4	Ê E circumflex	245	165	a5	Ë E umlaut
246	166	a6	Î I circumflex	247	167	a7	Ï I umlaut
250	168	a8	´ accent acute	251	169	a9	` accent grave
252	170	aa	ˆ circumflex	253	171	ab	¨ umlaut accent
254	172	ac	˜ tilde accent	255	173	ad	Ù U accent grave
256	174	ae	Û U circumflex	257	175	af	£ Italian lira

260	176	b0	‾	over line	261	177	b1		
262	178	b2			263	179	b3	°	degree
264	180	b4	Ç	C cedilla	265	181	b5	ç	c cedilla
266	182	b6	Ñ	N tilde	267	183	b7	ñ	n tilde
270	184	b8	¡	inv. exclamation	271	185	b9	¿	inv. question
272	186	ba	¤	general currency	273	187	bb	£	British pound
274	188	bc	¥	Japanese yen	275	189	bd	§	section
276	190	be	ƒ	Dutch guilder	277	191	bf	¢	U.S. cent
300	192	c0	â	a circumflex	301	193	c1	ê	e circumflex
302	194	c2	ô	o circumflex	303	195	c3	û	u circumflex
304	196	c4	á	a accent acute	305	197	c5	é	e accent acute
306	198	c6	ó	o accent acute	307	199	c7	ú	u accent acute
310	200	c8	à	a accent grave	311	201	c9	è	e accent grave
312	202	ca	ò	o accent grave	313	203	cb	ù	u accent grave
314	204	cc	ä	a umlaut	315	205	cd	ë	e umlaut
316	206	ce	ö	o umlaut	317	207	cf	ü	u umlaut
320	208	d0	Å	A degree	321	209	d1	î	i circumflex
322	210	d2	Ø	O crossbar	323	211	d3	Æ	AE ligature
324	212	d4	å	a degree	325	213	d5	í	i accent acute
326	214	d6	ø	o crossbar	327	215	d7	æ	ae ligature
330	216	d8	Ä	A umlaut	331	217	d9	ì	i accent grave
332	218	da	Ö	O umlaut	333	219	db	Ü	U umlaut
334	220	dc	É	E accent acute	335	221	dd	ï	i umlaut
336	222	de	ß	sharp s	337	223	df	Ô	O circumflex
340	224	e0	Á	A accent acute	341	225	e1	Ã	A tilde
342	226	e2	ã	a tilde	343	227	e3	Đ	D stroke
344	228	e4	đ	d stroke	345	229	e5	Í	I accent acute
346	230	e6	Ì	I accent grave	347	231	e7	Ó	O accent acute
350	232	e8	Ò	O accent grave	351	233	e9	Õ	O tilde
352	234	ea	õ	o tilde	353	235	eb	Š	S caron
354	236	ec	š	s caron	355	237	ed	Ú	U accent acute
356	238	ee	Ÿ	Y umlaut	357	239	ef	ÿ	y umlaut
360	240	f0	Þ	THORN	361	241	f1	þ	thorn
362	242	f2			363	243	f3		
364	244	f4			365	245	f5		
366	246	f6	–	long dash	367	247	f7	¼	one fourth
370	248	f8	½	one half	371	249	f9	ª	femin. ordinal
372	250	fa	º	masc. ordinal	373	251	fb	«	open guillemets
374	252	fc	■	solid	375	253	fd	»	close guillemets
376	254	fe	±	plus/minus	377	255	ff		

FILES

/usr/lib/nls/*

SEE ALSO

ascii(7), hpnls(7), kana8(7).

WARNINGS

Peripheral or software limitations may garble this manual page. Some printers and terminals do not support the ROMAN8 character set.

NAME

stat - data returned by stat/fstat system call

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: System III

DESCRIPTION

The system calls *stat* and *fstat*(2) return data whose structure is defined by this include file. The encoding of the field *st_mode* is defined in this file also.

```
/*
 * Structure of the result of stat
 */

struct  stat
{
        dev_t   st_dev;
        ino_t   st_ino;
        ushort  st_mode;
        short   st_nlink;
        ushort  st_uid;
        ushort  st_gid;
        dev_t   st_rdev;
        off_t   st_size;
        time_t  st_atime;
        time_t  st_mtime;
        time_t  st_ctime;
};


#define S_IFMT      0170000     /* type of file */
#define S_IFDIR     0040000     /* directory */
#define S_IFCHR     0020000     /* character special */
#define S_IFBLK     0060000     /* block special */
#define S_IFREG     0100000     /* regular */
#define S_IFIFO     0010000     /* fifo */
#define S_IFNWK     0110000     /* network special */
#define S_ISUID     04000       //* set user id on execution */
#define S_ISGID     02000       //* set group id on execution */
#define S_ISVTX     01000       //* save swapped text even after use */
#define S_IREAD     00400       //* read permission, owner */
#define S_IWRITE    00200       / write permission, owner */
#define S_IEXEC     00100       //* execute/search permission, owner */
```

FILES

/usr/include/sys/types.h
/usr/include/sys/stat.h

SEE ALSO

stat(2), types(5).

HARDWARE DEPENDENCIES

Integral PC:

The S_IFNWK i–node type is not supported.

NAME

term - conventional names for terminals

HP-UX COMPATIBILITY

Level: HP-UX/RUN ONLY

Origin: System III and UCB

DESCRIPTION

The environment variable TERM is used by certain commands (e.g., *tabs*(1), and is maintained as part of the shell environment (see *profile*(4), and *environ*(5)) The *tset*(1) command can be used to set the TERM variable When *tset* is used, the name to which TERM is set must be listed in the */terminfo* data base (see *terminfo*(5)).

hpsub	Minimal subset of the capabilities of all Hewlett-Packard terminals and terminal emulators supported on both Series 500 and Series 200 HP-UX.
hp	Minimal subset of the capabilities of Hewlett-Packard terminals supported on both Series 200 and Series 500 HP-UX (does not include 98x6 Internal Terminal Emulator).
9836	Internal Terminal Emulator (ITE) for the HP 9000 Models 236 and 220 computers.
9826	Internal Terminal Emulator (ITE) for the HP 9000 Model 226 computer.
262x	Hewlett-Packard 262x family. Includes the HP 2622, HP 2623, and HP 2624 terminals.
2622	Hewlett-Packard HP 2622 terminal.
2623	Hewlett-Packard HP 2623 graphics terminal.
2624	Hewlett-Packard HP 2624 terminal.

Other terminal names included in the */terminfo* data base do not imply support of those terminals.

The TERM variable is also used by certain commands (e.g. *nroff*(1), *man*(1), *tabs*(1)), some of which use terminal and printer description files from the */usr/lib/terms* directory. TERM names which have files in this directory include the following (note that the publication of these names and presence of these files does not imply support of these devices):

2631	Hewlett-Packard 2631 line printer.
2631-c	Hewlett-Packard 2631 line printer - compressed mode.
2631-e	Hewlett-Packard 2631 line printer - expanded mode.
300	DASI/DTC/GSI 300 and others using the Hy Type I printer.
300-12	Same as **300**, in 12-pitch mode.
300s	DASI/DTC/GSI 300s
300s-12	Same as **300s**, in 12-pitch mode.
382	DTC 382.
37	TELETYPE Model 27 KSR.
4000A	Trendata 4000A.
450	DASI 450 (same as Diablo 1620).
450-12	Same as **450**, in 12-pitch mode.
lp	Generic name for a line printer.
tn300	General Electric TermiNet 300.

A basic terminal name can be up to eight characters chosen from A-Z, a-z, 0-9, and -. Terminal sub-models and operational modes are distinguished by suffixes beginning with a -. Names should generally be based on original vendors, rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name.

Commands whose behavior depends on the type of terminal should accept arguments of the form **-T***term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable **$TERM**, which, in turn, should contain *term*.

SEE ALSO

ex(1), man(1), mm(1), more(1), nroff(1), sh(1), stty(1), tabs(1), tset(1), ul(1), curses(3), terminfo(5), profile(5), ttytype(5), environ(7).

NAME
types - primitive system data types

SYNOPSIS
#include <sys/types.h>

HP-UX COMPATIBILITY
Level: HP-UX/RUN ONLY

Origin: System III

Remarks: The example given on this page is a typical version; the type names are in general expected to be present, although exceptions can be described in HARDWARE DEPENDENCIES. The fundamental type which implements each typedef is implemetation dependent, as long as source code which uses those typedefs need not be changed.

DESCRIPTION
The data types defined in the include file are used in HP-UX system code; some data of these types are accessible to user code:

```
#define  NREGS_S          13                  /* no. of regs saved */

typedef  struct { int r[1]; } *               physadr;
typedef  long             daddr_t;
typedef  char *           caddr_t;
typedef  unsigned int     uint;
typedef  unsigned short   ushort;
typedef  ushort           ino_t;
typedef  short            cnt_t;
typedef  long             time_t;
typedef  int              label_t[NREGS_S];
typedef  long             dev_t;
typedef  long             off_t;
typedef  long             paddr_t;
typedef  long             key_t;
```

Note that the defined names above are standardized, but the actual type to which they are defined may vary between HP-UX implementations.

The form *daddr_t* is used for disc addresses except in an i-node on disc, see *fs*(5). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The *label_t* variables are used to save the processor state while another process is running.

HARDWARE DEPENDENCIES
Series 500:

The types NREGS_S and *label_t* are not implemented.

SEE ALSO
fs(5).

NAME
values - machine–dependent values

SYNOPSIS
#include <values.h>

HP–UX COMPATIBILITY
Level: HP–UX/STANDARD

Origin: System V Release 2

DESCRIPTION
This file contains a set of manifest constants, conditionally defined for particular processor architectures.

The model assumed for integers is binary representation (one's or two's complement), where the sign is represented by the value of the high–order bit.

BITS(*type*)	The number of bits in a specified type (e.g., int).
HIBITS	The value of a short integer with only the high–order bit set (in most implementations, 0x8000).
HIBITL	The value of a long integer with only the high–order bit set (in most implementations, 0x80000000).
HIBITI	The value of a regular integer with only the high–order bit set (usually the same as HIBITS or HIBITL).
MAXSHORT	The maximum value of a signed short integer (in most implementations, 0x7FFF ≡ 32767).
MAXLONG	The maximum value of a signed long integer (in most implementations, 0x7FFFFFFF ≡ 2147483647).
MAXINT	The maximum value of a signed regular integer (usually the same as MAXSHORT or MAXLONG).
MAXFLOAT, LN_MAXFLOAT	The maximum value of a single–precision floating–point number, and its natural logarithm.
MAXDOUBLE, LN_MAXDOUBLE	The maximum value of a double–precision floating–point number, and its natural logarithm.
MINFLOAT, LN_MINFLOAT	The minimum positive value of a single–precision floating–point number, and its natural logarithm.
MINDOUBLE, LN_MINDOUBLE	The minimum positive value of a double–precision floating–point number, and its natural logarithm.
FSIGNIF	The number of significant bits in the mantissa of a single–precision floating–point number.
DSIGNIF	The number of significant bits in the mantissa of a double–precision floating–point number.

FILES
/usr/include/values.h

SEE ALSO
intro(3), math(7).

NAME

varargs - handle variable argument list

SYNOPSIS

#include <varargs.h>

va_alist

va_dcl

void va_start(pvar)
va_list pvar;

type **va_arg(pvar,** *type***)**
va_list pvar;

void va_end(pvar)
va_list pvar;

DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists (such as *printf*(3S)) but do not use *varargs* are inherently nonportable, as different machines use different argument-passing conventions.

va_alist is used as the parameter list in a function header.

va_dcl is a declaration for *va_alist*. No semicolon should follow *va_dcl.*

va_list is a type defined for the variable used to traverse the list.

va_start is called to initialize *pvar* to the beginning of the list.

va_arg will return the next argument in the list pointed to by *pvar*. *Type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

va_end is used to clean up.

Multiple traversals, each bracketed by *va_start* ... *va_end,* are possible.

EXAMPLE

This example is a possible implementation of *execl*(2).

```
#include <varargs.h>
#define MAXARGS     100

/*      execl is called by
                execl(file, arg1, arg2, ..., (char *)0);
*/
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[MAXARGS];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while ((args[argno++] = va_arg(ap, char *)) != (char *)0)
                ;
        va_end(ap);
        return execv(file, args);
        }
```

SEE ALSO

exec(2), printf(3S).

BUGS

It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine this from the stack frame. For example, *execl* is passed a zero pointer to signal the end of the list. *Printf* can tell how many arguments are there by the format.

It is non–portable to specify a second argument of *char*, *short*, or *float* to *va_arg*, since arguments seen by the called function are not *char*, *short*, or *float*. C converts *char* and *short* arguments to *int* and converts *float* arguments to *double* before passing them to a function.

NAME

intro - introduction to system maintenance procedures

DESCRIPTION

This section outlines certain procedures that will be of interest to those charged with the task of system maintenance. Included are discussions on such topics as boot procedures, recovery from crashes, file backups, etc.

SEE ALSO

Section 1M. No manual pages are included in Section 8 for this printing. Commands formerly in this section have been moved to Section 1M.

NAME

intro - introduction to glossary section

DESCRIPTION

This section contains a glossary of common HP–UX terms. References to other HP–UX documentation are included as appropriate. References to entities such as *wait*(2), *sh*(1), or *fopen*(3S) refer to entries in the HP–UX Reference manual. References to items in italics but having no parenthetical suffixes refer to other entries in the glossary. Finally, any references to italicized manuals refer to separate manuals that are included with your system.

.o ("dot-oh")	A general name for an object file; also the format of an unlinked object file. See *a.out*.
absolute path name	A path name beginning with a slash (/). It indicates that the file's location is given relative to the root directory (/), and that the search begins there.
access	Access to system resources is governed by three entities: the effective user ID, the effective group ID, and the group access list.
access groups	The group access list is an additional set of group ID's used only in determining resource accessibility. Access checks are performed as described below in file access permissions.
address	In the context of peripheral devices, a set of values which specify the location of an I/O device to the computer. The exact details of the formation of an address differ between systems. On the Series 200 and 500, the address is composed of up to four elements: the select code, bus address, unit number (id), and volume number (id).
affiliation	See *terminal affiliation.*
a.out	*a.out* is the default output file name used by the linker, *ld*(1), and the C compiler, *cc*(1). It is also the format of executable object code files on HP-UX. The format is machine-dependent, and is described in the *a.out*(5) reference page for each implementation. Object code which is as yet unlinked is in the same format, but is referred to as a *.o* ("dot-oh") file.
archive	A file which is made up of the contents of other files (such as a group of object (usually *.o*) files to be used by the linker, *ld*(1)). An archive file is created and maintained by *ar*(1), or by similar programs, such as *tar*(1) or *cpio*(1). (Note that *tar*(1) and *cpio*(1) files are not usually *.o* files.) An archive is often called a library.
ASCII	An acronym for American Standard Code for Information Interchange. It consists of a set of characters including letters, numerals, punctuation, and control characters, each of which is represented internally by 7 bits (0 - 127).
asynchronous IO	An IO operation for which the user process need not wait for completion before continuing execution.
backup	The process of making a copy of all or part of the file system in order to preserve it should files be accidentally removed or destroyed (due to a power failure, hardware error, user mishap, etc.). This is a highly recommended practice.
block	(1) The fundamental unit of information HP-UX uses for access and storage allocation on a mass storage medium. The size of a block varies between implementations. On the Series 200 it varies from 1K to 8K bytes; for the Series 500, see *logical block size.* (2) On media such as 9 track tape which write variable length strings of data, *block* is equivalent to the size of those strings. Block is often used to distinguish from *record* with a block containing several records, with the number of records being the

	blocking factor.
block special file	A special file associated with a mass storage device (such as a disc or a CS–80 tape cartridge drive) that transfers data by first putting it in the buffer cache and then passing it to the user process. If the user process requests data from a mass storage device that already has the data in the buffer cache, then no I/O to the mass storage device is necessary. Block special files may be mounted.
boot or *boot–up*	The process of loading, initializing, and running an operating system.
boot area	On the Series 200, a portion of a mass storage medium (block zero) on which the volume header and a small "bootstrap" program used in booting the operating system reside. The boot area is reserved exclusively for use by HP–UX. On the Series 500, the portion of an SDF mass storage medium which contains an operating system.
boot ROM	A program residing in ROM (Read Only Memory) that executes each time the computer is powered–up. The function of the boot ROM is to run tests on the computer's hardware, find all devices accessible through the computer, and then load either a specified operating system or the first operating system found according to a specific search algorithm.
bus address	A number which makes up part of the address HP–UX uses to "find" a particular device. The *bus address* is determined by a switch setting on a peripheral device which allows the computer to distinguish between two devices connected to the same interface. A bus address is sometimes called a "device address".
CS/80 or *CS–80*	A family of mass storage devices that communicate via a common protocol, *CS/80* (Command Set '80) command set. This family includes hard discs, removable discs, and tape devices.
character special file	A special file associated with devices which transfer data by a means other than by using the buffer cache. Examples are printers, terminals, nine–track magnetic tapes, and discs accessed in "raw" mode (see *raw disc*).
child process	A new process created by an existing process via the *fork*(2) or *vfork*(2) system call. The new process is thereafter known to the existing process as its *child process.* The existing process is the *parent process* of the new process. See *parent process* and *fork.*
command	A stand–alone unit of executable code (a program), or a file containing a list of other programs to execute in order (a shell script). In HP–UX, commands are executed through a command interpreter called a shell, often *sh*(1) or *csh*(1). Arguments following the command name are passed on to the command program. You can write your own commands, either as executable programs, or as shell scripts (written in the shell programming language).
command interpreter	A program which reads lines of text from standard input (typed at the keyboard or redirected from a file), and interprets them as requests to execute other programs. A command interpreter for

HP-UX is called a shell. See *sh*(1) and *csh*(1).

configuration — The ability to "customize" your kernel with the drivers, code, and tunable parameter values desired.

control character — A member of a character set which produces action in a device other than printing or displaying a character. In the ASCII character set, control characters are those in the range 0 through 31, and 127. Control characters can be generated by holding down [CTRL], [CONTROL], or [CNTL] (depending on what the control key is labeled on your keyboard) and pressing a character key. These two-key sequences are often written as ctrl-d, for example, or ^D, where ^ stands for the control key. Both representations assume that the control key is held down while the second key is pressed.

crash — The unexpected shutdown of a program or system. If the operating system crashes, this is a "system crash", and requires the system to be re-booted.

current directory — See *working directory.*

current working directory — See *working directory.*

daemon — A process which runs in the background, and which is usually immune to termination instructions from a terminal. Its purpose is to perform various scheduling, clean-up, and maintenance jobs. *Lpsched*(1) is an example of a daemon that exists to perform these functions for line printer jobs queued by *lp*(1). An example of a permanent daemon (i.e. it never should die) is *cron*(1m).

data encryption — A method for encoding information in order to protect sensitive or proprietary data. For example, all users' passwords are automatically encrypted by HP-UX. The encryption method used by HP-UX converts ASCII text into a base-64 representation using the alphabet ., /, 0-9, A-Z, a-z. See *passwd*(5) for the numerical equivalents associated with this alphabet.

default search path — The sequence of directory prefixes that *sh*(1), *time*(1), and other HP-UX commands apply in searching for a file known by an incomplete path name (i.e. a path name not beginning with a slash, /). It is defined by the environment variable **PATH** (see *environ*(7)). *Login*(1) sets **PATH** equal to **:/bin:/usr/bin**, which means that your working directory is the first directory searched, followed by **/bin**, followed by **/usr/bin**. You can redefine the search path by modifying the value of **PATH.** This is usually done in **/etc/profile**, and/or in the **.profile** file found in your home directory (for the Bourne shell), or *csh.login*, *.login*, or *cshrc* (for the C-shell *csh).*

delta — A term used in the Source Code Control System (SCCS) to describe a unit of one or more textual changes to an SCCS file. Each time you edit an SCCS file, the changes you make to the file are stored separately as a delta. Then, using the *get*(1) command, you can specify which deltas are to be applied to or excluded from the SCCS file, thus yielding a particular version of the file. (Contrast this with the *vi* or *ed* editor, which incorporates your changes into the file immediately, prohibiting you from obtaining a previous version of that file.) See *SCCS*, *SCCS*

	file.
demon	See *daemon.*
device file	See *special file.*
directory	A file which provides the mapping between the names of files and their contents. For every file name contained in a directory, that directory contains a pointer to the file's *i-node* called a *link* . A file may have several links appearing anywhere on the same file system. Each user is free to create (using *mkdir*(1)) as many directories as he needs, providing that the parent directory of the new directory gives him permission to do so. Once a directory has been created, it is ready to contain ordinary files and other directories. An HP–UX directory is named and behaves exactly like an ordinary file, with one exception: no user (including the super–user) is allowed to write data on the directory itself; this privilege is reserved for the HP–UX operating system. By convention, a directory contains at least two links, **.** and **..**, referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot–dot refers to its parent directory. For purposes of deletion, a directory containing only **.** and **..** is considered empty. (In the root directory, "/", ".", and ".." are identical.)
effective group ID	Every process has an effective group ID that is used to determine file access permissions. A process's effective group ID is determined by the file (command) that process is executing. If that file's set–group–ID bit is set (located in the mode of the file – see *mode*), then the process's effective group ID is set equal to the file's group ID. This makes the process appear to belong to the file's group, perhaps enabling the process to access files which must be accessed in order for the program to execute successfully. If the file's set–group–ID bit is not set, then the process's effective group ID can only be set by an explicit call to *getuid*(2) or *setuid*(2). The id is always inherited from parent across a *fork*(2). The *setuid/getuid* bit determines whether it is inhereted across *exec*(2). See *group*, *real group ID*, and *set-group-ID bit.*
effective user ID	A process has an effective user ID that is used to determine file access permissions (and other permissions with respect to system calls, if the effective user ID is 0 – that of the super–user). A process's effective user ID is determined by the file (command) that process is executing. If that file's set–user–ID bit is set (located in the mode of the file – see *mode*), then the process's effective user ID is set equal to the file's user ID. This makes the process appear to be the file's owner, enabling the process to access files which must be accessed in order for the program to execute successfully. (Many HP–UX commands which are owned by *root*, such as *mail*(1), have their set–user–ID bit set so other users can execute these commands.) If the file's set–group–ID bit is not set, then the process's effective group ID can only be set by an explicit call to *getuid*(2) or *setuid*(2). The id is always inherited from parent across a *fork*(2). The *setuid/getuid* bit determines whether it is inhereted across *exec*(2). See *real user ID* and *set-user-ID bit.*

environment	The set of defined shell variables (some of which are PATH, TERM, SHELL, EXINIT, HOME, etc.) which define the conditions under which your commands run. These conditions can include your terminal characteristics, your home directory, and your default search path. Each shell variable setting in the current process is passed on to all child processes that are created, provided that each shell variable setting has been exported via the *export* command (see *sh*(1)) or *setenv*(1) with *csh*(1). Unexported shell variable settings are meaningful only to the current process, and any child processes created are given the default settings given certain shell variables in */etc/profile* and/or *$HOME/.profile* (when using the Bourne shell) or */etc/csh.login*, *.login*, or *cshrc* (when using the C–shell).
end–of–file	(1) the data returned when attempting to read past the logical end of a file via *stdio*(3S) routines. In this case end–of–file is not properly a character. (2) The character [CTRL]–[D]. (3) A character defined by *stty*(1) or *ioctl(2)* (see *termio*(4)). to act as end–of–file on your terminal. Usually this is [CTRL]–[D]. (4) The indication (as the function result) which indicates end of data when using *read*(2).
file	An HP–UX file is simply a group of logically related bytes of information. These bytes, for example, could be a bytes of executable code or bytes of data. Thus, directories, ordinary files, special files, etc. can all be considered files. Every file must have a file name (see *file name*) which enables the user (and many of the HP–UX commands) to reference the contents of the file. The size of a file is exactly the number of bytes the file contains - the system imposes no particular structure on the contents of a file (although some programs do). Files may be accessed serially or randomly (indexed by byte offset). The interpretation of file contents and structure is up to the programs that access the file.
file access permissions	Every file in the file system has a set of access per– missions. These permissions are used in determining whether a process may perform a requested operation on the file (such as opening a file for writing). Access permissions are established at the time a file is created. They may be changed at some later time through the chmod(2) call.

File access is broken down according to whether a file may be: read, written, or executed. Directory files use the execute permission to control if the directory may be searched.

File access permissions are interpreted by the system as they apply to three different classes of users: the owner of the file, those users in the file's group, anyone else. Every file has an independent set of access permissions for each of these classes. When an access check is made, the system decides if permission should be granted by checking the access information applicable to the caller.

Read, write, and execute/search permissions on a file are granted to a process if:

The process's effective user ID is super–user.

The process's effective user ID matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

The process's effective user ID does not match the user ID of the owner of the file, and either the process's effective group ID matches the group ID of the file, or the group ID of the file is in the process's group access list, and the appropriate access bit of the "group" portion (070) of the file mode is set.

The process's effective user ID does not match the user ID of the owner of the file, and the process's effective group ID does not match the group ID of the file, and the appropriate access bit of the "other" portion (07) of the file mode is set.

Otherwise, the corresponding permissions are denied.

file descriptor

A small integer identifier, which is used to refer to a file that has been opened for reading and/or writing, and is an index into the user's table of open files. The opened file must be identified by its file descriptor when using system calls to read or write the file.

The value of a file descriptor has a range from 0 to a system defined maximum. For systems at HP-UX STANDARD and above, the minimum value for this number is 60. For systems below HP-UX STANDARD the minimum value is 20. No file descriptor may have a value outside the range 0–59 or 0–19, depending on the implementation.

A file descriptor is obtained through system calls such as *open*(2), *creat*(2), *dup*(2), *fcntl*(2) or *pipe*(2). The file descriptor is used as an argument by calls such as *read*(2), *write*(2), *ioctl*(2), and *close*(2).

file name

A string of up to 14 characters which is used to refer to the contents of an ordinary file, special file, or directory. These characters may be any ASCII character except ASCII values 0 (null) and 47 (slash – /). Note that it is generally unwise to use *, ?, [, !, or] as part of file names because of the special meaning the shell attaches to these characters (see *sh*(1)). It is also not wise to begin a file name with -, +, or =, because some programs assume that these characters indicate that a command argument follows. Although permitted, it is advisable to avoid the use of characters which do not have a printable graphic on the hardware you commonly use, or which are likely to confuse the hardware.

file pointer

A data element, obtained through any of the *fopen*(3S) standard I/O library routines, which "points to" (refers to) a file opened for reading and/or writing, and which keeps track of where the next I/O operation will take place in the file (in the form of a byte offset relative to the beginning of the file). After obtaining the file pointer, it must thereafter be used to refer to the open file when using any of the standard I/O library routines. (See *stdio*(3S) for a list of these routines.)

file system	The supporting data structures, HP–UX directory structure, and associated files that reside on one or more mass storage volumes. Refer to the *System Administrator Manual* supplied with your system for details concerning file system implementation and maintenance.
filter	A command which reads data from the standard input, performs a transformation on the data, and writes it to the standard output.
fork	An HP–UX system call (*fork*(2)) which, when invoked by an existing process, causes a new process to be created. The new process is called the *child process*; the existing process is called the *parent process*. The child process is created by making an exact copy of the parent process. The parent and child processes are able to identify themselves by the value returned by their corresponding *fork* call (see *fork*(2) for details).
group	A *group* is a set of 0 or more users who are usually logically related in some way (e.g., all users who are working on a particular project) and who generally require the sharing of data between each other. The members of a group are defined in the file */etc/passwd* via a numerical *group ID* (users with identical group IDs are members of the same group). An ASCII *group name* is associated with each group ID in the file */etc/group* (the members of each group can be listed in /etc/group, also, but this information is purely for user benefit, and is of little use to the system). A group ID is associated with every file in the file system, and the mode of each file contains a set of permission bits which apply only to groups of which the file owner is a member. Thus, **if** you are a member of the group associated with the file, and **if** the appropriate permissions are given to your group in the file's mode, you may access the file. See *real group ID*, *effective group ID*, *access*groups, *privileged*group, and *set-group-ID bit.*
group access list	The group access list is an additional set of group ID's used only in determining resource accessibility. Access checks are performed as described in file access permissions.
hierarchical directory	A directory (or file system) structure in which each directory may contain other directories as well as files.
home directory	The directory name given by the value of the shell variable HOME. When you first log in, *login*(1) automatically sets HOME equal to your login directory (see *login directory*). You may change its value at any time, however. This is usually done in the *.profile* file contained in your login directory. Setting HOME in no way affects your login directory, but simply gives you a convenient way of referring to what should be your most commonly–used directory.
host name	An ASCII string of at most 8 characters (of which only 6 are supported by all the various manufactuer's UNIX systems) which uniquely identifies an HP–UX system on a *uucp* network. The host name for your system may be viewed and/or set with the *hostname*(1) command. Systems without a defined host name are described as "unknown" on the *uucp* network. Do not confuse a host name with a *node name*, which is a string that

	uniquely identifies an HP–UX system on a Local Area Network (LAN). Although your host and node names may be identical (and this is often advisable), they are set and used by totally different software. See *node name.*
i–node	Each ordinary or special file, or directory has associated with it an i–node. The i–node contains, among other things, the file's size, protection mask, the number of links, and pointers to the disc blocks where the file's contents can be found. Each connection between an i–node and its entry in one or more directories is called a link.
image	The current state of your computer (or your portion of the computer, on a multi–user system) during the execution of a command. Often thought of as a "snapshot" of the state of the machine at any particular moment during execution.
init	A special process (the initialization process) usually with a process ID of 1. It is the ancestor of every other process in the system and is used to start login processes.
interleave factor	A number which determines the order in which sectors on a mass storage medium are accessed. It can be optimized to make data acquisition more efficient.
Internal Terminal Emulator (*ITE*)	The "device driver" code contained in the HP–UX kernel and associated with the computer's built–in keyboard and display or a particular keyboard and display connected to the computer, depending on the Series and Model of your HP–UX computer. See *system console* and the *System Adminstrator Manual* supplied with your system for details.
interrupt signal	The signal sent by *SIGINT* (see *signal*(2)). This signal generally terminates whatever program you are running. The key which sends this signal can be redefined with *ioctl*(2) or *stty*(1) (see *termio*(4)). It defaults to the ASCII DEL (rubout) character (the [DEL] key) or the [BREAK] key. [CONTROL]–[C] is often used instead.
intrinsic	See *system call.*
I/O redirection	A mechanism provided by the HP–UX shell for changing the source of data for standard input and/or the destination of data for standard output and standard error. See *sh*(1).
kernel	The HP–UX operating system. The kernel is the executable code responsible for managing the computer's resources, such as allocating memory, creating processes, and scheduling programs for execution.
library	An archive file containing a set of subroutines and variables which may be accessed by user programs. For example, */lib/libc.a* is a library containing all functions of section 2, and all functions of section 3 marked (3C) and (3S), in the HP–UX Reference. Similarly, */lib/libm.a* is a library containing all functions in section 3 marked (3M) in the HP–UX Reference. See *intro*(3).

LIF	An acronym for Logical Interchange Format. A standard format for mass storage implemented on many Hewlett-Packard computers to aid in media transportability. The *lif**(1) commands are used to perform various functions using LIF.
link	A directory entry for any type of file. The information constituting a link includes the name of the file, and where the contents of that file may be found on a mass storage medium. One physical file may have several links to it. If the links appear in different directories, the file may or may not have the same name in each. If the links appear in one directory, however, each link must have a unique name in that directory. Multiple links to directories are not allowed (except for the super-user). See *cp*(1), *link*(1), *link*(2), and *unlink*(2). Also, to prepare a program for execution, see *linker.*
linker	The linker combines one or more object programs into one program, searches libraries to resolve user program references, and builds an executable file in *a.out* format. This executable file is ready to be executed through the program loader, *exec*(2). The linker is invoked with the *ld*(1) command. The linker is often called a *link editor.*
logical block size	The smallest unit of memory which can be allocated on a Series 500 SDF volume; a multiple of the physical sector size. This value is set at system initialization time; see *sdfinit*(1M).
login	The process of gaining access to HP-UX. This consists of successful execution of the login sequence defined by *login*(1) which varies depending on the system configuration. It includes providing a login name and possibly one or more passwords.
login directory	The directory in which you are placed immediately after you log in. This directory is defined for each user in the file */etc/passwd.* The shell variable HOME is set automatically to your login directory by *login*(1) immediately after you log in. See *home directory.*
magic number	The first word of an *a.out*(5) or archive file. This word contains the system ID, which tells what machine (hardware) the file will run on, and the file type (executable, shareable executable, archive, etc.).
major number	A number used exclusively to create special files that enable I/O to/from specific devices. This number indicates which device driver to use for the device. Refer to *mknod*(1M) and the *System Administrator Manual* supplied with your system for details.
message queue identifier	A message queue identifier (msqid) is a unique positive integer created by a *msgget*(2) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct   ipc_perm msg_perm; /* operation permission struct */
ushort   msg_qnum;          /* number of msgs on q */
ushort   msg_qbytes;        /* max number of bytes on q */
ushort   msg_lspid;         /* pid of last msgsnd operation */
ushort   msg_lrpid;         /* pid of last msgrcv operation */
```

```
time_t  msg_stime;          /* last msgsnd time */
time_t  msg_rtime;          /* last msgrcv time */
time_t  msg_ctime;          /* last change time */
                            /* Times measured in secs since */
                            /* 00:00:00 GMT, Jan. 1, 1970 */
```

Msg_perm is a ipc_perm structure that specifies the message operation permission (see below). This structure includes the following members:

```
ushort  cuid;       /* creator user id */
ushort  cgid;       /* creator group id */
ushort  uid;        /* user id */
ushort  gid;        /* group id */
ushort  mode;       /* r/w permission */
```

Msg_qnum is the number of messages currently on the queue. **Msg_qbytes** is the maximum number of bytes allowed on the queue. **Msg_lspid** is the process id of the last process that performed a *msgsnd* operation. **Msg_lrpid** is the process id of the last process that performed a *msgrcv* operation. **Msg_stime** is the time of the last *msgsnd* operation, **msg_rtime** is the time of the last *msgrcv* operation, and **msg_ctime** is the time of the last *msgctl*(2) operation that changed a member of the above structure.

message operation permissions In the *msgop*(2) and *msgctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

Read and Write permissions on a msqid are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches **msg_perm.[c]uid** in the data structure associated with *msqid* and the appropriate bit of the "user" portion (0600) of **msg_perm.mode** is set.

The process's effective user ID does not match **msg_perm.[c]uid** and the process's effective group ID matches **msg_perm.[c]gid** and the appropriate bit of the "group" portion (060) of **msg_perm.mode** is set.

The process's effective user ID does not match **msg_perm.[c]uid** and the process's effective group ID does not match **msg_perm.[c]gid** and the appropriate bit of the "other" portion (06) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

metacharacter A character which has special meaning to the HP-UX shell. The set of metacharacters includes: *, ?, !, [,], <, >, ;, |, ´, `, ", and &. Refer to *sh*(1) for the meaning associated with each.

minor number	A number used exclusively to create special files that enable I/O to/from specific devices. This number is passed to the device driver and is used to select which device in a family of devices is to be used, and possibly some operational modes. The exact format and meaning of the minor number is both system and driver dependent. Refer to the *System Administrator Manual* supplied with your system for details. See *address.* On the Series 200 and 500, for HP–IB devices, this number indicates the HP–IB address, select code, and the unit and/or volume numbers.
mode	A 16–bit word associated with every file in the file system, stored in the *i–node.* The least–significant 12 bits of this word determine the read, write, and execute permissions for the file owner, file group, and all others, and contain the set–user–ID, set–group–ID, and "sticky" (save text image after execution) bits. The least–significant 12 bits are settable by the *chmod*(1) command if you are the file's owner or the super–user. The sticky bit can only be set by the super–user. These 12 bits are sometimes referred to as *permission bits.* The most–significant 4 bits specify the file type for the associated file and are set as the result of *creat*(2), *open*(2), or *mknod*(2) system calls.
mountable file system	A (blocked special) file system contained on some mass storage medium with its own root directory and an independent hierarchy of directories and files. See *block special file* and *mount*(1).
multi–user state	The condition of the HP–UX operating system in which terminals in addition to the system console are allowing communication between the system and its users. By default, the Series 200 multi–user state is state 2, and the Series 500 multi–user state is state 1. Do not confuse the multi–user system with the multi–user state. A multi–user system is a system which may have more than one user actively communicating with the system when it is in the multi–user state. Multi–user states—there can be more than one multi–user state in *inittab*— remove the single–user restriction imposed by the single–user state. See *single–user state.* See *inittab*(5).
new–line	The character with an ASCII value of 10 (line–feed) used to separate lines of characters. It is represented by \n in the C language and in various utilities. The terminal driver (see *tty*(4)) normally interprets the carriage–return/line–feed sequence sent by a terminal as a single new–line character.
node name	A string of up to 31 characters, not including control characters or spaces, that uniquely identifies a node on a Local Area Network (LAN). The node name for each system is set by the *npowerup* command, which is one of the commands supplied with the optional LAN/9000 product. Do not confuse a node name with a *host name*, which is a string that uniquely identifies an HP–UX system on a *uucp* network. Your node and host names can be identical, but they are used and set by totally different software. See *host name*, *LAN/9000 User's Guide*, and *LAN/9000 Node Manager's Guide.*

ordinary file	A type of HP–UX file containing ASCII text (e.g. program source), binary data (e.g. executable code), etc. Ordinary files can be created by the user through I/O redirection, editors, or HP–UX commands.
orphan process	Whenever a parent process terminates for any reason and leaves behind one or more child processes that are still active, those child processes are called *orphan processes. Init*(1M) inherits (becomes the effective parent of) all orphan processes.
OSF	An acronym for Operating System File. An OSF resides in the SDF boot area on a Series 500 system, and contains all or part of an operating system. See *osmgr*(1M), *oscp*(1M), *osck*(1M), and *osmark*(1M).
owner	The owner of a file is usually the creator of that file. However, the ownership of a file can be changed by the super–user or the current owner with the *chown*(1) command or the *chown*(2) system call. The file owner is able to do whatever he wants with his files, including remove them, copy them, move them, change their contents, etc. He is also able to change the files' modes.
parent directory	A directory's parent directory is the directory one level above it in the file hierarchy. All directories except the root directory (/) have one (and only one) parent directory. The parent directory is sometimes referred to as the *superior directory.*
parent process	Whenever a new process is created by a currently–existing process (via *fork*(2) or *vfork*(2)), the currently–existing process is said to be the parent process of the newly–created process. Every process has exactly one parent process (except the init process - see *init*), but each process can create several new processes with the *fork*(2) system call. The parent process ID of any process is the process ID of its creator.
password	A string of ASCII characters used to verify the identity of a user. Passwords can be associated with users and groups. If a user has a password, it is automatically encrypted and entered in the second field of that user's line in the */etc/passwd* file. A user may create or change a password for himself with the *passwd*(1) command.
path name	(sometimes written as one word, *pathname*). A sequence of directory names separated by slashes, and ending with any file name. All file names except the last in the sequence *must* be directories. If a path name begins with a slash (/), it is an *absolute* path name (see *absolute path name*); otherwise it is a *relative* path name (see *relative path name*). A path name defines the path to be followed through the hierarchical file system in order to find a particular file. More precisely, a path name is a null–terminated character string constructed as follows:

```
<path-name>::=<file-name>|<path-prefix><file-name>|/
<path-prefix>::=<rtprefix>|/<rtprefix>
<rtprefix>::=<dirname>/|<rtprefix><dirname>/
```

where <file-name> is a string of 1 to 14 characters other than the ASCII slash and null, and <dirname> is a string of 1 to 14 characters (other than the ASCII slash and null) that names a directory or network special file with RFA (Remote File Access).

A slash by itself names the root directory.

Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.

permission bits	The nine least-significant bits of a file's mode. These bits determine read, write, and execute permissions for the file's owner, the file's group, and all others. See *chmod*(2) for further details.
pipe	An inter-process I/O channel used to pass data between two processes. It is commonly used by the shell to transfer data from the standard output of one process to the standard input of another. On a command line, a pipe is signaled by a vertical bar (|). The output from the command(s) on the left of the vertical bar is channeled directly into the standard input of the command(s) on the right. The *pipe*(2) intrinsic function allows user programs to take advantage of this feature.
privileged groups	A privileged group is a group which has had a *setprivgrp* (see *getprivgrp*(2)) operation performed on it giving it access to some system calls otherwise reserved for the super-user.
proc1	See *init.*
process	An invocation of a program, or the execution of an image. No command can be executed without a process in which it can execute. Alternately, a process cannot exist without a command or image in some stage of execution. Several processes can all be running the same program, but each may have different data and be in different stages of execution.
process group	An association of one or more processes is called a process group. A process's membership in a particular process group is established by a numerical process group ID. Each process can belong to only one process group. Every process group has a process group leader. See *process group ID* and *process group leader.*
process group ID	A positive integer in the range 1 - 30000 associated with every active process, which establishes that process's membership with a particular process group. All members of a process group have the same process group ID. A process group ID is always the process ID of the process group leader. This grouping permits the signalling of related processes. See *kill*(2), *process group*, and *process group leader.*
process group leader	A process group leader is a process whose process group ID and process ID are equal. A process becomes a process group leader through the *setpgrp*(2) system call. All processes created by the process group leader become members of that process group. All processes created by the *init* process (see *init*) are process group leaders. For example, when you log in on the system, the shell you receive to interpret your commands is a process group leader, and all subsequent process's created by your shell are members of

your shell's process group. See *process group ID* and *process group.*

process ID

Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 1 to 30 000. This permits the selective sending of signals to processes with *kill*(1), *kill*(2), or *signal*(2). The process ID of any user process is available with the *ps*(1) command. If a background process is created, the shell reports its process ID to you when execution has begun.

program

A sequence of instructions, either binary (as machine object code) or text (as source code or shell scripts), that define an algorithm that can be carried out by a computer. C, FORTRAN, Pascal, and BASIC source; awk source; shell scripts; and executable object (*a.out*(5)) are all examples of programs.

prompt

The character(s) displayed by the shell on the display indicating that the system is ready for a command. The prompt is usually a dollar sign ($) for ordinary users and a pound sign (#) for the super–user, but the user can re–define it to be any string by setting the shell variable **PS1** in his *.profile* file.

quit signal

The signal sent by *SIGQUIT.* See *signal*(2). The quit signal is generated by typing the character defined by the teletype handler as your quit signal. (See *stty*(1), *ioctl*(2), and *termio*(4).) The default is the ASCII FS character (ASCII value 28, generated by typing [CONTROL]–[\].) This signal usually causes a running program to terminate and generates a file containing the "core image" of the terminated process. The core image is useful for debugging purposes. (Some systems do not support core images, and on those systems no such file is generated.)

raw disc

The name given to a disc for which there exists a character special file which allows direct transmission between the disc and the user's read or write buffer.

real group ID

A positive integer which is assigned to every user on the system. The association of a user and his real group ID is done in the file */etc/passwd.* The modifier "real" is used because a user can also have an *effective* group ID (see *effective group ID*). The real group ID can then be mapped to a group name in the file */etc/group*, although it need not be. Thus, every user is a member of some group (which may be nameless), even if that group has only one member.

Every time a process creates a child process (via *fork*(2)), that process has a real group ID equal to the parent process's real group ID.

real user ID

A positive integer which is assigned to every user on the system. A real user ID is assigned to every valid login name in the file */etc/passwd.* The modifier "real" is used because a user can also have an *effective* user ID (see *effective user ID*).

Every time a process creates a child process (via *fork*(2)), that process has a real user ID equal to the parent process's real user ID.

regular expression	A string of zero or more characters; the characters contained in the string may all be literal, which means that the regular expression matches itself only, or one or more of the characters may be a metacharacter, which means that a single regular expression could match several literal strings. Regular expressions are most often encountered in text editors (*ed*(1), *ex*(1), *vi*(1)), where searches are performed for a specific piece of text, or in commands that were created to search for a particular string in a file (most notably *grep*(1)). *Sh*(1) and *csh*(1) also use metacharacters to match one or more patterns; this is a different mechanism that regular expressions. See *ed*(1).
relative path name	A path name that does not begin with a slash. It indicates that a file's location is given relative to your current working directory, and that the search begins there (instead of at the root directory). An example is **dir1/file2**, which searches for the directory **dir1** in your current working directory. **Dir1** is then searched for the file **file2**.
root directory	1) The highest level directory of the hierarchical file system, from which all other files branch. In HP-UX, the "/" character refers to the root directory. The root directory is the only directory in the file system which is its own parent directory. 2) Each process has associated with it a concept of a root directory for the purpose of resolving path name searches for those paths beginning with "/". A process's root directory need not be the root directory of the root file system, and can be changed by the *chroot*(1) command or *chroot*(2) system call. Such a directory appears to the process involved to have **..** pointing to itself.
root volume	The mass storage volume which contains the boot area (which contains the HP-UX kernel) and the root directory of the HP-UX file system.
saved user ID	Every process has a saved user ID which retains the process's effective user ID from the last successful *exec*(2), or from the last super-user call to *setuid*(2). *Setuid*(2) permits a process to set its effective user ID to this remembered value. Consequently, a process which executes a program with the set-user-ID bit set and with an owner ID of 5 (for example) can set its effective user ID to 5, or to its real user ID, any time until the program terminates. See *exec*(2), *setuid*(2), *saved group ID*, *effective user ID*, and *set-user-ID bit.*
saved group ID	Every process has a saved group ID which retains the process's effective group ID from the last successful *exec*(2), or from the last super-user call to *setgid. Setgid* permits a process to set its effective group ID to this remembered value. Consequently, a process which executes a program with the set-group-ID bit set and with a group ID of 5 (for example) can set its effective group ID to 5, or to its real group ID, any time until the program terminates. See *exec*(2), *setuid*(2), *saved user ID*, *effective group ID*, and *set-group-ID bit.*
SCCS	An acronym for Source Code Control System. The Source Code Control System is a set of HP-UX commands which enable you to store changes to an SCCS file as separate "units" (called

	deltas). These units, each of which contains one or more textual changes to the file, can then be applied to or excluded from the SCCS file to obtain different versions of the file. The commands that make up SCCS are *admin*(1), *cdc*(1), *delta*(1), *get*(1), *prs*(1), *rmdel*(1), *sact*(1), *sccsdiff*(1), *unget*(1), *val*(1), and *what*(1). See *delta*, *SCCS file.*
SCCS file	An ordinary text file which has been modified so that the Source Code Control System (SCCS) may be used with it. This modification is done automatically by the *admin*(1) command. See *SCCS*, *delta.*
SDF	An acronym for Structured Directory Format. SDF is implemented on the Series 500 computers only, and provides tree-structured access to files through the root directory of the volume.
secondary prompt	One or more characters that the Bourne shell *sh*(1) prints on the display, indicating that more input is needed. This prompt is much less often encountered than the shell's primary prompt (see *prompt*). When it occurs, it is usually caused by an omitted right quote on a string (which confuses the shell), or when you enter a shell programming language control-flow construct (such as a **for** construct) from the command line. By default, the shell's secondary prompt is the greater-than sign (>), but you can re-define it by setting the shell variable **PS2** appropriately in your *.profile* file.
select code	On the series 200 and 500 part of an address used for devices. A number determined by a setting on the interface card to which a peripheral device is connected, or by the particular I/O slot the I/O card resides in. Multiple peripherals connected to the same interface card share the same select code.
semaphore identifier	A semaphore identifier (semid) is a unique positive integer created by a *semget*(2) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct  ipc_perm sem_perm; /* operation permission struct */
ushort  sem_nsems;         /* number of sems in set */
time_t  sem_otime;         /* last operation time */
time_t  sem_ctime;         /* last change time */
                           /* Times measured in secs since */
                           /* 00:00:00 GMT, Jan. 1, 1970 */
```

Sem_perm is a ipc_perm structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
ushort  cuid;       /* creator user id */
ushort  cgid;       /* creator group id */
ushort  uid;        /* user id */
ushort  gid;        /* group id */
ushort  mode;       /* r/a permission */
```

The value of **sem_nsems** is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. Sem_num values run sequentially from 0 to the value of sem_nsems minus 1. **Sem_otime** is the time of the last *semop*(2) operation, and **sem_ctime** is the time of the last *semctl*(2) operation that changed a member of the above structure.

A semaphore is a data structure that contains the following members:

```
ushort   semval;      /* semaphore value */
short    sempid;      /* pid of last operation  */
ushort   semncnt;     /* # awaiting semval > cval */
ushort   semzcnt;     /* # awaiting semval = 0 */
```

Semval is a non-negative integer. **Sempid** is equal to the process ID of the last process that performed a semaphore operation on this semaphore. **Semncnt** is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value. **Semzcnt** is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

semaphore operation permissions

In the *semop*(2) and *semctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Alter by user
00060	Read, Alter by group
00006	Read, Alter by others

Read and Alter permissions on a semid are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches **sem_perm.[c]uid** in the data structure associated with *semid* and the appropriate bit of the "user" portion (0600) of **sem_perm.mode** is set.

The process's effective user ID does not match **sem_perm.[c]uid**, and either the process's effective group ID matches **sem_perm.[c]gid** or **sem_perm.[c]gid** is in the process's group access list, and the appropriate bit of the "group" portion (060) of **sem_perm.mode** is set.

The process's effective user ID does not match **sem_perm.[c]uid**, and the process's effective group ID does not match **sem_perm.[c]gid** and neither is **sem_perm.[c]gid** in the process's group access list, and the appropriate bit of the "other" portion (06) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

set-group-ID bit

A single bit in the mode of every file in the file system. If a file is executed whose set-group-ID bit is set, then the effective group ID of the process which executed the file is set equal to the real group ID of the owner of the file. See *effective group ID*, *group*, and *real group ID*.

set-user-ID bit
A single bit in the mode of every file in the file system. If a file is executed whose set-user-ID bit is set, then the effective user ID of the process which executed the file is set equal to the real user ID of the owner of the file. See *effective user ID* and *real user ID*.

shared memory identifier
A shared memory identifier (shmid) is a unique positive integer created by a *shmget*(2) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. The data structure is referred to as *shmid_ds*; some of its members are:

```
struct  ipc_perm shm_perm; /* operation permission struct */
int     shm_segsz;         /* size of segment */
ushort  shm_cpid;          /* creator pid */
ushort  shm_lpid;          /* pid of last operation */
short   shm_nattch;        /* number of current attaches */
time_t  shm_atime;         /* last attach time */
time_t  shm_dtime;         /* last detach time */
time_t  shm_ctime;         /* last change time */
                           /* Times measured in secs since */
                           /* 00:00:00 GMT, Jan. 1, 1970 */
```

Shm_perm is a ipc_perm structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
ushort  cuid;       /* creator user id */
ushort  cgid;       /* creator group id */
ushort  uid;        /* user id */
ushort  gid;        /* group id */
ushort  mode;       /* r/w permission */
```

Shm_segsz specifies the size (in bytes) of the shared memory segment. **Shm_cpid** is the process id of the process that created the shared memory identifier. **Shm_lpid** is the process id of the last process that performed a *shmop*(2) operation. **Shm_nattch** is the number of times the segments are currently attaching to other processes. **Shm_atime** is the time of the last *shmat* operation, **shm_dtime** is the time of the last *shmdt* operation, and **shm_ctime** is the time of the last *shmctl*(2) operation that changed one of the members of the above structure.

shared memory operation permissions
In the *shmop*(2) and *shmctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed and is interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

Read and Write permissions on a shmid are granted to a process if one or more of the following are true:

> The process's effective user ID is super–user.
>
> The process's effective user ID matches **shm_perm.[c]uid** in the data structure associated with *shmid* and the appropriate bit of the "user" portion (0600) of **shm_perm.mode** is set.
>
> The process's effective user ID does not match **sem_perm.[c]uid**, and either the process's effective group ID matches **sem_perm.[c]gid** or **sem_perm.[c]gid** is in the process's group access list, and the appropriate bit of the "group" portion (060) of **sem_perm.mode** is set.
>
> The process's effective user ID does not match **sem_perm.[c]uid**, and the process's effective group ID does not match **sem_perm.[c]gid** and neither is **sem_perm.[c]gid** in the process's group access list, and the appropriate bit of the "other" portion (06) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

shell	A shell is a command which functions as both a command interpreter and an interpretive programming language. A shell is usually automatically invoked (via */etc/passwd)foreveryuser*who in order to provide a user–interface to the HP–UX operating system. See *sh*(1), *csh*(1), or *rsh*(1), and the tutorials supplied with your system for details.
shell program	See *shell script.*
shell script	A sequence of shell commands and shell programming language constructs stored in a file and invoked as a user command (program). No compilation is needed prior to execution, because the shell recognizes the commands and constructs that make up the shell programming language. A shell script is often called a *shell program* or a *command file.* See the shell programming article included in *HP–UX Concepts and Tutorials.*
signal	Signals are software interrupts sent to processes, informing them of special situations or events. They are frequently used to synchronize the operation of two or more processes. See *signal*(2) and *kill*(2).
single–user state	A condition of the HP–UX operating system in which the system console provides the only communication mechanism between the system and its user. By default, the Series 200 single–user state is state 1, and the Series 500 multi–user state is state 2. Do not confuse the single–user state, in which the software is limiting a multi–user system to a single–user communication, with a single–user system, which can never communicate with more than one fixed terminal. See *multi–user state.*
special file	Often called a *device file*, this is a file associated with an I/O device. Special files are read and written just like ordinary files, but requests to read or write result in activation of the associated device. Most standard special files reside in */dev*; however,

	network special files reside in */net*, and fifo special files can exist in any directory.
special processes	Processes with certain (small) process ID's are special. On a typical system, the ID's of 0, 1, and 2 are assigned as follows: Process 0 is the scheduler. Process 1 is the initialization process *init*, and is the ancestor of every other process in the system. It is used to control the process structure. On paging systems with virtual memory process 2 is the paging daemon.
	On the Series 500, there is no process 0 and the scheduler does not exist as an identifiable entity. The paging demon also does not exist as an identifiable entity.
standard error	The destination of error and special messages from a program. The standard error file is often called *stderr*, and is automatically opened for writing on file descriptor 2 for every command invoked. By default, the user's terminal is the destination of all data written to stderr, but it can be redirected elsewhere.
standard input	The source of input data for a program. The standard input file is often called *stdin*, and is automatically opened for reading on file descriptor 0 for every command invoked. By default, the user's terminal is the source of all data read from stdin, but it can be redirected from another source.
standard output	The destination of output data from a program. The standard output file is often called *stdout*, and is automatically opened for writing on file descriptor 1 for every command invoked. By default, the user's terminal is the destination of all data written to stdout, but it can be redirected elsewhere.
stream	A term most often used in conjunction with the standard I/O library routines documented in section 3 of this manual. A stream is simply a file pointer (declared as **FILE *stream**) returned by the *fopen*(3S) library routines. It may or may not have buffering associated with it (by default, buffering is assigned, but this may be modified with *setbuf*(3S)).
sticky bit	A single bit in the mode of every file in the file system. If set, then the data structure and heap storage for the text portion of the file is retained even if no process is currently attaching (using) to it. The objective is to reduce startup time of future processes that may use the same text file. Only the super-user can set the sticky bit. The sticky bit is read each time the file is executed (via *exec*(2)).
sub-directory	A directory that is one (or perhaps more) levels lower in the file system hierarchy than a given directory. Sometimes called a *subordinate directory*.
subordinate directory	See *sub-directory*.
super block	A block on each file system's mass storage medium which describes the file system. The contents of the super-block vary between implementations. Refer to the *System Administrator Manual* supplied with your system, and the appropriate *fs*(5) entry for details.

super–user	The HP–UX system administrator. This user has access to all files, and can perform privileged operations. He has a real and effective user ID of 0, and, by convention, the user name of *root.*
superior directory	See *parent directory.*
system call	An HP–UX operating system kernel function available to the user through a high–level language (such as FORTRAN, Pascal, or C). Also called an "intrinsic" or a "system intrinsic". The available system calls are documented in section 2 of the HP–UX Reference manual.
system console	A keyboard and display (or terminal) given a unique status by HP–UX and associated with the special file */dev/console.* All boot ROM or system loader error messages, HP–UX system error messages, and certain system status messages are sent to the system console. Under certain conditions (such as the single–user state), the system console provides the only mechanism for communicating with HP–UX. See *HP–UX Concepts and Tutorials* and the *System Administrator Manual* provided with your system for details on configuration and use of the system console.
terminal affiliation	The means by which a process group leader establishes an association between itself and a particular terminal. A terminal becomes affiliated with a process group leader (and subsequently all processes created by the process group leader - see *terminal group*) whenever the process group leader executes (either directly or indirectly) an *open*(2) or *creat*(2) system call for that a terminal. Then, **if** the process which is executing *open*(2) or *creat*(2) is a process group leader, and **if** that process group leader is not yet affiliated with a terminal, and **if** the terminal being opened is not yet affiliated with a process group, the affiliation is established. An affiliated terminal keeps track of its process group affiliation by storing the process group's process group ID in an internal structure. Two benefits are realized by terminal affiliation. First, all signals sent from the terminal are sent to all processes in the terminal group. Second, all processes in the terminal group can perform I/O from/to the generic terminal driver */dev/tty*, which automatically selects the affiliated terminal. Terminal affiliation is broken with a terminal group when the process group leader terminates, after which the hangup signal is sent to all processes remaining in the process group. Also, if a process (which is not a process group leader) in the terminal group becomes a process group leader via the *setpgrp*(2) system call, its terminal affiliation is broken. See *process group*, *process group leader*, *terminal group*, and *setpgrp*(2).
terminal group	A terminal group is a process group whose process group leader has established affiliation with a particular terminal. Once a process group leader has established affiliation with a terminal, all processes in that process group created **after** the affiliation

	are members of that terminal group. Processes existing before and during the time when affiliation is established do not inherit the affiliation, and are thus not part of the terminal group. A terminal group is sometimes called a *tty group.* This grouping is used to terminate a group of related process upon termination of one of the processes in the group; see *exit*(2) and *signal*(2). See *process group*, *process group leader*, *terminal affiliation*, and *setpgrp*(2).
tty group ID	See *terminal group*.
unit number	Part of an address used for devices. A number whose meaning is software- and device-dependent, but which is often used to specify a particular disc drive in a device with a multi-drive controller. See the *System Administrator Manual* supplied with your system for details.
volume number	Part of an address used for devices. A number whose meaning is software- and device-dependent, but which is often used to specify a particular volume on a multi-volume disc drive. See the *System Administrator Manual* supplied with your system for details.
working directory	Each process has associated with it the concept of a current working directory. For a shell, this appears as the directory in which you currently reside. This is the directory in which relative path name (i.e. when a given path name does not begin with "/") searches begin. It is sometimes referred to as the *current directory*, or the *current working directory.*
zombie process	The state of a process where the only system resource allocated to it is a slot in the process table data structure. This state is arrived at when the process is being terminated. It is a harmless occurrence which rectifies itself the next time that the parent process waits. The *ps*(1) command lists zombie processes as "<defunct>".

PERMUTED INDEX

initialize SDF volume	sdfinit(1M)
initialize terminal type and mode on login	tset(1)
inittab	inittab(5)
i-node, clear i-node by zeroing it out	clri(1M)
i-node, description of i-node format	inode(5)
i-node, enable access to i-node for file system repair	fsdb(1M)
i-nodes, create file name vs. i-node list	ncheck(1M)
i-nodes, find number of free i-nodes in mounted file system	ustat(2)
input and format data from buffered open file	scanf(3S)
input and format data from standard input	scanf(3S)
input and format data from string	scanf(3S)
input commands to shell	sh(1)
input control, description of input control for terminal	tty(4)
input/output between process and command	popen(3S)
input/output, description of buffered file	stdio(3S)
input/output operation, get current byte offset of	fseek(3S)
input/output operation, reposition next	fseek(3S)
input/output, output character/word to open file or standard output	putc(3S)
input/output, push character back into input stream	ungetc(3S)
input/output redirection	sh(1)
input/output, write string to open file or standard output	puts(3S)
insert calls to getmsg using findstring output	insertmsg(1)
install	install(1M)
install commands into file system	install(1M)
install object files in binary directories	cpset(1M)
integer, get largest integer smaller than x	floor(3M)
integer, get smallest integer larger than x	floor(3M)
integers, convert between 3-byte and long	l3tol(3C)
integer trap control	intrapoff(3M)
integrity check of operating system in SDF boot area(s)	osck(1M)
interactive IMAGE database access	query(1)
interactively write (talk) to another user	write(1)
interface, description of hpib	hpib(4)
interface to blocked/unblocked disc, description of	disc(4)
interface to terminal I/O, description of	tty(4)
interleave factor, establish for SDF volume	sdfinit(1M)
interprocess communication, create	pipe(2)
inter-process communication facilities status	ipcs(1)
inter-process communication routines	stdipc(3C)
interrupt character, description of	tty(4)
intrapoff	intrapoff(3M)
I/O between process and command	popen(3S)
I/O, description of buffered file	stdio(3S)
I/O operation, get current byte offset of	fseek(3S)
I/O operation, reposition next	fseek(3S)
I/O, output character/word to open file or standard output	putc(3S)
I/O, push character back into input stream	ungetc(3S)
I/O redirection	sh(1)
I/O: GPIO routines (device I/O library)	gpio_*(3I)
I/O: HP-IB routines (device I/O library)	hpib_*(3I)
I/O: I/O routines (device I/O library)	io_*(3I)
I/O, write string to open file or standard output	puts(3S)
ioctl	ioctl(2)
ioctl(2) system calls, description of	tty(4)
iomap	iomap(4)

MANUAL COMMENT CARD

HP-UX Reference

Manual Reorder No. 09000-90008

Name: ____________________

Company: ____________________

Address: ____________________

____

Phone No: ____________________

Please note the latest printing date from the Printing History (page ii) of this manual and any applicable update(s); so we know which material you are commenting on ____________________.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

HEWLETT
PACKARD

HP Part Number
09000-90008
Printed in U.S.A. 6/86

09000-90657
For Internal Use Only