68NW9209H48A

SYSTEM V/88 Release 3.2 Programmer's Reference Manual

(Part 2)

MOTOROLA

MOTOROL

Motorola welcomes your

Manual Title ____________

Part Number ____________

Your Name ____________

Your Title ____________

Company ____________

Address ____________

____

____

General Information:

Do you read this manu

☐ Install the product

☐ Reference inform

In general, how do you

☐ Index ☐ Table o

Completeness: ☐ Ex

What topic would you

____

____

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 2565 PHOENIX, ARIZONA

POSTAGE WILL BE PAID BY ADDRESSEE

MOTOROLA INC.
Computer Group, Microcomputer Division, DW164
2900 South Diablo Way
Tempe, AZ 85282-9741

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

Presentation: ☐ Excellent ☐ Very Good ☐ Good ☐ Fair ☐ Poor

What features of the manual are most useful (tables, figures, appendixes, index, etc.)?

Is the information easy to understand? ☐ Yes ☐ No If you checked no, please explain:

Is the information easy to find? ☐ Yes ☐ No If you checked no, please explain:

Technical Accuracy: ☐ Excellent ☐ Very Good ☐ Good ☐ Fair ☐ Poor

If you have found technical or typographical errors, please list them here.

Page Number	Description of Error

SYSTEM V/88 Release 3.2

Programmer's

Reference Manual

Part 2

(68NW9209H48A)

PREFACE

The *SYSTEM V/88 Release 3.2 Programmer's Reference Manual, Part 2* is fo programmers and technical personnel who have a general familiarity with th SYSTEM V/88 operating system.

SYSTEM V/88 Release 3.2 is based on the AT&T UNIX System V Release 3.2.
YSTEM V/68 Release 3 is based on the AT&T UNIX System V Release 3.

First Edition February 1990

CONTENTS (Part 2)

3. Subroutines (Cont'd)

4. File Formats

5. Miscellaneous Facilities

NAME

getpeername - gets name of connected peer

SYNOPSIS

getpeername (*s*, *name*, *namelen*)
int s;
struct *sockaddr *name;*
int **namelenf;*

DESCRIPTION

getpeername returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return, it contains the actual size of the name returned (in bytes).

DIAGNOSTICS

A 0 is returned if the call succeeds; -1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

bind(2), socket(2), getsockname(2)

NAME

nlsgetcall – get client's data passed via the listener.

SYNOPSIS

#include <sysf/tiuser.h>

struct t_call *nlsgetcall (*fd*);
int *fd*;

DESCRIPTION

nlsgetcall allows server processes started by the *listener* process to access the client's *t_call* structure, that is, the *sndcall* argument of *t_connect(3N)*.

The *t_call* structure returned by *nlsgetcall* can be released using *t_free(3N)*.

nlsgetcall returns the address of an allocated *t_call* structure or NULL if a *t_call* structure cannot be allocated. If the *t_alloc* succeeds, undefined environment variables are indicated by a negative *len* field in the appropriate *netbuf* structure. A *len* field of zero in the *netbuf* structure is valid and means that the original buffer in the listener's *t_call* structure was NULL.

WARNING

The *len* field in the *netbuf* structure is defined as being unsigned. In order to check for error returns, it should first be cast to an int.

SEE ALSO

nlsadmin(1), getenv(3), t_connect(3N), t_alloc(3N), t_free(3N), t_error(3N)

DIAGNOSTICS

A NULL pointer is returned if a *t_call* structure cannot be allocated by *t_alloc*. *t_errno* can be inspected for further error information. Undefined environment variables are indicated by a negative length field (*len*) in the appropriate *netbuf* structure.

CAVEATS

The listener process limits the amount of user data (*udata*) and options data (*opt*) to 128 bytes each. Address data *addr* is limited to 64 bytes. If the original data was longer, no indication of overflow is given.

FILES

/usr/lib/libnsl_s.a

NOTES

Server processes must call *t_sync(3N)* before calling this routine.

NAME

nlsprovider – get name of transport provider.

SYNOPSIS

char *nlsprovider();

DESCRIPTION

nlsprovider returns a pointer to a NULL terminated character string which contains the name of the transport provider as placed in the environment by the *listener* process. If the variable is not defined in the environment, a NULL pointer is returned.

The environment variable is only available to server processes started by the *listener* process.

SEE ALSO

nlsadmin(1M)

DIAGNOSTICS

If the variable is not defined in the environment, a NULL pointer is returned.

FILES

/usr/lib/libslan.a (7300)
/usr/lib/libnls.a (3B2 Computer)
/usr/lib/libnsl_s.a

NAME

nlsrequest – format and send listener service request message

SYNOPSIS

#include <listen.h>

int nlsrequest (*fd*, *service_code*);
int *fd;*
char **service_code;*

extern int _nlslog, t_errno;
extern char *_nlsrmsg;

DESCRIPTION

Given a virtual circuit to a listener process (*fd*) and a service code of a server process, *nlsrequest* formats and sends a *service request message* to the remote listener process requesting that it start the given service. *nlsrequest* waits for the remote listener process to return a *service request response message*, which is made available to the caller in the static, null terminated data buffer pointed to by *_nlsrmsg*. The *service request response message* includes a success or failure code and a text message. The entire message is printable.

SEE ALSO

nlsadmin(1), t_error(3)

FILES

/usr/lib/libnsl_s.a

DIAGNOSTICS

The success or failure code is the integer return code from *nlsrequest*. Zero indicates success, other negative values indicate *nlsrequest* failures as:

–1: Error encountered by nlsrequest, see t_errno.

Postive values are error return codes from the *listener* process. Mnemonics for these codes are defined in **listen.h**:

2: Request message not interpretable.
3: Request service code unknown.
4: Service code known, but currently disabled.

If non-null, *_nlsrmsg* contains a pointer to a static, NULL terminated character buffer containing the *service request response message*. Note that both *_nlsrmsg* and the data buffer are overwritten by each call to *nlsrequest*.

If *_nlslog* is non-zero, *nlsrequest* prints error messages on stderr. Initially, *_nlslog* is zero.

WARNING

nlsrequest cannot always be certain that the remote server process has been successfully started. In this case, *nlsrequest* returns with no indication of an error and the caller will receive notification of a disconnect event via a **T_LOOK** error before or during the first *t_snd* or *t_rcv* call.

NAME

cfgetospeed, cfgetispeed, cfsetospeed, cfsetispeed – get or set the value of the output and input baud rate

SYNOPSIS

int cfgetospeed (*termio_p*)
struct *termio *termio_p;*

int cfgetispeed (*termio_p*)
struct *termio *termio_p;*

int cfsetospeed (*termio_p, speed*)
struct *termio *termio_p;*
int *speed;*

int cfsetispeed (*termio_p, speed*)
struct *termio *termio_p;*
int *speed;*

DESCRIPTION

These routines are used to get and set input and output baud rates.

cfgetospeed returns the output baud rate stored in *c_cflag* pointed to by *termios_p*.

cfgetispeed returns the input baud rate stored in *c_cflag* pointed to by *termios_p*.

The following baud rate values are supported for the value of *speed*:

0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud

cfsetospeed sets the baud rate stored in *c_cflag* pointed to by *termios_p* to *speed.* B0 is used to terminate the connection; if B0 is specified, the modem control lines will no longer be asserted.

cfsetispeed sets the baud rate stored in *c_cflag* pointed to by *termios_p* to *speed.* If the *speed* is 0, the input rate will be specified by the output rate.

For any particular hardware, unsupported baud rate changes are ignored.

cfsetispeed and *cfsetospeed* only modify the *termios* structure. For the baud rate changes to take place, *tcsetattr*(3P) must be called with the modified structure as an argument.

RETURN VALUE

cfgetispeed and *cfgetospeed* return the appropriate baud rate. *cfsetispeed* and *cfsetospeed* returns zero upon successful completion.

SEE ALSO

termios(7), tcgetattr(3P)

NAME

getgroups – get group access list

SYNOPSIS

#include <sys/types.h>
#include <sys/param.h>

int getgroups (*gidsetlen, gidset*)
int *gidsetlen;*
gid_t **gidset;*

DESCRIPTION

getgroups gets the current group access list of the user process and stores it in the array *gidset*. The parameter *gidsetlen* indicates the number of entries that may be placed in *gidset*.

getgroups returns the actual number of groups returned in *gidset*. No more than **NGROUPS_MAX** , as defined in **<limits.h>** , will ever be returned. If *gidsetlen* is zero, *getgroups* returns the number of supplementary group IDs associated with the calling process without modifying the array pointed to by *gidset*.

RETURN VALUE

If the *getgroups* is successful the number of groups in the group set will be returned. If an error is detected, –1 will be returned and *errno* will be set to indicate the error.

ERRORS

If any of the following conditions occur, -1 will be returned and *errno* set to the corresponding value:

[EINVAL] The argument *gidsetlen* is smaller than the number of groups in the group set.

[EFAULT] The argument *gidset* specifies an invalid address.

SEE ALSO

setgroups (2), initgroups (3X)

NAME

sigsetjmp, siglongjmp - non-local jumps

SYNOPSIS

#include <setjmp.h>

int sigsetjmp (*env*, *savemask*)
sigjmp_buf *env*;
int *savemask*;

void siglongjmp (*env*, *val*)
sigjmp_buf *env*;
int *val*;

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

sigsetjmp saves its stack environment in *env* for later use by *siglongjmp* . If the value of *savemask* is not zero, *sigsetjmp* also saves the process's current signal mask as part of the calling environment. The environment type **sigjmp_buf** is defined in the **<setjmp.h>** header file.

siglongjmp restores the environment saved by the last call of *sigsetjmp* with the corresponding *env* argument. If *env* was initialized by a call to *sigsetjmp* with a non-zero value for *savemask*, *siglongjmp* also restores the saved signal mask.

RETURN VALUE

When *sigsetjmp* has been invoked by the calling process, zero is returned.

After *siglongjmp* is completed, program execution continues as if the corresponding call of *sigsetjmp* (which must not itself have returned in the interim) had just returned the value *val*. *siglongjmp* cannot cause *sigsetjmp* to return the value zero. If *val* is zero, *sigsetjmp* returns 1. All accessible data have values as of the time *siglongjmp* was called.

WARNING

siglongjmp fails if *env* was never initialized by a call to *sigsetjmp* or when the last such call is in a function which has since returned.

SEE ALSO

sigaction(3P), sigprocmask(3P), sigsuspend(3P)

NAME

tcdrain, tcflow, tcflush, tcsendbreak – line control functions

SYNOPSIS

#include <termios.h>

int tcdrain (*fildes*)
int *fildes;*

int tcflow (*fildes, action*)
int *fildes, action;*

int tcflush (*fildes, queue_selector*)
int *fildes, queue_selector;*

int tcsendbreak (*fildes, duration*)
int *fildes, duration;*

DESCRIPTION

tcdrain causes the process to wait until all output written to the object indicated by *fildes* has been transmitted.

tcflow will suspend transmission or reception of data on the object indicated by fildes, depending on the value of *action*. If *action* is **TCOOFF**, output will be suspended. If *action* is **TCOON**, suspended output will be restarted. If *action* is **TCIOF**, input will be suspended. If *action* is *TCION*, suspended input will be restarted.

tcflush will discard data written to the object indicated by *fildes* but not transmitted, or data received but not read, depending on the value of *queue_selector*. If *queue_selector* is **TCIFLUSH**, data received but not read will be flushed. If *queue_selector* is **TCOFLUSH**, data written but not transmitted will be flushed. If *queue_selector* is **TCIOFLUSH**, both data received but not read and data written but not transmitted will be flushed.

tcsendbreak will assert a break condition on the serial line associated with *fildes* depending on the value of *duration*. If *duration* is zero, the break condition will be asserted for 0.25 seconds. If *duration* is not zero, break will last 'duration' milliseconds.

RETURN VALUE

Upon successful completion, zero is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

If any of the following conditions occur, -1 will be returned and *errno* set to the corresponding value:

[EBADF]	*fildes* is not a valid file descriptor.
[EINVAL]	The device does not support the function or if the function called was *tcflush*, *queue_selector* is invalid.
[ENOTTY]	The file associated with *fildes* is not a terminal.

tcdrain will report the following error, in addition to those listed above:

[EINTR]	*tcdrain* was interrupted by a signal.

SEE ALSO

termios(7)

NAME

tcgetattr, tcsetattr – get and set terminal state

SYNOPSIS

#include <termios.h>

int tcgetattr (*fildes*, *termios_p*)
int *fildes*;
struct *termio* **termio_p*;

int tcsetattr (*fildes*, *optional_actions*, *termio_p*)
int *fildes*, *optional_actions*;
struct *termio* **termio_p*;

DESCRIPTION

tcgetattr retrieves the parameters associated with the device indicated by *fildes* and stores them in the *termios* structure indicated by **termios_p**.

tcsetattr sets the parameters associated with the terminal using the information in the *termios* structure pointed to by **termios_p**. The action taken is dependent on the value of **optional_actions**. If **optional_actions** is **TCSANOW**, the change occurs immediately. If **optional_actions** is **TCSADRAIN**, the change occurs after all output written to *fildes* has been transmitted. **TCSADRAIN** should be used when changing parameters that affect output. If **optional_actions** is **TCSADFLUSH**, the change occurs after all output written to the object indicated by *fildes* has been transmitted; all input that has been received but not read is discarded before the change is made.

RETURN VALUE

Upon successful completion, zero will be returned. Otherwise, –1 will be returned and *errno* set to indicate the error.

ERRORS

If any of the following conditions occur, *tcgetattr* and *tcsetattr* will return –1 and set *errno* to the corresponding value:

[EBADF]	*fildes* is not a valid file descriptor.
[EINVAL]	The device does not support the function called, or if the function called was *tcsetattr*, **optional_actions** is an invalid value.
[ENOTTY]	The file associated with *fildes* is not a terminal.
[EFAULT]	**termino_p** is an invalid address.

SEE ALSO

cfgetospeed(3P), termios(7)

NAME

tcgetpgrp – get distinguished process group ID

SYNOPSIS

#include <termios.h>

int tcgetpgrp (*fildes*)
int *fildes;*

DESCRIPTION

tcgetpgrp returns the value of the process group ID of the distinguished process group associated with the terminal.

tcgetpgrp is part of the POSIX Job Control Option.

RETURN VALUE

Upon successful completion, *tcgetpgrp* returns the process group ID of the distinguished process group associated with the terminal. Otherwise, –1 is returned and *errno* is set to indicate the error.

ERRORS

If any of the following conditions occur, *tcgetpgrp* will return –1 and set *errno* to the corresponding value:

[EBADF] *fildes* is not a valid file descriptor.

[EINVAL] *tcgetpgrp* is not permitted for the device associated with *fildes*.

[ENOTTY] The calling process does not have a controlling terminal or the file is not the controlling terminal.

SEE ALSO

setpgrp(2), setpgid(2), tcsetpgrp(3P)

P)

NAME

tcsetpgrp – set distinguished process group ID

SYNOPSIS

#include <termios.h>

int tcsetpgrp (*fildes*, *pgrp_id*)
int *fildes*;
int *pgrp_id*;

DESCRIPTION

If the process has a controlling terminal, *tcsetpgrp* will set the distinguished process group ID associated with the terminal to **pgrp_id**. The file associated with *fildes* must be the controlling terminal of the calling process. There must be at least one process in **pgrp_id** that has the same controlling terminal as the calling process.

tcsetpgrp is part of the POSIX Job Control Option.

RETURN VALUE

Upon successful completion, *tcsetpgrp* returns zero. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

[EBADF] *fildes* is not a valid file descriptor.

[EINVAL] *tcsetpgrp* is not permitted for the device associated with *fildes* or the value of **pgrp_id** is less than or equal to zero or exceeds **{PID_MAX}**.

[ENOTTY] The calling process does not have a controlling terminal or the file is not the controlling terminal.

[EPERM] **pgrp_id** is greater than zero and less than or equal to **{PID_MAX}**, but does not match the process group ID of a process in the same session as the calling process.

SEE ALSO

setpgrp(2), setpgid(2), tcgetpgrp(3P)

NAME

ctermid – generate file name for terminal

SYNOPSIS

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

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

cuserid – get character login name of the user

SYNOPSIS

#include <stdio.h>

char *cuserid (s)
char *s;

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]*.

SEE ALSO

getlogin(3C), getpwent(3C)

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

#include <stdio.h>

int fclose (*stream*)
FILE **stream;*

int fflush (*stream*)
FILE **stream;*

DESCRIPTION

fclose causes any buffered data for the named *stream* to be written out, and the *stream* to be closed.

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.

SEE ALSO

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

DIAGNOSTICS

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

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;**

DESCRIPTION

ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise, zero.

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).

NOTES

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

SEE ALSO

open(2), fopen(3S), stdio(3S)

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

```
#include <stdio.h>

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

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

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

DESCRIPTION

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

filename 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 EOF, 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 EOF

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), pipe(2), fclose(3S), fseek(3S), stdio(3S)

DIAGNOSTICS

fopen, *fdopen*, and *freopen* return a **NULL** pointer on failure.

(3

NAME

fread, fwrite – binary input/output

SYNOPSIS

```
#include <stdio.h>
#include <sys/types.h>

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

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

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), stdio(3S)

DIAGNOSTICS

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

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*;

void rewind (*stream*)
FILE ***stream*;**

long ftell (*stream*)
FILE ***stream*;**

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), except that no value is returned.

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.

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), stdio(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).

WARNING

Although on SYSTEM V/88, 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

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

SYNOPSIS

#include <stdio.h>

int getc (*stream*)
FILE **stream;*

int getchar ()

int fgetc (*stream*)
FILE **stream;*

int getw (*stream*)
FILE **stream;*

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.

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., integer) 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), stdio(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.

CAVEATS

Because it is implemented as a macro, *getc* evaluates a *stream* argument more than once. 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 processor.

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;*

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 EOF condition is encountered. The newline 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 newline character is read and transferred to *s*, or an EOF condition is encountered. The string is then terminated with a **NULL** character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S), stdio(3S)

DIAGNOSTICS

If EOF 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

popen, pclose – initiate pipe to/from a process

SYNOPSIS

#include <stdio.h>

FILE **popen* (*command*, *type*)
char **command*, **type*;

int pclose (*stream*)
FILE **stream*;

DESCRIPTION

popen creates a pipe between the calling program and the command to be executed. The arguments to *popen* are pointers to null-terminated strings. *Command* consists of a shell command line. *type* is an I/O mode, either **r** for reading or **w** for writing. 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.

EXAMPLE

The following is a typical call:

```
char *cmd = "ls *.c";
FILE *ptr;
if ((ptr = popen(cmd, "r")) != NULL)
        while (fgets(buf, n, ptr) != NULL)
                (void) printf("%s ",buf);
```

This will print in *stdout* [see *stdio* (3S)] all the file names in the current directory that have a ".c" suffix.

SEE ALSO

pipe(2), wait(2), fclose(3S), fopen(3S), stdio(3S), system(3S)

DIAGNOSTICS

popen returns a **NULL** pointer if files or processes cannot be created.

pclose returns –1 if *stream* is not associated with a "popened" command.

WARNING

If the original and ''popened'' 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*;

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. The padding is with blanks unless the field width digit string starts with a zero, in which case the padding is with zeros.

A *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **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. Padding specified by the precision overrides the padding specified by the field width.

An optional **l** (ell) specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*. An **l** before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision or both 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. A negative field width argument is taken as a '**–**' flag followed by a positive field width. If the precision argument is negative, it will be changed to zero.

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**, **i**, **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,i,o,u,x,X
: The integer *arg* is converted to signed decimal (**d** or **i**), unsigned octal, (**o**), decimal (**u**), or hexadecimal notation (**x** or **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. 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±dd," 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 at least two 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 printing floating point types (float and double), if the exponent is 0x7FF and the mantissa is not equal to zero, then the output is:

[-]NaN0xdddddddd

where 0xdddddddd is the hexadecimal representation of the leftmost 32 bits of the mantissa. If the mantissa is zero, the output is:

[±]inf.

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 %i, %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)

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;***

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 function rather than a macro. *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., integer) *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.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S), stdio(3S)

DIAGNOSTICS

On success, these functions (with the exception of *putw*) each return the value they have written. (*putw* returns *ferror (stream)*). 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 grow. Because EOF is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

CAVEATS

Because it is implemented as a macro, *putc* evaluates a *stream* argument more than once. 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 processor.

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;*

DESCRIPTION

puts writes the null-terminated string pointed to by *s*, followed by a newline 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.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S), stdio(3S)

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.

NOTES

puts appends a newline character while *fputs* does not.

NAME

scanf, fscanf, sscanf – convert formatted input

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*;

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 results are undefined in there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s are simply ignored.

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, newlines, 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.

i an integer is expected; the corresponding argument should be an integer pointer. It will store the value of the next input item interpreted according to C conventions: a leading "0" implies octal; a leading "0x" implies hexadecimal; otherwise, decimal.

n stores in an integer argument the total number of characters (including white space) that have been scanned so far since the function call. No input is consumed.

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.

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 is called 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**, **x** and **i** 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 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, j; float x; char name[50];
(void) scanf("%i%2d%f%*d %[0–9] ", &j, &i, &x, name);
```

with input:

```
011 56789 0123 56a72
```

will assign **9** to *j*, **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**. Or:

```
int i, j, s, e; char name[50];
(void) scanf("%i %i %n%s%n", &i, &j, &s, name, &e);
```

with input:

```
0x11 0xy johnson
```

will assign **17** to *i*, **0** to *j*, **6** to *s*, will place the string **xy\0** in *name*, and will assign **8** to *e*. Thus, the length of *name* is *e* - *s* = 2 . The next call to *getchar* [see *getc*(3S)] will return a blank.

SEE ALSO

getc(3S), printf(3S), stdio(3S), strtod(3C), strtol(3C)

DIAGNOSTICS

These functions return EOF, on end of input and a short count for missing or illegal data items.

CAVEATS

Trailing white space (including a newline) is left unread unless matched in the control string.

NAME

setbuf, setvbuf – assign buffering to a stream

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;*

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. *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.

S)

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.

NOTES

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

stdio – standard buffered input/output package

SYNOPSIS

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The inline 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*.

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). *setbuf*(3S) or *setvbuf*() in *setbuf*(3S) may be used to change the stream's buffering strategy.

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

system – issue a shell command

SYNOPSIS

#include <stdio.h>

int system (string)
char *string;

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

exec(2)
wait(2)
sh(1) in the *User's Reference Manual*

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*.

If the shell fails to execute, a status of 127 is returned. If the shell executes successfully, a status of 0 is returned.

NAME

tmpfile – create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile ()

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), stdio(3S), 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;*

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. 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 that 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.

SEE ALSO

creat(2), unlink(2), fopen(3S), malloc(3C), mktemp(3C), tmpfile(3S)

CAVEATS

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 to render duplication by other means unlikely.

NAME

ungetc – push character back into input stream

SYNOPSIS

#include <stdio.h>

int ungetc (*c, stream*)
int *c;*
FILE **stream;*

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.

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), stdio(3S)

DIAGNOSTICS

ungetc returns EOF if it cannot insert the character.

BUGS

When *stream* is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

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;*

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 the use of *vfprintf* to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
 *     error should be called like
 *      error(function_name, format, arg1, arg2...);  */
/*VARARGS*/
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

assert – verify program assertion

SYNOPSIS

#include <assert.h>

assert (*expression*)
int *expression;*

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)

CAVEAT

Since *assert* is implemented as a macro, the *expression* may not contain any string literals.

NAME

crypt – password and file encryption functions

SYNOPSIS

cc [flag ...] file ... –lcrypt

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

void setkey (*key*)
**char ****key;*

void encrypt (*block, flag*)
**char ****block;*
int *flag;*

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

void des_setkey (*key*)
**char ****key;*

void des_encrypt (*block, flag*)
**char ****block;*
int *flag;*

int run_setkey (*p, key*)
int *p*[2];
**char ****key;*

int run_crypt (*offset, buffer, count, p*)
long *offset;*
**char ****buffer;*
unsigned int *count;*
int *p*[2];

int crypt_close (*p*)
int *p*[2];

DESCRIPTION

des_crypt is the password encryption function. It is based on a one way hashing encryption algorithm with variations intended (among other things) to frustrate use of hardware implementations of a key search.

key is a user's typed password. *salt* is a two-character string chosen from the set **[a-zA-Z0-9./]**; this string is used to perturb the hashing 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. The first two characters are the salt itself.

The *des_setkey* and *des_encrypt* entries provide (rather primitive) access to the actual hashing algorithm. The argument of *des_setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. This is the key that will be used with the hashing algorithm to encrypt the string *block* with the function *des_encrypt*.

The argument to the *des_encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by *des_setkey*. If *edflag* is zero, the argument is encrypted; if non-zero, it is decrypted.

Note that decryption is not provided in the international version of *crypt*(3X). The international version is part of the C *Programming Language Utilities*, and the domestic version is part of the *Security Administration Utilities*. If decryption is attempted with the international version of *des_encrypt*, an error message is printed.

crypt, *setkey*, and *encrypt* are front-end routines that invoke *des_crypt*, *des_setkey*, and *des_encrypt* respectively.

The routines *run_setkey* and *run_crypt* are designed for use by applications that need cryptographic capabilities (such as *ed*(1) and *vi*(1)) that must be compatible with the *crypt*(1) user-level utility. *run_setkey* establishes a two-way pipe connection with *crypt*(1), using *key* as the password argument. *run_crypt* takes a block of characters and transforms the cleartext or ciphertext into their ciphertext or cleartext using *crypt*(1). *Offset* is the relative byte position from the beginning of the file that the block of text provided in *block* is coming from. *count* is the number of characters in *block*, and *connection* is an array containing indices to a table of input and output file streams. When encryption is finished, *crypt_close* is used to terminate the connection with *crypt*(1).

run_setkey returns -1 if a connection with *crypt*(1) cannot be established. This will occur on international versions of UNIX where *crypt*(1) is not available. If a null key is passed to *run_setkey*, 0 is returned. Otherwise, 1 is returned. *run_crypt* returns -1 if it cannot write output or read input from the pipe attached to *crypt*. Otherwise, it returns 0.

DIAGNOSTICS

In the international version of *crypt*(3X), a flag argument of 1 to *des_encrypt* is not accepted, and an error message is printed.

SEE ALSO

getpass(3C), passwd(4)
crypt(1), login(1), passwd(1) in the *User's Reference Manual*.

CAVEAT

The return value in *crypt* points to static data that are overwritten by each call.

NAME

curses – terminal screen handling and optimization package

SYNOPSIS

cc [flag ...] file ... **–lcurses** [library ...]

#include <curses.h> (automatically includes **<stdio.h>**, **<termio.h>**, and **<unctrl.h>**).

The parameters in the following summary are the arguments used by the *curses* library routines: they are not global variables. All routines return the **int** values **ERR** or **OK** unless it is stated otherwise in the ROUTINES section. Routines that return pointers always return **NULL** on error. (**ERR**, **OK**, and **NULL** are all defined in **<curses.h>**.)

bool bf;

char **area,*boolnames[ ], *boolcodes[ ], *boolfnames[ ], *bp;
char *cap, *capname, codename[2], erasechar, *filename, *fmt;
char *keyname, killchar, *label, *longname;
char *name, *numnames[ ], *numcodes[ ], *numfnames[ ];
char *slk_label, *str, *strnames[ ], *strcodes[ ], *strfnames[ ];
char *term, *tgetstr, *tigetstr, *tgoto, *tparm, *type;

chtype attrs, ch, horch, vertch;

FILE *infd, *outfd;

int begin_x, begin_y, begline, bot, c, col, count;
int dmaxcol, dmaxrow, dmincol, dminrow, *errret, fildes;
int (*init( )), labfmt, labnum, line;
int ms, ncols, new, newcol, newrow, nlines, numlines;
int oldcol, oldrow, overlay;
int p1, p2, p9, pmincol, pminrow, (*putc( )), row;
int smaxcol, smaxrow, smincol, sminrow, start;
int tenths, top, visibility, x, y;
short pair, color, f, r, g, b;

SCREEN *new, *newterm, *set_term;
TERMINAL *cur_term, *nterm, *oterm;

va_list varglist;

WINDOW *curscr, *dstwin, *initscr, *newpad, *newwin, *orig;
WINDOW *pad, *srcwin, *stdscr, *subpad, *subwin, *win;

NAME

addch(ch)
addstr(str)
attroff(attrs)
attron(attrs)
attrset(attrs)
baudrate( )
beep( )
box(win, vertch, horch)
can_change_color( )
cbreak( )
clear( )
clearok(win, bf)
clrtobot( )
clrtoeol( )
color_content(color, &r, &g, &b)
copywin(srcwin, dstwin, sminrow, smincol,
  dminrow, dmincol, dmaxrow, dmaxcol, overlay)
curs_set(visibility)
def_prog_mode( )
def_shell_mode( )
del_curterm(oterm)
delay_output(ms)
delch( )
deleteln( )
delwin(win)
doupdate( )
draino(ms)
echo( )
echochar(ch)
endwin( )
erase( )
erasechar( )
filter( )
flash( )
flushinp( )
garbagedlines(win, begline, numlines)
getbegyx(win, y, x)
getch( )
getmaxyx(win, y, x)

NAME

getstr(str)
getsyx(y, x)
getyx(win, y, x)
halfdelay(tenths)
has_colors( )
has_ic( )
has_il( )
idlok(win, bf)
inch( )
init_color(color, r, g, b)
init_pair(pair, f, b)
initscr( )
insch(ch)
insertln( )
intrflush(win, bf)
isendwin( )
keyname(c)
keypad(win, bf)
killchar( )
leaveok(win, bf)
longname( )
meta(win, bf)
move(y, x)
mvaddch(y, x, ch)
mvaddstr(y, x, str)
mvcur(oldrow, oldcol, newrow, newcol)
mvdelch(y, x)
mvgetch(y, x)
mvgetstr(y, x, str)
mvinch(y, x)
mvinsch(y, x, ch)
mvprintw(y, x, fmt [, arg...])
mvscanw(y, x, fmt [, arg...])
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, y, x)

NAME

mvwinch(win, y, x)
mvwinsch(win, y, x, ch)
mvwprintw(win, y, x, fmt [, arg...])
mvwscanw(win, y, x, fmt [, arg...])
napms(ms)
newpad(nlines, ncols)
newterm(type, outfd, infd)
newwin(nlines, ncols, begin_y, begin_x)
nl()
nocbreak()
nodelay(win, bf)
noecho()
nonl()
noraw()
notimeout(win, bf)
overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)
pair_content(pair, &f, &b)
pechochar(pad, ch)
pnoutrefresh(pad, pminrow, pmincol, sminrow,
  smincol, smaxrow, smaxcol)
prefresh(pad, pminrow, pmincol, sminrow,
  smincol, smaxrow, smaxcol)
printw(fmt [, arg...])
putp(str)
raw()
refresh()
reset_prog_mode()
reset_shell_mode()
resetty()
restartterm(term, fildes, errret)
ripoffline(line, init)
savetty()
scanw(fmt [, arg...])
scr_dump(filename)
scr_init(filename)
scr_restore(filename)
scroll(win)
scrollok(win, bf)

NAME

set_curterm(nterm)
set_term(new)
setscrreg(top, bot)
setsyx(y, x)
setupterm(term, fildes, errret)
slk_attroff(attrs)
slk_attron(attrs)
slk_attrset(attrs)
slk_clear( )
slk_init(fmt)
slk_label(labnum)
slk_noutrefresh( )
slk_refresh( )
slk_restore( )
slk_set(labnum, label, fmt)
slk_touch( )
standend( )
standout( )
start_color( )
subpad(orig, nlines, ncols, begin_y, begin_x)
subwin(orig, nlines, ncols, begin_y, begin_x)
tgetent(bp, name)
tgetflag(codename)
tgetnum(codename)
tgetstr(codename, area)
tgoto(cap, col, row)
tigetflag(capname)
tigetnum(capname)
tigetstr(capname)
touchline(win, start, count)
touchwin(win)
tparm(str, p1, p2, . . ., p9)
tputs(str, count, putc)
traceoff( )
traceon( )
typeahead(fildes)
unctrl(c)
ungetch(c)
vidattr(attrs)

X)

NAME

vidputs(attrs, putc)
vwprintw(win, fmt, varglist)
vwscanw(win, fmt, varglist)
waddch(win, ch)
waddstr(win, str)
wattroff(win, attrs)
wattron(win, attrs)
wattrset(win, attrs)
wclear(win)
wclrtobot(win)
wclrtoeol(win)
wdelch(win)
wdeleteln(win)
wechochar(win, ch)
werase(win)
wgetch(win)
wgetstr(win, str)
winch(win)
winsch(win, ch)
winsertln(win)
wmove(win, y, x)
wnoutrefresh(win)
wprintw(win, fmt [, arg...])
wrefresh(win)
wscanw(win, fmt [, arg...])
wsetscrreg(win, top, bot)
wstandend(win)
wstandout(win)

DESCRIPTION

The *curses* routines give the user a terminal-independent method of updating screens with reasonable optimization.

The file **<curses.h>** must be included at the beginning of programs that use any *curses* routines. In addition, the routine **initscr**() or **newterm**() must be called before any of the other routines that deal with windows and screens are used. (Three exceptions are noted where they apply.) The routine **endwin**() must be called before exiting. To get character-at-a-time input without echoing (most interactive, screen-oriented programs want this), after calling **initscr**() you should call "**cbreak**(); **noecho**();" Most programs would additionally call "**nonl**(); **intrflush (stdscr, FALSE); keypad(stdscr, TRUE);**".

Before a *curses* program is run, a terminal's tab stops should be set and its initialization strings, if defined, must be output. To do this, execute **tput init** after the shell environment variable **TERM** has been set and exported. For further details, see *profile*(4), *tput*(1), and the "Tabs and Initialization" subsection of *terminfo*(4).

The *curses* library contains routines that manipulate data structures called *windows* that can be thought of as two-dimensional arrays of characters representing all or part of a terminal screen. A default window called **stdscr** is supplied, which is the size of the terminal screen. Others may be created with **newwin**(). Windows are referred to by variables declared as **WINDOW ***; the type **WINDOW** is defined in **<curses.h>** to be a structure. These data structures are manipulated with routines described below, among which the most basic are **move**() and **addch**(). (More general versions of these routines are included, with names beginning with **w**, allowing you to specify a window. The routines not beginning with **w** usually affect **stdscr**.) Then **refresh**() is called, telling the routines to make the user's terminal screen look like **stdscr**. The characters in a window are actually of type **chtype**, defined in **<curses.h>**, so that other information about the character may also be stored with each character.

Special windows called *pads* may also be manipulated. These are windows which are not constrained to the size of the screen and whose contents need not be displayed completely. See the description of **newpad**( ) under "Window and Pad Manipulation" for more information.

In addition to drawing characters on the screen, video attributes may be included which cause the characters to be underlined or shown in reverse video on terminals that support such display enhancements. Line drawing characters may be specified to be output. On input, *curses* is also able to translate arrow and function keys that transmit escape sequences into single values. The video attributes, line drawing characters, and input values use names, defined in **<curses.h>**, such as **A_REVERSE**, **ACS_HLINE**, and **KEY_LEFT**.

Routines that manipulate color on color alphanumeric terminals are new in this release of *curses*. To use these routines **start_color**( ) must be called, usually right after **initscr**( ). Colors are always used in pairs (referred to as color-pairs). A color-pair consists of a foregound color (for characters) and a background color (for the field the characters are displayed on). A programmer initializes a color-pair with the routine **init_pair**( ). After it has been initialized, **COLOR_PAIR**(n), a macro defined in **<curses.h>**, can be used in the same ways other video attributes can be used. If a terminal is capable of redefining colors the programmer can use the routine **init_color**( ) to change the definition of a color. The routines **has_color**( ) and **can_change_color**( ) return **TRUE** or **FALSE**, depending on whether the terminal has color capabilities and whether the user can change the colors. The routine **color_content**( ) allows a user to identify the amounts of red, green, and blue components in an initialized color. The routine **pair_content**( ) allows a user to find out how a given color-pair is currently defined.

curses also defines the **WINDOW *** variable, **curscr**, which is used only for certain low-level operations like clearing and redrawing a garbaged screen. **curscr** can be used in only a few routines. If the window argument to **clearok**( ) is **curscr**, the next call to **wrefresh**( ) with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh**( ) is **curscr**, the screen is immediately cleared and repainted from scratch. This is how most programs would implement a "repaint-screen" function. More information on using **curscr** is provided where its use is appropriate.

The environment variables **LINES** and **COLUMNS** may be set to override **terminfo**'s idea of how large a screen is. These may be used in an AT&T Teletype 5620 layer, for example, where the size of a screen is changeable.

If the environment variable **TERMINFO** is defined, any program using *curses* will check for a local terminal definition before checking in the standard place. For example, if the environment variable **TERM** is set to **att4425,** then the compiled terminal definition is found in */usr/lib/terminfo/a/att4425*. (The **a** is copied from the first letter of **att4425** to avoid creation of huge directories.) However, if **TERMINFO** is set to *$HOME/myterms*, *curses* will first check *$HOME/myterms/a/att4425*, and, if that fails, will then check */usr/lib/terminfo/a/att4425*. This is useful for developing experimental definitions or when write permission on */usr/lib/terminfo* is not available.

The integer variables **LINES** and **COLS** are defined in **<curses.h>**, and will be initialized by **initscr**( ) with the size of the screen. (For more information, see the subsection "Terminfo-Level Manipulations".) The integer variables **COLORS** and **COLOR_PAIRS** are also defined in **<curses.h>** and contain, respectively, the maximum number of colors and color-pairs the terminal can support. They are initialized by **start_color**( ). The constants **TRUE** and **FALSE** have the values **1** and **0**, respectively. The constants **ERR** and **OK** are returned by routines to indicate whether the routine successfully completed. These constants are also defined in **<curses.h>**.

ROUTINES

Many of the following routines have two or more versions. The routines prefixed with **w** require a *window* argument. The routines prefixed with p require a *pad* argument. Those without a prefix generally use **stdscr**.

The routines prefixed with **mv** require *y* and *x* coordinates to move to before performing the appropriate action. The **mv**( ) routines imply a call to **move**( ) before the call to the other routine. The window argument is always specified before the coordinates. *y* always refers to the row (of the window), and *x* always refers to the column. The upper left corner is always **(0,0)**, not **(1,1)**. The routines prefixed with **mvw** take both a *window* argument and *y* and *x* coordinates.

In each case, *win* is the window affected and *pad* is the pad affected. (**win** and **pad** are always of type **WINDOW ***.) Option-setting routines require a boolean flag *bf* with the value **TRUE** or **FALSE.** (*bf* is always of type **bool**.) The types **WINDOW, bool,** and **chtype** are defined in **<curses.h>**. See the SYNOPSIS for a summary of what types all variables are.

All routines return either the integer **ERR** or the integer **OK,** unless otherwise noted. Routines that return pointers always return **NULL** on error.

Sometimes the description of a routine refers to a second routine. If the routine referred to is prefixed with a **w**, then you should assume that other versions of the second routine behave similarly. For example, the description of **initscr**( ) refers to **wrefresh**( ). This implies that the same result will occur if **refresh**( ) is called.

Section 1: Overall Screen Manipulation

WINDOW *initscr( )
The first routine called should almost always be **initscr**( ). (The exceptions are **slk_init**( ), **filter**( ), and **ripoffline**( ).) This will determine the terminal type and initialize all *curses* data structures. **initscr**( ) also arranges that the first call to **wrefresh**( ) will clear the screen. If errors occur, **initscr**( ) will write an appropriate error message to standard error and exit; otherwise, a pointer to **stdscr** is returned. If the program wants an indication of error conditions, **newterm**( ) should be used instead of **initscr**( ). **initscr**( ) should only be called once per application.

endwin( ) A program should always call **endwin**( ) before exiting or escaping from *curses* mode temporarily, to do a shell escape or *system*(3S) call, for example. This routine will restore *tty*(7) modes, move the cursor to the lower left corner of the screen and reset the terminal into the proper non-visual mode. To resume after a temporary escape, call **wrefresh**( ) or **doupdate**( ).

isendwin( )
Returns **TRUE** if **endwin**( ) has been called without any subsequent calls to **wrefresh**( ).

SCREEN ***newterm**(type, outfd, infd)

A program that outputs to more than one terminal must use **newterm**( ) for each terminal instead of **initscr**( ). A program that wants an indication of error conditions, so that it may continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, must also use this routine. **newterm**( ) should be called once for each terminal. It returns a variable of type **SCREEN*** that should be saved as a reference to that terminal. The arguments are the *type* of the terminal to be used in place of the environment variable **TERM**; *outfd*, a *stdio*(3S) file pointer for output to the terminal; and *infd*, another file pointer for input from the terminal. When it is done running, the program must also call **endwin**( ) for each terminal being used. If **newterm**( ) is called more than once for the same terminal, the first terminal referred to must be the last one for which **endwin**( ) is called.

SCREEN ***set_term**(new)

This routine is used to switch between different terminals. The screen reference *new* becomes the new current terminal. A pointer to the screen of the previous terminal is returned by the routine. This is the only routine which manipulates SCREEN pointers; all other routines affect only the current terminal.

Section 2: Window and Pad Manipulation

refresh( )

wrefresh (win)

These routines (or **prefresh**( ), **pnoutrefresh**( ), **wnoutrefresh**( ), or **doupdate**( )) must be called to write output to the terminal, as most other routines merely manipulate data structures. **wrefresh**( ) copies the named window to the physical terminal screen, taking into account what is already there in order to minimize the amount of information that's sent to the terminal (called optimization). **refresh**( ) does the same thing, except it uses **stdscr** as a default window. Unless **leaveok**( ) has been enabled, the physical cursor of the terminal is left at the location of the window's cursor. The number of characters output to the terminal is returned.

Note that **refresh**( ) is a macro.

wnoutrefresh(win)
doupdate( )

These two routines allow multiple updates to the physical terminal screen with more efficiency than **wrefresh**( ) alone. How this is accomplished is described in the next paragraph.

curses keeps two data structures representing the terminal screen: a *physical* terminal screen, describing what is actually on the screen, and a *virtual* terminal screen, describing what the programmer wants to have on the screen. **wrefresh**( ) works by first calling **wnoutrefresh**( ), which copys the named window to the virtual screen, and then by calling **doupdate**( ), which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to **wrefresh**( ) will result in alternating calls to **wnoutrefresh**( ) and **doupdate**( ), causing several bursts of output to the screen. By first calling **wnoutrefresh**( ) for each window, it is then possible to call **doupdate**( ) once, resulting in only one burst of output, with probably fewer total characters transmitted and certainly less processor time used.

WINDOW ***newwin**(nlines, ncols, begin_y, begin_x)

Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The upper left corner of the window is at line *begin_y*, column *begin_x*. If either *nlines* or *ncols* is **0**, they will be set to the value of **lines**–*begin_y* and **cols**–*begin_x*. A new full-screen window is created by calling **newwin(0, 0, 0, 0)**.

mvwin(win, y, x)

Move the window so that the upper left corner will be at position (*y*, *x*). If the move would cause any portion of the window to be moved off the screen, it is an error and the window is not moved.

WINDOW ***subwin**(orig, nlines, ncols, begin_y, begin_x)

Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The window is at position (*begin_y*, *begin_x*) on the screen. (This position is relative to the screen, and not to the window *orig*.) The window is made in the middle of the window *orig*, so that changes made to one window will affect the character image of both windows. When changing the image of a subwindow, it will be necessary to call **touchwin**( ) or **touchline**( ) on *orig* before calling **wrefresh**( ) on *orig*.

delwin(win)

Delete the named window, freeing all memory associated with it. If you try to delete a main window before all of its subwindows have been deleted, ERR will be returned.

WINDOW ***newpad**(nlines, ncols)

Create and return a pointer to a new pad data structure with the given number of lines (or rows), *nlines*, and columns, *ncols*. A pad is a window that is not restricted by the screen size and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g. from scrolling or echoing of input) do not occur. It is not legal to call **wrefresh**( ) with a pad as an argument; the routines **prefresh**( ) or **pnoutrefresh**( ) should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

WINDOW ***subpad**(orig, nlines, ncols, begin_y, begin_x)

Create and return a pointer to a subwindow within a pad with the given number of lines (or rows), *nlines*, and columns, *ncols*. Unlike **subwin**( ), which uses screen coordinates, the window is at position (*begin_y*, *begin_x*) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window will affect the character image of both windows. When changing the image of a subwindow, it will be necessary to call **touchwin**( ) or **touchline**( ) on *orig* before calling **prefresh**( ) on *orig*.

X)

prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol

These routines are analogous to **wrefresh**( ) and **wnoutrefresh**( ) except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left corner, in the pad, of the rectangle to be displayed. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

Section 3: Output

These routines are used to manipulate text in windows.

addch(ch)
waddch(win, ch)
mvaddch(y, x, ch)
mvwaddch(win, y, x, ch)

The character *ch* is put into the window at the current cursor position of the window and the position of the window cursor is advanced. Its function is similar to that of *putchar* (see *putc*(3S)). At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok**( ) is enabled, the scrolling region will be scrolled up one line.

If *ch* is a tab, newline, or backspace, the cursor will be moved appropriately within the window. A newline also does a **wclrtoeol**( ) before moving. Tabs are considered to be at every eighth column. If *ch* is another control character, it will be drawn in the ^X notation. (Calling **winch**( ) on a position in the window containing a control character will not return the control character, but instead will return one character of the representation of the control character.)

Video attributes can be combined with a character by OR-ing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using **winch**( ) and **waddch**( ).) See **wstandout**( ), below.

Note that *ch* is actually of type **chtype**, not a character.

Note that **addch**( ), **mvaddch**( ), and **mvwaddch**( ), are macros.

echochar(ch)
wechochar(win, ch)
pechochar(pad, ch)

> These routines are functionally equivalent to a call to **addch**(ch) followed by a call to **refresh**( ), a call to **waddch**(win, ch) followed by a call to **wrefresh**(win), or a call to **waddch**(pad, ch) followed by a call to **prefresh**(pad). The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain can be seen by using these routines instead of their equivalents. In the case of **pechochar**( ), the last location of the pad on the screen is reused for the arguments to **prefresh**( ).
>
> Note that *ch* is actually of type **chtype**, not a character.
>
> Note that **echochar**( ) is a macro.

addstr(str)
waddstr(win, str)
mvwaddstr(win, y, x, str)
mvaddstr(y, x, str)

> These routines write all the characters of the null-terminated character string *str* on the given window. This is equivalent to calling **waddch**( ) once for each character in the string.
>
> Note that **addstr**( ), **mvaddstr**( ), and **mvwaddstr**( ) are macros.

attroff(attrs)
wattroff(win, attrs)
attron(attrs)
wattron(win, attrs)
attrset(attrs)
wattrset(win, attrs)
standend( )
wstandend(win)
standout( )
wstandout(win)

These routines manipulate the current attributes of the named window. These attributes can be any combination of the constants **A_STANDOUT, A_REVERSE, A_BOLD, A_DIM, A_BLINK, A_UNDERLINE**, and **A_ALTCHARSET**, as well as the macro **COLOR_PAIR**(n). These attributes are defined in <**curses.h**> and can be combined with the C logical OR (|) operator.

The current attributes of a window are applied to all characters that are written into the window with **waddch**( ). Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of the characters put on the screen.

wattrset(win, attrs) sets the current attributes of the given window to *attrs*. **wattroff**(win, attrs) turns off the named attributes without turning on or off any other attributes. **wattron**(win, attrs) turns on the named attributes without affecting any others. **wstandout**(win, attrs) is the same as **wattron**(win, **A_STANDOUT**). **wstandend**(win, attrs) is the same as **wattrset**(win, **0**), that is, it turns off all attributes.

Note that **wattroff**( ), **wattron**( ), **wattrset**( ), **wstandend**( ), and **wstandout**( ) return **1** at all times.

Note that *attrs* is actually of type **chtype**, not a character.

Note that **attroff**( ), **attron**( ), **attrset**( ), **standend**( ), and **standout**( ) are macros.

beep( )
flash( ) These routines are used to signal the user. **beep**() will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. **flash**( ) will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible, nothing will happen. Nearly all terminals have an audible signal (bell or beep) but only some can flash the screen.

box(win, vertch, horch)
A box is drawn around the edge of the window, *win*. *vertch* and *horch* are the characters the box is to be drawn with. If *vertch* and *horch* are **0**, then appropriate default characters, **ACS_VLINE** and **ACS_HLINE**, will be used.

Note that *vertch* and *horch* are actually of type **chtype**, not characters.

erase( )
werase(win)
These routines copy blanks to every position in the window.

Note that **erase**( ) is a macro.

clear( )
wclear(win)
These routines are like **erase**( ) and **werase**( ), but they also call **clearok**( ), arranging that the screen will be cleared completely on the next call to **wrefresh**( ) for that window, and repainted from scratch.

Note that **clear**( ) is a macro.

clrtobot( )
wclrtobot(win)
All lines below the cursor in this window are erased. Also, the current line to the right of the cursor, inclusive, is erased.

Note that **clrtobot**( ) is a macro.

clrtoeol( )
wclrtoeol(win)
The current line to the right of the cursor, inclusive, is erased.

Note that **clrtoeol**( ) is a macro.

delay_output(ms)

Insert a *ms* millisecond pause in the output. It is not recommended that this routine be used extensively, because padding characters are used rather than a processor pause.

delch( )
wdelch(win)
mvdelch(y, x)
mvwdelch(win, y, x)

The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "delete-character" feature.)

Note that **delch**( ), **mvdelch**( ), and **mvwdelch**( ) are macros.

deleteln( )
wdeleteln(win)

The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change. (This does not imply use of the hardware "delete-line" feature.)

Note that **deleteln**( ) is a macro.

getyx(win, y, x)

The cursor position of the window is placed in the two integer variables *y* and *x*.

Note that **getyx**( ) is a macro, so no "&" is necessary before the variables *y* and *x*.

getbegyx(win, y, x)
getmaxyx(win, y, x)

The current beginning coordinates (**getbegyx**( )) or size (**getmaxyx**( )) of the specified window are placed in the two integer variables *y* and *x*.

Note that **getbegyx**( ) and **getmaxyx**( ) are macros, so no "&" is necessary before the variables *y* and *x*.

insch(ch)
winsch(win, ch)
mvwinsch(win, y, x, ch)
mvinsch(y, x, ch)

The character *ch* is inserted before the character under the cursor. All characters to the right are moved one space to the right, losing the rightmost character of the line. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "insert-character" feature.)

Note that *ch* is actually of type **chtype**, not a character.

Note that **insch**( ), **mvinsch**( ), and **mvwinsch**( ) are macros.

insertln( )
winsertln(win)

A blank line is inserted above the current line and the bottom line is lost. (This does not imply use of the hardware "insert-line" feature.)

Note that **insertln**( ) is a macro.

move(y, x)
wmove(win, y, x)

The cursor associated with the window is moved to line (row) *y*, column *x*. This does not move the physical cursor of the terminal until **wrefresh**( ) is called. The position specified is relative to the upper left corner of the window, which is (**0**, **0**).

Note that **move**( ) is a macro.

overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)

These routines overlay text from *srcwin* on top of text from *dstwin* wherever the two windows overlap. The difference is that **overlay**( ) is non-destructive (blanks are not copied), while **overwrite**( ) is destructive.

copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)

This routine provides finer control over the **overlay**( ) and **overwrite**( ) routines. As in the **prefresh**( ) routine, a rectangle is specified in the destination window, (*dminrow*, *dmincol*) and (*dmaxrow*, *dmaxcol*), and the upper-left-corner coordinates of the source window, (*sminrow*, *smincol*). If the argument *overlay* is true, then copying is non-destructive, as in **overlay**( ).

printw(fmt [, arg . . .])
wprintw(win, fmt [, arg . . .])
mvprintw(y, x, fmt [, arg . . .])
mvwprintw(win, y, x, fmt [, arg . . .])

These routines are analogous to **printf**(3). The string which would be output by **printf**(3) is instead output using **waddstr**( ) on the given window.

vwprintw(win, fmt, varglist)

This routine corresponds to *vfprintf*(3S). It performs a **wprintw**( ) using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in **<varargs.h>**. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

scroll(win)

The window is scrolled up one line. This involves moving the lines in the window data structure.

touchwin(win)
touchline(win, start, count)

Throw away all optimization information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change. **touchline**( ) only pretends that *count* lines have been changed, beginning with line *start* .

Section 4: Input

getch()
wgetch(win)
mvgetch(y, x)
mvwgetch(win, y, x)

A character is read from the terminal associated with the window. In NODELAY mode, if there is no input waiting, the value **ERR** is returned. In DELAY mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**( ), this will be after one character (CBREAK mode), or after the first newline (NOCBREAK mode). In HALF-DELAY mode, the program will hang until a character is typed or the specified timeout has been reached. Unless **noecho**( ) has been set, the character will also be echoed into the designated window.

When **wgetch**( ) is called, before getting a character, it will call **wrefresh**( ) if anything in the window has changed (for example, the cursor has moved or text changed).

When using **getch**( ), **wgetch**( ), **mvgetch**( ), or **mvwgetch**( ), do not set both NOCBREAK mode (**nocbreak**( )) and ECHO mode (**echo**( )) at the same time. Depending on the state of the *tty*(7) driver when each character is typed, the program may produce undesirable results.

If **wgetch**( ) encounters a ^D, it is returned (unlike *stdio* routines, which would return a null string and have a return code of **-1**).

If **keypad**(win, **TRUE**) has been called, and a function key is pressed, the token for that function key will be returned instead of the raw characters. (See **keypad**() under "Input Options Setting.") Possible function keys are defined in <**curses.h**> with integers beginning with **0401**, whose names begin with **KEY_**. If a character is received that could be the beginning of a function key (such as escape), *curses* will set a timer. If the remainder of the sequence is not received within the designated time, the character will be passed through, otherwise the function key value will be returned. For this reason, on many terminals, there will be a delay after a user presses the escape key before the escape is returned to the program. (Use by a programmer of the escape key for a single character routine is discouraged. Also see **notimeout**() below.)

Note that **getch**(), **mvgetch**(), and **mvwgetch**() are macros.

getstr(str)
wgetstr(win, str)
mvgetstr(y, x, str)
mvwgetstr(win, y, x, str)

A series of calls to **wgetch**() is made, until a newline, carriage return, or enter key is received. The resulting value (except for this terminating character) is placed in the area pointed at by the character pointer *str*. The user's erase and kill characters are interpreted. See **wgetch**() for how it handles characters differently from *stdio* routines (especially ^D).

Note that **getstr**(), **mvgetstr**(), and **mvwgetstr**() are macros.

ungetch(c)

Place *c* onto the input queue, to be returned by the next call to **wgetch**().

flushinp()

Throws away any typeahead that has been typed by the user and has not yet been read by the program. Note that **flushinp**() will not throw away any characters supplied by **ungetch**().

inch()
winch(win)
mvinch(y, x)
mvwinch(win, y, x)

The character, of type **chtype**, at the current position in the named window is returned. If any attributes are set for that position, their values will be OR'ed into the value returned. The predefined constants **A_CHARTEXT** and **A_ATTRIBUTES**, defined in <**curses.h**>, can be used with the C logical AND (&) operator to extract the character or attributes alone.

Note that **inch**(), **winch**(), **mvinch**(), and **mvwinch**() are macros.

scanw(fmt [, arg . . .])
wscanw(win, fmt [, arg . . .])
mvscanw(y, x, fmt [, arg . . .])
mvwscanw(win, y, x, fmt [, arg . . .])

These routines correspond to *scanf*(3S), as do their arguments and return values. **wgetstr**() is called on the window, and the resulting line is used as input for the scan. The return value for these routines is the number of *arg* values that are converted by *fmt*. *arg* values that are not converted are lost. See **wgetstr**() for how it handles strings differently than the *stdio* routines (especially **^D**).

vwscanw(win, fmt, ap)

This routine is similar to **vwprintw**() in that it performs a **wscanw**() using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in <**varargs.h**>. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

Section 5: Output Options Setting

These routines set options within *curses* that deal with output. All options are initially FALSE, unless otherwise stated. It is not necessary to turn these options off before calling **endwin**().

X)

clearok(win, bf)

If enabled (*bf* is **TRUE**), the next call to **wrefresh**( ) with this window will clear the screen completely and redraw the entire screen from scratch. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

idlok(win, bf)

If enabled (*bf* is **TRUE**), *curses* will consider using the hardware "insert/delete-line" feature of terminals so equipped. If disabled (*bf* is FALSE), *curses* will very seldom use this feature. (The "insert/delete-character" feature is always considered.) This option should be enabled only if your application needs "insert/delete-line", for example, for a screen editor. It is disabled by default because "insert/delete-line" tends to be visually annoying when used in applications where it isn't really needed. If "insert/delete-line" cannot be used, *curses* will redraw the changed portions of all lines. Not calling **idlok**( ) saves approximately 5000 bytes of memory.

leaveok(win, bf)

Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

setscrreg(top, bot)
wsetscrreg(win, top, bot)

These routines allow the user to set a software scrolling region in a window. *top* and *bot* are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok**( ) are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. (Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in the DEC VT100. Only the text of the window is scrolled; if **idlok**( ) is enabled and the terminal has either a scrolling region or "insert/delete-line" capability, they will probably be used by the output routines.)

Note that **setscrreg**( ) is a macro.

scrollok(win, bf)

This option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either from a newline on the bottom line, or typing the last character of the last line. If disabled (*bf* is **FALSE**), the cursor is left on the bottom line at the location where the offending character was entered. If enabled (*bf* is **TRUE**), **wrefresh**( ) is called on the window, and then the physical terminal and window are scrolled up one line. (Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call **idlok**( ).)

Note that **scrollok**( ) will always return OK.

Section 6: Input Options Setting

These routines set options within *curses* that deal with input. The options involve using *ioctl*(2) and therefore interact with *curses* routines. It is not necessary to turn these options off before calling **endwin**( ).

cbreak( )
nocbreak( )

These two routines put the terminal into and out of CBREAK mode, respectively. In CBREAK mode, characters typed by the user are immediately available to the program and erase/kill character processing is not performed. When in NOCBREAK mode, the tty driver will buffer characters typed until a newline or carriage return is typed. Interrupt and flow-control characters are unaffected by this mode (see *termio*(7)). Initially the terminal may or may not be in CBREAK mode, as it is inherited, therefore, a program should call **cbreak**( ) or **nocbreak**( ) explicitly. Most interactive programs using *curses* will set CBREAK mode.

Note that **cbreak**( ) performs a subset of the functionality of **raw**( ). See **wgetch**( ) under "Input" for a discussion of how these routines interact with **echo**( ) and **noecho**( ).

echo( )
noecho( ) These routines control whether characters typed by the user are echoed by **wgetch**( ) as they are typed. Echoing by the tty driver is always disabled, but initially **wgetch**( ) is in ECHO mode, so characters typed are echoed. Authors of most interactive programs prefer to do their own echoing in a

controlled area of the screen, or not to echo at all, so they disable echoing by calling **noecho**(). See **wgetch**() under "Input" for a discussion of how these routines interact with **cbreak**() and **nocbreak**().

nl()
nonl() These routines control whether carriage return is translated into newline on input by **wgetch**(). Initially, this translation is done; **nonl**() turns the translation off. Note that translation by the *tty*(7) driver is disabled in CBREAK mode.

halfdelay(tenths)
Half-delay mode is similar to CBREAK mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, **ERR** will be returned if nothing has been typed. *tenths* must be a number between 1 and 255. Use **nocbreak**() to leave half-delay mode.

intrflush(win, bf)
If this option is enabled, when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt, but causing *curses* to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

keypad(win, bf)
This option enables *curses* to obtain information from the keypad of the user's terminal. If enabled, the user can press a function key (such as an arrow key) and **wgetch**() will return a single value representing the function key, as in **KEY_LEFT**. If disabled, *curses* will not treat function keys specially and the program would have to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit), calling **keypad** (*win*, **TRUE**) will turn it on.

meta(win, bf)
Initially, whether the terminal returns 7 or 8 significant bits on input depends on the control mode of the tty driver (see *termio*(7)). To force 8 bits to be returned, invoke **meta** (*win*, **TRUE**). To force 7 bits to be returned, invoke **meta** (*win*,

FALSE). The window argument, *win*, is always ignored. If the *terminfo*(4) capabilities **smm** (meta_on) and **rmm** (meta_off) are defined for the terminal, **smm** will be sent to the terminal when **meta** (*win*, **TRUE**) is called and **rmm** will be sent when **meta** (*win*, **FALSE**) is called.

nodelay(win, bf)
This option causes **wgetch**() to be a non-blocking call. If no input is ready, **wgetch**() will return **ERR**. If disabled, **wgetch**() will hang until a key is pressed.

notimeout(win, bf)
While interpreting an input escape sequence, **wgetch**() will set a timer while waiting for the next character. If **notimeout**(*win*, **TRUE**) is called, then **wgetch**() will not set a timer. The purpose of the timeout is to differentiate between sequences received from a function key and those typed by a user.

raw()
noraw() The terminal is placed into or out of RAW mode. RAW mode is similar to CBREAK mode, in that characters typed are immediately passed through to the user program; however, in RAW mode, the interrupt, quit, suspend, and flow control characters are passed through uninterpreted, instead of generating a signal as they do in CBREAK mode. The behavior of the BREAK key depends on other bits in the *tty*(7) driver that are not set by *curses*.

typeahead(fildes)
curses does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update will be postponed until **wrefresh**() or **doupdate**() is called again. This allows faster response to commands typed in advance. Normally, the file descriptor for the input FILE pointer passed to **newterm**(), or **stdin** in the case that **initscr**() was used, will be used to do this typeahead checking. The **typeahead**() routine specifies that the file descriptor *fildes* is to be used to check for typeahead instead. If *fildes* is **–1**, then no typeahead checking will be done.

Note that *fildes* is a file descriptor, not a **<stdio.h>** FILE pointer.

Section 7: Environment Queries

baudrate()
Returns the output speed of the terminal. The number returned is in bits per second, for example, 9600, and is an integer.

char erasechar()
The user's current erase character is returned.

has_ic() True if the terminal has insert- and delete-character capabilities.

has_il() True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to check to see if it would be appropriate to turn on physical scrolling using **scrollok()** or **idlok()**.

char killchar()
The user's current line-kill character is returned.

char *longname()
This routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to **initscr()** or **newterm()**. The area is overwritten by each call to **newterm()** and is not restored by **set_term()**, so the value should be saved between calls to **newterm()** if **longname()** is going to be used with multiple terminals.

Section 8: Color Manipulation

This section describes the color manipulation routines introduced in this release of *curses*.

start_color()
This routine requires no arguments. It must be called if the user wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after **initscr()**. **start_color()** initializes eight basic colors (black, blue, green, cyan, red, magenta, yellow, and white), and two global variables, **COLORS** and **COLOR_PAIRS** (respectively defining the maximum number of colors and color-pairs the terminal can support). It also restores the terminal's colors to the values they had when the terminal was just turned on.

init_pair(pair, f, b)
This routine changes the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of the first argument must be between 1 and **COLOR_PAIRS-1**. The value of the second and third arguments must be between 0 and **COLORS**-1. If the color-pair was previously initialized, the screen will be refreshed and all occurrences of that color-pair will be changed to the new definition.

init_color(color, r, g, b)
This routine changes the definition of a color. It takes four arguments: the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of the first argument must be between 0 and **COLORS**-1. (See the section COLOR for the default color index.) The last three arguments must each be a value between 0 and 1000. When **init_color**( ) is used, all occurrences of that color on the screen immediately change to the new definition.

has_colors( )
This routine requires no arguments. It returns **TRUE** if the terminal can manipulate colors, **FALSE** otherwise. This routine facilitates writing terminal-independent programs. For example, a programmer can use it to decide whether to use color or some other video attribute.

can_change_color( )
This routine requires no arguments. It returns **TRUE** if the terminal supports colors and can change their definitions, **FALSE** otherwise. This routine facilitates writing terminal-independent programs.

color_content(color, &r, &g, &b)
This routine gives users a way to find the intensity of the red, green, and blue (RGB) components in a color. It requires four arguments: the color number, and three addresses of **shorts** for storing the information about the amounts of red, green, and blue components in the given color. The value of the first argument must be between 0 and **COLORS**-1. The values that will be stored at the addresses pointed to by the last three

arguments will be between 0 (no component) and 1000 (maximum amount of component).

pair_content(pair, &f, &b)

This routine allows users to find out what colors a given color-pair consists of. It requires three arguments: the color-pair number, and two addresses of **shorts** for storing the foreground and the background color numbers. The value of the first argument must be between 1 and **COLOR_PAIRS-1**. The values that will be stored at the addresses pointed to by the second and third arguments will be between 0 and **COLORS**-1.

Section 9: SOFT LABELS

If desired, *curses* will manipulate the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, *curses* can simulate them by taking over the bottom line of **stdscr**, reducing the size of **stdscr** and the variable **LINES**. *curses* standardizes on 8 labels of 8 characters each. If a *curses* program changes the values of the soft labels, it can restore them only to the default settings for that terminal. (Note also that soft labels are shown in reverse video by default.) Therefore, if before calling a *curses* program a user changes the values of the soft labels, those values cannot be reset when the *curses* program terminates.

slk_init(labfmt)

In order to use soft labels, this routine must be called before **initscr**( ) or **newterm**( ) is called. If **initscr**( ) winds up using a line from **stdscr** to emulate the soft labels, then *labfmt* determines how the labels are arranged on the screen. Setting *labfmt* to **0** indicates that the labels are to be arranged in a 3-2-3 arrangement; **1** asks for a 4-4 arrangement.

slk_set(labnum, label, labfmt)

labnum is the label number, from 1 to 8. *label* is the string to be put on the label, up to 8 characters in length. A **NULL** string or a **NULL** pointer will put up a blank label. *labfmt* is one of **0, 1** or **2**, to indicate whether the label is to be left-justified, centered, or right-justified within the label.

slk_refresh( )
slk_noutrefresh( )

These routines correspond to the routines **wrefresh**( ) and **wnoutrefresh**( ). Most applications would use

slk_noutrefresh( ) because a **wrefresh**( ) will most likely soon follow.

char *slk_label(labnum)
The current label for label number *labnum* is returned, in the same format as it was in when it was passed to **slk_set**( ); that is, how it looked prior to being justified according to the *labfmt* argument of **slk_set**( ).

slk_clear( )
The soft labels are cleared from the screen.

slk_restore( )
The soft labels are restored to the screen after a **slk_clear**( ).

slk_touch( )
All of the soft labels are forced to be output the next time a **slk_noutrefresh**( ) is performed.

slk_attron(attrs)
slk_attrset(attrs)
slk_attroff(attrs)
These routines correspond to **attron**( ), **attrset**( ), and **attroff**( ). They will have effect only if soft labels are simulated on the bottom line of the screen.

Section 10: Low-Level *curses* Access

The following routines give low-level access to various *curses* functionality. These routines typically would be used inside library routines.

def_prog_mode( )
def_shell_mode( )
Save the current terminal modes as the "program" (in **curses**) or "shell" (not in **curses**) state for use by the **reset_prog_mode**( ) and **reset_shell_mode**( ) routines. This is done automatically by **initscr**( ).

reset_prog_mode( )
reset_shell_mode( )
Restore the terminal to "program" (in **curses**) or "shell" (out of *curses*) state. These are done automatically by **endwin**( ) and **doupdate**( ) after an **endwin**( ), so they normally would not be called.

resetty( )

savetty() These routines save and restore the state of the terminal modes. **savetty()** saves the current state of the terminal in a buffer and **resetty()** restores the state to what it was at the last call to **savetty()**.

getsyx(y, x)

The current coordinates of the virtual screen cursor are returned in *y* and *x*. If **leaveok()** is currently **TRUE**, then **–1,–1** will be returned. If lines have been removed from the top of the screen using **ripoffline()**, *y* and *x* include these lines; therefore, *y* and *x* should be used only as arguments for **setsyx()**.

Note that **getsyx()** is a macro, so no "**&**" is necessary before the variables *y* and *x*.

setsyx(y, x)

The virtual screen cursor is set to *y*, *x*. If *y* and *x* are both **–1**, then **leaveok()** will be set. The two routines **getsyx()** and **setsyx()** are designed to be used by a library routine which manipulates *curses* windows but does not want to change the current position of the program's cursor. The library routine would call **getsyx()** at the beginning, do its manipulation of its own windows, do a **wnoutrefresh()** on its windows, call **setsyx()**, and then call **doupdate()**.

ripoffline(line, init)

This routine provides access to the same facility that **slk_init()** uses to reduce the size of the screen. **ripoffline()** must be called before **initscr()** or **newterm()** is called. If *line* is positive, a line will be removed from the top of **stdscr**; if negative, a line will be removed from the bottom. When this is done inside **initscr()**, the routine *init*() is called with two arguments: a window pointer to the 1-line window that has been allocated and an integer with the number of columns in the window. Inside this initialization routine, the integer variables **LINES** and **COLS** (defined in **<curses.h>**) are not guaranteed to be accurate and **wrefresh()** or **doupdate()** must not be called. It is allowable to call **wnoutrefresh()** during the initialization routine.

ripoffline() can be called up to five times before calling **initscr()** or **newterm()**.

scr_dump(filename)
The current contents of the virtual screen are written to the file *filename*.

scr_restore(filename)
The virtual screen is set to the contents of *filename*, which must have been written using **scr_dump**( ). **ERR** is returned if the contents of *filename* are not compatible with the current release of *curses* software. The next call to **doupdate**( ) will restore the screen to what it looked like in the dump file.

scr_init(filename)
The contents of *filename* are read in and used to initialize the *curses* data structures about what the terminal currently has on its screen. If the data is determined to be valid, *curses* will base its next update of the screen on this information rather than clearing the screen and starting from scratch. **scr_init**( ) would be used after **initscr**( ) or a *system*(3S) call to share the screen with another process which has done a **scr_dump**( ) after its **endwin**( ) call. The data will be declared invalid if the *terminfo*(4) capability **nrrmc** is true or the time-stamp of the tty is old. Note that **keypad**( ), **meta**( ), **slk_clear**( ), **curs_set**( ), **flash**( ), and **beep**( ) do not affect the contents of the screen, but will make the tty's time-stamp old.

curs_set(visibility)
The cursor state is set to invisible, normal, or very visible for *visibility* equal to **0**, **1** or **2**. If the terminal supports the *visibility* requested, the previous *cursor* state is returned; otherwise, **ERR** is returned.

draino(ms)
Wait until the output has drained enough that it will only take *ms* more milliseconds to drain completely.

garbagedlines(win, begline, numlines)
This routine indicates to *curses* that a screen line is garbaged and should be thrown away before having anything written over the top of it. It could be used for programs such as editors which want a command to redraw just a single line. Such a command could be used in cases where there is a noisy communications line and redrawing the entire screen would be subject to even more communication noise. Just redrawing the

single line gives some semblance of hope that it would show up unblemished. The current location of the window is used to determine which lines are to be redrawn.

napms(ms)
Sleep for *ms* milliseconds.

mvcur(oldrow, oldcol, newrow, newcol)
Low-level cursor motion.

Section 11: Terminfo-Level Manipulations

These low-level routines must be called by programs that need to deal directly with the *terminfo*(4) database to handle certain terminal capabilities, such as programming function keys. For all other functionality, *curses* routines are more suitable and their use is recommended.

Initially, **setupterm**( ) should be called. (Note that **setupterm**( ) is automatically called by **initscr**( ) and **newterm**( ).) This will define the set of terminal-dependent variables defined in the *terminfo*(4) database. The *terminfo*(4) variables **lines** and **columns** (see *terminfo*(4)) are initialized by **setupterm**( ) as follows: if the environment variables **LINES** and **COLUMNS** exist, their values are used. If the above environment variables do not exist and the program is running in a layer (see *layers*(1)), the size of the current layer is used. Otherwise, the values for **lines** and **columns** specified in the *terminfo*(4) database are used.

The header files **<curses.h>** and **<term.h>** should be included, in this order, to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through **tparm**( ) to instantiate them. All *terminfo*(4) strings (including the output of **tparm**( )) should be printed with **tputs**( ) or **putp**( ). Before exiting, **reset_shell_mode**( ) should be called to restore the tty modes. Programs which use cursor addressing should output **enter_ca_mode** upon startup and should output **exit_ca_mode** before exiting (see *terminfo*(4)). (Programs desiring shell escapes should call **reset_shell_mode**( ) and output **exit_ca_mode** before the shell is called and should output **enter_ca_mode** and call **reset_prog_mode**( ) after returning from the shell. Note that this is different from the *curses* routines (see **endwin**( )).

setupterm(term, fildes, errret)
Reads in the *terminfo*(4) database, initializing the *terminfo*(4) structures, but does not set up the output virtualization structures used by *curses*. The terminal type is in the character string *term*: if *term* is NULL, the environment variable **TERM**

will be used. All output is to the file descriptor *fildes*. If *errret* is not NULL, then **setupterm**( ) will return **OK** or **ERR** and store a status value in the integer pointed to by *errret*. A status of **1** in *errret* is normal, **0** means that the terminal could not be found, and **–1** means that the *terminfo*(4) database could not be found. If *errret* is NULL, **setupterm**( ) will print an error message upon finding an error and exit. Thus, the simplest call is **setupterm ((char *)0, 1, (int *)0)**, which uses all the defaults.

The *terminfo*(4) boolean, numeric and string variables are stored in a structure of type **TERMINAL**. After **setupterm**( ) returns successfully, the variable **cur_term** (of type **TERMINAL ***) is initialized with all of the information that the *terminfo*(4) boolean, numeric and string variables refer to. The pointer may be saved before calling **setupterm**( ) again. Further calls to **setupterm**( ) will allocate new space rather than reuse the space pointed to by **cur_term**.

set_curterm(nterm)
nterm is of type **TERMINAL ***. **set_curterm**( ) sets the variable **cur_term** to *nterm*, and makes all of the *terminfo*(4) boolean, numeric and string variables use the values from *nterm*.

del_curterm(oterm)
oterm is of type **TERMINAL ***. **del_curterm**( ) frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as **cur_term**, then references to any of the *terminfo*(4) boolean, numeric and string variables thereafter may refer to invalid memory locations until another **setupterm**( ) has been called.

restartterm(term, fildes, errret)
Similar to **setupterm**( ), except that it is called after restoring memory to a previous state; for example, after a call to **scr_restore**( ). It assumes that the windows and the input and output options are the same as when memory was saved, but the terminal type and baud rate may be different.

char *tparm(str, p_1, p_2, . . ., p_9)
Instantiate the string *str* with parms p_i. A pointer is returned to the result of *str* with the parameters applied.

tputs(str, count, putc)
Apply padding to the string *str* and output it. *str* must be a

terminfo(4) string variable or the return value from **tparm**( ), **tgetstr**( ), **tigetstr**( ) or **tgoto**( ). *count* is the number of lines affected, or **1** if not applicable. *putc* is a *putchar*(3S)-like routine to which the characters are passed, one at a time.

putp(str) A routine that calls **tputs** (*str*, **1**, **putchar**).

vidputs(attrs, putc)
: Output a string that puts the terminal in the video attribute mode *attrs*, which is any combination of the attributes listed below. The characters are passed to the *putchar*(3S)-like routine *putc*( ).

vidattr(attrs)
: Similar to **vidputs**( ), except that it outputs through *putchar*(3S).

The following routines return the value of the capability corresponding to the character string containing the *terminfo*(4) *capname* passed to them. For example, **rc** = **tigetstr("acsc")** causes the value of **acsc** to be returned in **rc**.

tigetflag(capname)
: The value **–1** is returned if *capname* is not a boolean capability. The value **0** is returned if *capname* is not defined for this terminal.

tigetnum(capname)
: The value **–2** is returned if *capname* is not a numeric capability. The value **–1** is returned if *capname* is not defined for this terminal.

tigetstr(capname)
: The value (char *) **–1** is returned if *capname* is not a string capability. A null value is returned if *capname* is not defined for this terminal.

char *boolnames[ ], ***boolcodes**[ ], ***boolfnames**[ ]
char *numnames[ ], ***numcodes**[ ], ***numfnames**[ ]
char *strnames[ ], ***strcodes**[ ], ***strfnames**[ ]
: These null-terminated arrays contain the *capnames*, the *termcap* codes, and the full C names, for each of the *terminfo*(4) variables.

Section 12: Termcap Emulation

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo*(4) database.

tgetent(bp, name)
: Look up *termcap* entry for *name*. The emulation ignores the buffer pointer *bp*.

tgetflag(codename)
: Get the boolean entry for *codename*.

tgetnum(codename)
: Get numeric entry for *codename*.

char ***tgetstr**(codename, area)
: Return the string entry for *codename*. If *area* is not **NULL,** then also store it in the buffer pointed to by *area* and advance *area*. **tputs**( ) should be used to output the returned string.

char ***tgoto**(cap, col, row)
: Instantiate the parameters into the given capability. The output from this routine is to be passed to **tputs**( ).

tputs(str, affcnt, putc)
: See **tputs**( ) above, under "TERMINFO-LEVEL MANIPULATIONS".

Section 13: Miscellaneous

traceoff( )
traceon( ) Turn off and on debugging trace output when using the debug version of the *curses* library, */usr/lib/libdcurses.a*. This facility is available only to customers with a source license.

unctrl(c) This macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the ^X notation. Printing characters are displayed as is.

unctrl( ) is a macro, defined in <**unctrl.h**>, which is automatically included by <**curses.h**>.

char ***keyname**(c)
: A character string corresponding to the key *c* is returned.

filter( ) This routine is one of the few that is to be called before **initscr**( ) or **newterm**( ) is called. It arranges things so that *curses* thinks that there is a 1-line screen. *curses* will not use any terminal capabilities that assume that they know what line on the screen the cursor is on.

Section 14: Use of curscr

The special window **curscr** can be used in only a few routines. If the window argument to **clearok**( ) is **curscr**, the next call to **wrefresh**( ) with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh**( ) is **curscr**, the screen is immediately cleared and repainted from scratch. (This is how most programs would implement a "repaint-screen" routine.) The source window argument to **overlay**( ), **overwrite**( ), and **copywin**( ) may be **curscr**, in which case the current contents of the virtual terminal screen will be accessed.

Section 15: Obsolete Calls

Various routines are provided to maintain compatibility in programs written for older versions of the curses library. These routines are all emulated as indicated below.

crmode( )	Replaced by **cbreak**( ).
fixterm( )	Replaced by **reset_prog_mode**( ).
gettmode( )	A no-op.
nocrmode( )	Replaced by **nocbreak**( ).
resetterm( )	Replaced by **reset_shell_mode**( ).
saveterm( )	Replaced by **def_prog_mode**( ).
setterm( )	Replaced by **setupterm**( ).

ATTRIBUTES

The following video attributes, defined in <**curses.h**>, can be passed to the routines **wattron**( ), **wattroff**( ), and **wattrset**( ), or OR'ed with the characters passed to **waddch**( ).

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_ALTCHARSET	Alternate character set
A_NORMAL	Turn all attributes off, for example: **wattrset (win, A_NORMAL)**
COLOR_PAIR(n)	Color-pair defined in n (note that this is a macro)

The following bit-masks may be AND'ed with characters returned by **winch**( ).

A_CHARTEXT	Extract character
A_ATTRIBUTES	Extract attributes
A_COLOR	Extract color-pair field information

The following macro is the reverse of COLOR_PAIR(n).

PAIR_NUMBER(attrs)	Returns the pair number associated with the COLOR_PAIR(n) attribute (note that this is a macro)

COLORS

In <**curses.h**> the following macros are defined to have the numeric value shown. These are the default colors. *curses* also assumes that color 0 (zero) is the default background color for all terminals.

COLOR_BLACK	0
COLOR_BLUE	1
COLOR_GREEN	2
COLOR_CYAN	3
COLOR_RED	4
COLOR_MAGENTA	5
COLOR_YELLOW	6
COLOR_WHITE	7

FUNCTION KEYS

The following function keys, defined in <**curses.h**>, might be returned by **wgetch**( ) if **keypad**( ) has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or the definition for the key is not present in the *terminfo*(4) database.

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 keys is reser
KEY_F(n)	(KEY_F0+(n))	Formula for f_n.

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
KEY_SRESET	0530	soft (partial) reset
KEY_RESET	0531	reset or hard reset
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left) keypad is arranged like this: A1 up A3 left B2 right C1 down C3
KEY_A1	0534	Upper left of keypad
KEY_A3	0535	Upper right of keypad
KEY_B2	0536	Center of keypad
KEY_C1	0537	Lower left of keypad
KEY_C3	0540	Lower right of keypad
KEY_BTAB	0541	Back tab key
KEY_BEG	0542	beg(inning) key
KEY_CANCEL	0543	cancel key
KEY_CLOSE	0544	close key
KEY_COMMAND	0545	cmd (command) key
KEY_COPY	0546	copy key
KEY_CREATE	0547	create key
KEY_END	0550	end key
KEY_EXIT	0551	exit key
KEY_FIND	0552	find key
KEY_HELP	0553	help key

KEY_MARK	0554	mark key
KEY_MESSAGE	0555	message key
KEY_MOVE	0556	move key
KEY_NEXT	0557	next object key
KEY_OPEN	0560	open key
KEY_OPTIONS	0561	options key
KEY_PREVIOUS	0562	previous object key
KEY_REDO	0563	redo key
KEY_REFERENCE	0564	ref(erence) key
KEY_REFRESH	0565	refresh key
KEY_REPLACE	0566	replace key
KEY_RESTART	0567	restart key
KEY_RESUME	0570	resume key
KEY_SAVE	0571	save key
KEY_SBEG	0572	shifted beginning key
KEY_SCANCEL	0573	shifted cancel key
KEY_SCOMMAND	0574	shifted command key
KEY_SCOPY	0575	shifted copy key
KEY_SCREATE	0576	shifted create key
KEY_SDC	0577	shifted delete char key
KEY_SDL	0600	shifted delete line key
KEY_SELECT	0601	select key
KEY_SEND	0602	shifted end key
KEY_SEOL	0603	shifted clear line key
KEY_SEXIT	0604	shifted exit key
KEY_SFIND	0605	shifted find key
KEY_SHELP	0606	shifted help key
KEY_SHOME	0607	shifted home key
KEY_SIC	0610	shifted input key
KEY_SLEFT	0611	shifted left arrow key
KEY_SMESSAGE	0612	shifted message key
KEY_SMOVE	0613	shifted move key
KEY_SNEXT	0614	shifted next key
KEY_SOPTIONS	0615	shifted options key
KEY_SPREVIOUS	0616	shifted prev key
KEY_SPRINT	0617	shifted print key
KEY_SREDO	0620	shifted redo key
KEY_SREPLACE	0621	shifted replace key
KEY_SRIGHT	0622	shifted right arrow
KEY_SRSUME	0623	shifted resume key

KEY_SSAVE	0624	shifted save key
KEY_SSUSPEND	0625	shifted suspend key
KEY_SUNDO	0626	shifted undo key
KEY_SUSPEND	0627	suspend key
KEY_UNDO	0630	undo key

LINE GRAPHICS

The following variables may be used to add line-drawing characters to the screen with **waddch()**. When defined for the terminal, the variable will have the **A_ALTCHARSET** bit turned on. Otherwise, the default character listed below will be stored in the variable. The names were chosen to be consistent with the DEC VT100 nomenclature.

Name	*Default*	*Glyph Description*
ACS_ULCORNER	+	upper left corner
ACS_LLCORNER	+	lower left corner
ACS_URCORNER	+	upper right corner
ACS_LRCORNER	+	lower right corner
ACS_RTEE	+	right tee (┤)
ACS_LTEE	+	left tee (├)
ACS_BTEE	+	bottom tee (┴)
ACS_TTEE	+	top tee (┬)
ACS_HLINE	–	horizontal line
ACS_VLINE	|	vertical line
ACS_PLUS	+	plus
ACS_S1	–	scan line 1
ACS_S9	_	scan line 9
ACS_DIAMOND	+	diamond
ACS_CKBOARD	:	checker board (stipple)
ACS_DEGREE	'	degree symbol
ACS_PLMINUS	#	plus/minus
ACS_BULLET	o	bullet
ACS_LARROW	<	arrow pointing left
ACS_RARROW	>	arrow pointing right
ACS_DARROW	v	arrow pointing down
ACS_UARROW	^	arrow pointing up
ACS_BOARD	#	board of squares
ACS_LANTERN	#	lantern symbol
ACS_BLOCK	#	solid square block

DIAGNOSTICS

All routines return the integer **OK** upon successful completion and the integer **ERR** upon failure, unless otherwise noted in the preceding routine descriptions.

All macros return the value of their **w** version, except **getsyx**( ), **getyx**( ), **getbegyx**( ), **getmaxyx**( ). For these macros, no useful value is returned.

Routines that return pointers always return **(type *) NULL** on error.

BUGS

Currently typeahead checking is done using a nodelay read followed by an **ungetch**( ) of any character that may have been read. Typeahead checking is done only if **wgetch**( ) has been called at least once. This may change when proper kernel support is available. Programs which use a mixture of their own input routines with *curses* input routines may wish to call **typeahead(–1)** to turn off typeahead checking.

The argument to **napms**( ) is currently rounded up to the nearest second.

draino (ms) only works for *ms* equal to **0**.

WARNINGS

To use the new *curses* features, use the version of *curses* on SYSTEM V/88. All programs that ran with prior releases of *curses* will also run on SYSTEM V/88. You can link applications with object files based on prior releases of *curses/terminfo* with SYSTEM V/88 *libcurses.a* library; however, the opposite is not true.

Between the time a call to **initscr**( ) and **endwin**( ) has been issued, use only the routines in the *curses* library to generate output. Using system calls or the "standard I/O package" (see *stdio*(3S)) for output during that time can cause unpredictable results.

If a pointer passed to a routine as a window argument is null or out of range, the results are undefined (core may be dumped).

SEE ALSO

cc(1), ld(1), tput(1) in the *User's Reference Manual*.
ioctl(2), plot(3X), putc(3S), scanf(3S), stdio(3S), system(3S), vprintf(3S) in the *Programmer's Reference Manual*.
profile(4), term(4), terminfo(4), varargs(5), termio(7), tty(7) in the *System Administrator's Reference Manual*.
curses/terminfo Chapter 10 of the *Programmer's Guide*.

NAME

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

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>

DIR *opendir (*filename*)
char **filename;*

struct dirent *readdir (*dirp*)
DIR **dirp;*

long telldir (*dirp*)
DIR **dirp;*

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

void rewinddir (*dirp*)
DIR *dirp;

void closedir (*dirp*)
DIR **dirp;*

DESCRIPTION

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 or is not a directory, or if it cannot *malloc*(3X) enough memory to hold a DIR structure or a buffer for the directory entries.

readdir returns a pointer to the next active directory entry. No inactive entries are returned. It returns **NULL** upon reaching the end of the directory or upon detecting an invalid location in the directory.

telldir returns the current location associated with the named *directory stream*.

seekdir sets the position of the next *readdir* operation on the *directory stream*. The new position reverts to the one associated with the *directory stream* when the *telldir* operation from which *loc* was obtained was performed. Values returned by *telldir* are good only if the directory has not changed due to compaction or expansion. This is not a problem with System V/88, but it may be with some file system types.

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

closedir closes the named *directory stream* and frees the DIR structure.

The following errors can occur as a result of these operations:

opendir:

[ENOTDIR] A component of *filename* is not a directory.

[EACCES] A component of *filename* denies search permission.

[EMFILE] The maximum number of file descriptors are currently open.

[EFAULT] *filename* points outside the allocated address space.

readdir:

[ENOENT] The current file pointer for the directory is not located at a valid entry.

[EBADF] The file descriptor determined by the DIR stream is no longer valid. This results if the DIR stream has been closed.

telldir, seekdir, and *closedir:*

[EBADF] The file descriptor determined by the DIR stream is no longer valid. This results if the DIR stream has been closed.

EXAMPLE

Sample code that searches a directory for entry *name*:

```
dirp = opendir( "." );
while ( (dp = readdir( dirp )) != NULL )
        if ( strcmp( dp->d_name, name ) == 0 )
                {
                closedir( dirp );
                return FOUND;
                }
closedir( dirp );
return NOT_FOUND;
```

SEE ALSO

getdents(2), dirent(4)

(3

WARNINGS

rewinddir is implemented as a macro, so its function address cannot be taken.

NAME

getnum – calculate an integer value from a string of characters.

SYNOPSIS

int getnum (*string*)
char **string;*

DESCRIPTION

getnum returns the integer value of a character string. *getnum* uses the following rules when calculating a number from a character string:

- Skip over any white space.
- Change a series of the numerical characters (0 - 9) into a number assuming base 10 representation.
- A series of numerical characters may end with **k**, **b**, or **w** to specify multiplication by 1024, 512, or 2 respectively;
- A pair of numbers may be separated by x or * to indicate a product of those two numbers.
- Use the **null** and **colon** as the termination characters.
- If an illegal character is encountered before a *termination* character, an error conditions exists and -1 is returned.

The program must be loaded with the disk access library **libmnt.a**

SEE ALSO

scanf(3S)

NAME

getperms – read the **permissions** file

SYNOPSIS

int getperms (*disk*)
struct usrdev **disk;*

DESCRIPTION

When *getperms* is invoked, the member *alias* is used to find a match in the **permissions** file. When a match of either the *slice* entry or the *alias* entry is found, *getperms* returns the structure filled with the contents of the matching line.

The program must be loaded with the disk access library **libmnt.a**.

```
struct usrdev
{
  char slice[ ];/* real device to access */
  char alias[ ];/* an alternative name for the device */
  char fsize[ ];/* maximum file system size on the device */
  char modes[ ];/* access permissions */
  char mnt_pt[ ];/* the default mount directory */
  char pgm[ ];/* format utitily to envoke */
};
```

FILES

/usr/include/mnt.h

SEE ALSO

filesys(4)

NAME

getspent, getspnam, setspent, endspent, fgetspent, lckpwdf, ulckpwdf – get shadow password file entry

SYNOPSIS

#include <shadow.h>

struct spwd *getspent ( )

struct spwd *getspnam (name)
char *name;

int lckpwdf ( )

int ulckpwdf ( )

void setspent ( )

void endspent ( )

struct spwd *fgetspent (fp)
FILE *fp;

DESCRIPTION

The *getspent* and *getspnam* routines each return a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/shadow** file. Each line in the file contains a "shadow password" structure, declared in the < **shadow.h**> header file:

```
struct spwd{
        char    *sp_namp;
        char    *sp_pwdp;
        long    sp_lstchg;
        long    sp_min;
        long    sp_max;
};
```

The *getspent* routine when first called returns a pointer to the first *spwd* structure in the file; thereafter, it returns a pointer to the next *spwd* structure in the file so successive calls can be used to search the entire file. The *getspnam* routine 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. The *getspent* and *getspnam* routines populate the **sp_min**, **sp_max**, or **sp_lstchg** field with –1 if the corresponding field in **/etc/shadow** is empty. If an end-of-file or an error is encountered on reading, or there is a format error in the file, these functions return a NULL pointer.

/etc/.pwd.lock is the lock file. It is used to coordinate modification access to the password files **/etc/passwd** and **/etc/shadow**. *lckpwdf*( ) and *ulckpwdf*( ) are routines that are used to gain modification access to the password files, through the lock file. A process first uses *lckpwdf*( ) to lock the lock file thereby gaining exclusive rights to modify the **/etc/passwd** or **/etc/shadow** password file. Upon completing modifications, a process should release the lock on the lock file via *ulckpwdf*( ). This mechanism prevents simultaneous modification of the password files.

The *lckpwdf*( ) routine attempts to lock the file **/etc/.pwd.lock**. If file **/etc/.pwd.lock** is already locked, *lckpwdf*( ) tries for 15 seconds to lock the file. If *lckpwdf*( ) is unsuccessful, then *lckpwdf*( ) returns a –1. If *lckpwdf*( ) succeeds to lock the file **/etc/.pwd.lock** within 15 seconds, then a return code other than –1 is returned.

The *ulckpwdf*( ) routine attempts to unlock the file **/etc/.pwd.lock**. If successful, *ulckpwdf*( ) returns a 0. If the unlocking failed, as in the case that file **/etc/.pwd.lock** was not locked initially, then *ulckpwdf*( ) returns a –1.

A call to the *setspent* routine has the effect of rewinding the shadow password file to allow repeated searches. The *endspent* routine may be called to close the shadow password file when processing is complete.

The *fgetspent* routine returns a pointer to the next *spwd* structure in the stream *fp*, which matches the format of **/etc/shadow**.

FILES

/etc/shadow, /etc/passwd, /etc/.pwd.lock

SEE ALSO

putspent(3X)

DIAGNOSTICS

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

WARNING

If a program not otherwise using standard I/O uses this routine, the size of the program will increase more than might be expected.

This routine is for internal use only, compatibility is not guaranteed.

CAVEAT

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

NAME

ldahread – read the archive header of a member of an archive file

SYNOPSIS

```
#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

DESCRIPTION

If **TYPE(***ldptr***)** is the archive file magic number, *ldahread* reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead*.

ldahread returns **SUCCESS** or **FAILURE**. *ldahread* will fail if **TYPE(***ldptr***)** does not represent an archive file, or if it cannot read the archive header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4), ar(4)

NAME

ldclose, ldaclose – close a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldclose (*ldptr*)
LDFILE **ldptr;*

int ldaclose (*ldptr*)
LDFILE **ldptr;*

DESCRIPTION

ldopen(3X) and *ldclose* are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of common object files can be processed as if it were a series of simple common object files.

If **TYPE(***ldptr***)** does not represent an archive file, *ldclose* will close the file and free the memory allocated to the **LDFILE** structure associated with *ldptr*. If **TYPE(***ldptr***)** is the magic number of an archive file, and if there are any more files in the archive, *ldclose* will reinitialize **OFFSET(***ldptr***)** to the file address of the next archive member and return **FAILURE**. The **LDFILE** structure is prepared for a subsequent *ldopen*(3X). In all other cases, *ldclose* returns **SUCCESS**.

ldaclose closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr* regardless of the value of **TYPE(***ldptr***)**. *ldaclose* always returns **SUCCESS**. The function is often used in conjunction with *ldaopen*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fclose(3S), ldopen(3X), ldfcn(4)

NAME

ldfhread – read the file header of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;

DESCRIPTION

ldfhread reads the file header of the common object file currently associated with *ldptr* into the area of memory beginning at *filehead*.

ldfhread returns **SUCCESS** or **FAILURE**. *ldfhread* will fail if it cannot read the file header.

In most cases the use of *ldfhread* can be avoided by using the macro **HEADER(*ldptr*)** defined in **ldfcn.h** [see ldfcn (4)]. The information in any field, *fieldname*, of the file header may be accessed using **HEADER(ldptr).fieldname**.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldgetname – retrieve symbol name for common object file symbol table entry

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

char *ldgetname (*ldptr, symbol*)
LDFILE **ldptr;*
SYMENT **symbol;*

DESCRIPTION

ldgetname returns a pointer to the name associated with symbol as a string. The string is contained in a static buffer local to *ldgetname* that is overwritten by each call to *ldgetname*, and therefore must be copied by the caller if the name is to be saved.

ldgetname can be used to retrieve names from object files without any backward compatibility problems. *ldgetname* will return NULL (defined in **stdio.h**) for an object file if the name cannot be retrieved. This situation can occur: if the "string table" cannot be found, if not enough memory can be allocated for the string table, if the string table appears not to be a string table (for example, if an auxiliary entry is handed to *ldgetname* that looks like a reference to a name in a nonexistent string table), or if the name's offset into the string table is past the end of the string table.

Typically, *ldgetname* will be called immediately after a successful call to *ldtbread* to retrieve the name associated with the symbol table entry filled by *ldtbread*.

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4)

NAME

ldlread, ldlinit, ldlitem – manipulate line number entries of a common object file function

SYNOPSIS

```
#include &lt;stdio.h&gt;
#include &lt;filehdr.h&gt;
#include &lt;linenum.h&gt;
#include &lt;ldfcn.h&gt;

int ldlread (ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO *linent;

int ldlinit (ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem (ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO *linent;
```

DESCRIPTION

ldlread searches the line number entries of the common object file currently associated with *ldptr*. *ldlread* begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, the index of its entry in the object file symbol table. *ldlread* reads the entry with the smallest line number equal to or greater than *linenum* into the memory beginning at *linent*.

ldlinit and *ldlitem* together perform exactly the same function as *ldlread*. After an initial call to *ldlread* or *ldlinit*, *ldlitem* may be used to retrieve a series of line number entries associated with a single function. *ldlinit* simply locates the line number entries for the function identified by *fcnindx*. *ldlitem* finds and reads the entry with the smallest line number equal to or greater than *linenum* into the memory beginning at *linent*.

ldlread, *ldlinit*, and *ldlitem* each return either **SUCCESS** or **FAILURE**. *ldlread* will fail if there are no line number entries in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum*. *ldlinit* will fail if there are no line number entries in the object file or if *fcnindx* does not index a function entry in the symbol table. *ldlitem* will fail if it finds no line number equal to or greater than *linenum*.

The programs must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbindex(3X), ldfcn(4)

NAME

ldlseek, ldnlseek – seek to line number entries of a section of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldlseek (*ldptr, sectindx*)
LDFILE **ldptr;*
unsigned short *sectindx;*

int ldnlseek (*ldptr, sectname*)
LDFILE **ldptr;*
char **sectname;*

DESCRIPTION

ldlseek seeks to the line number entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

ldnlseek seeks to the line number entries of the section specified by *sectname*.

ldlseek and *ldnlseek* return **SUCCESS** or **FAILURE.** *ldlseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnlseek* will fail if there is no section name corresponding with **sectname*. Either function will fail if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4)

NAME

ldohseek – seek to the optional file header of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldohseek (*ldptr*)
LDFILE **ldptr*;

DESCRIPTION

ldohseek seeks to the optional file header of the common object file currently associated with *ldptr*.

ldohseek returns **SUCCESS** or **FAILURE.** *ldohseek* will fail if the object file has no optional header or if it cannot seek to the optional header.

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldfhread(3X), ldfcn(4)

NAME

ldopen, ldaopen – open a common object file for reading

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

LDFILE *ldopen (*filename, ldptr*)
char **filename;*
LDFILE **ldptr;*

LDFILE **ldaopen* (*filename, oldptr*)
char **filename;*
LDFILE **oldptr;*

DESCRIPTION

ldopen and *ldclose*(3X) are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of common object files can be processed as if it were a series of simple common object files.

If *ldptr* has the value **NULL**, *ldopen* will open *filename* and allocate and initialize the **LDFILE** structure, and return a pointer to the structure to the calling program.

If *ldptr* is valid and if **TYPE(***ldptr***)** is the archive magic number, *ldopen* will reinitialize the **LDFILE** structure for the next archive member of *filename*.

ldopen and *ldclose*(3X) are designed to work in concert. *ldclose* will return **FAILURE** only when **TYPE(***ldptr***)** is the archive magic number and there is another file in the archive to be processed. Only then should *ldopen* be called with the current value of *ldptr*. In all other cases, in particular whenever a new *filename* is opened, *ldopen* should be called with a NULL *ldptr* argument.

The following is a prototype for the use of *ldopen* and *ldclose*(3X):

```
/* for each filename to be processed */

ldptr = NULL;
do
{
        if ((ldptr = ldopen(filename, ldptr)) != NULL)
        {
                /* check magic number */
                /* process the file */
        }
} while (ldclose(ldptr) == FAILURE);
```

If the value of *oldptr* is not **NULL**, *ldaopen* will open *filename* anew and allocate and initialize a new **LDFILE** structure, copying the **TYPE**, **OFFSET**, and **HEADER** fields from *oldptr*. *Ldaopen* returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both *ldopen* and *ldaopen* open *filename* for reading. Both functions return **NULL** if *filename* cannot be opened, or if memory for the **LDFILE** structure cannot be allocated. A successful open does not insure that the given file is a common object file or an archived object file.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fopen(3S), ldclose(3X), ldfcn(4)

(3

NAME

ldrseek, ldnrseek – seek to relocation entries of a section of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldrseek (*ldptr, sectindx*)
LDFILE **ldptr;*
unsigned *short sectindx;*

int ldnrseek (*ldptr, sectname*)
LDFILE **ldptr;*
char **sectname;*

DESCRIPTION

ldrseek seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

ldnrseek seeks to the relocation entries of the section specified by *sectname*.

ldrseek and *ldnrseek* return **SUCCESS** or **FAILURE**. *ldrseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnrseek* will fail if there is no section name corresponding with *sectname*. Either function will fail if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4)

NAME

ldshread, ldnshread – read an indexed/named section header of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <ldfcn.h>

int ldshread (*ldptr, sectindx, secthead*)
LDFILE **ldptr;*
unsigned short *sectindx;*
SCNHDR **secthead;*

int ldnshread (*ldptr, sectname, secthead*)
LDFILE **ldptr;*
char **sectname;*
SCNHDR **secthead;*

DESCRIPTION

ldshread reads the section header specified by *sectindx* of the common object file currently associated with *ldptr* into the area of memory beginning at *secthead*.

ldnshread reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

ldshread and *ldnshread* return **SUCCESS** or **FAILURE**. *ldshread* will fail if *sectindx* is greater than the number of sections in the object file; *ldnshread* will fail if there is no section name corresponding with *sectname*. Either function will fail if it cannot read the specified section header.

Note that the first section header has an index of *one*.

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4)

NAME

ldsseek, ldnsseek – seek to an indexed/named section of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldsseek (*ldptr, sectindx*)
LDFILE **ldptr;*
unsigned short *sectindx;*

int ldnsseek (*ldptr, sectname*)
LDFILE **ldptr;*
char **sectname;*

DESCRIPTION

ldsseek seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

ldnsseek seeks to the section specified by *sectname*.

ldsseek and *ldnsseek* return **SUCCESS** or **FAILURE**. *ldsseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnsseek* will fail if there is no section name corresponding with *sectname*. Either function will fail if there is no section data for the specified section or if it cannot seek to the specified section.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4)

NAME

ldtbindex – compute the index of a symbol table entry of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (*ldptr*)
LDFILE ******ldptr;*

DESCRIPTION

ldtbindex returns the (**long**) index of the symbol table entry at the current position of the common object file associated with *ldptr*.

The index returned by *ldtbindex* may be used in subsequent calls to *ldtbread*(3X). However, since *ldtbindex* returns the index of the symbol table entry that begins at the current position of the object file, if *ldtbindex* is called immediately after a particular symbol table entry has been read, it will return the index of the next entry.

ldtbindex will fail if there are no symbols in the object file, or if the object file is not positioned at the beginning of a symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4)

(3

NAME

ldtbread – read an indexed symbol table entry of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (*ldptr, symindex, symbol*)
LDFILE **ldptr;*
long *symindex;*
SYMENT **symbol;*

DESCRIPTION

ldtbread reads the symbol table entry specified by *symindex* of the common object file currently associated with *ldptr* into the area of memory beginning at **symbol**.

ldtbread returns **SUCCESS** or **FAILURE**. *ldtbread* will fail if *symindex* is greater than or equal to the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldgetname(3X), ldfcn(4)

NAME

ldtbseek – seek to the symbol table of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldtbseek (*ldptr*)
LDFILE **ldptr;*

DESCRIPTION

ldtbseek seeks to the symbol table of the common object file currently associated with *ldptr*.

ldtbseek returns **SUCCESS** or **FAILURE**. *ldtbseek* will fail if the symbol table has been stripped from the object file, or if it cannot seek to the symbol table.

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldfcn(4)

NAME

logname – return login name of user

SYNOPSIS

char *logname ( )

DESCRIPTION

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

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

FILES

/etc/profile

SEE ALSO

getenv(3C), profile(4), environ(5)
env(1), login(1) in the *User's Reference Manual*.

CAVEATS

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

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()

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 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 0. 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. 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 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

putspent – write shadow password file entry

SYNOPSIS

#include <shadow.h>

int putspent (*p*, *fp*)
struct *spwd* ******p*;
**FILE ****fp*;

DESCRIPTION

The *putspent* routine is the inverse of *getspent*(3X). Given a pointer to a *spwd* structure created by the *getspent* routine (or the *getspnam* routine), the *putspent* routine writes a line on the stream *fp*, which matches the format of **/etc/shadow**.

If the **sp_min**, **sp_max**, or **sp_lstchg** field of the *spwd* structure is –1, the corresponding **/etc/shadow** field is cleared.

This program must be loaded with the library **libsec.a**.

SEE ALSO

getspent(3X)

DIAGNOSTICS

The *putspent* routine returns non-zero if an error was detected during its operation, otherwise, zero.

WARNING

If a program not otherwise using standard I/O uses this routine, the size of the program will increase more than might be expected.

This routine is for internal use only, compatibility is not guaranteed.

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*;

DESCRIPTION

regcmp compiles a regular expression (consisting of the concatenated arguments) and returns a pointer to the compiled form. *malloc*(3C) is used to create space for the compiled form. 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 meaning in *ed*(1).

$ 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]***.

X)

{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 with a \ (backslash) to be used as themselves.

EXAMPLES

Example 1:

```
char *cursor, *newcursor, *ptr;
        ...
newcursor = regex((ptr = regcmp("^\n", (char *)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", (char *)0);
newcursor = regex(name, "012Testing345", ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (the "4"). 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*.

These routines are kept in **/lib/libPW.a**.

SEE ALSO

regcmp(1), malloc(3C)
ed(1) in the *User's Reference Manual*.

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required.

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;*

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 that uses these functions must be loaded with the object-file access routine library **libld.a**.

NAME

intro – introduction to file formats

DESCRIPTION

This section outlines the formats of various files. The C structure declarations for the file formats are given where applicable. Usually, the header files containing these structure declarations can be found in the directories **/usr/include** or **/usr/include/sys**. For inclusion in C language programs, however, the syntax **#include <filename.h>** or **#include <sys/filename.h>** should be used.

NAME

a.out – common assembler and link editor output

SYNOPSIS

#include <a.out.h>

DESCRIPTION

The file name **a.out** is the default output file name from the link editor *ld*(1). The link editor will make **a.out** executable if there were no errors in linking. The output file of the assembler *as*(1), also follows the common object file format of the **a.out** file although the default file name is different.

A common object file consists of a file header, a SYSTEM V/88 system header (if the file is link editor output), a table of section headers, relocation information, (optional) line numbers, a symbol table, and a string table. The following is the order:

File header.
SYSTEM V/88 system header.
Section 1 header.
...
Section n header.
Section 1 data.
...
Section n data.
Section 1 relocation.
...
Section n relocation.
Section 1 line numbers.
...
Section n line numbers.
Symbol table.
String table.

The last three parts of an object file (line numbers, symbol table, and string table) may be missing if the program was linked with the **–s** option of *ld*(1) or if they were removed by *strip*(1). Also note that the relocation information will be absent after linking unless the **–r** option of *ld*(1) was used. The string table exists only if the symbol table contains symbols with names longer than eight characters.

The sizes of each section (contained in the header, discussed below) are in bytes.

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. On the M88000 Family processors, the text segment starts at location 0x20000.

The **a.out** file produced by *ld*(1) has the magic number 0555 in the first field of the SYSTEM V/88 system header. The headers (file header, SYSTEM V/88 system header, and section headers) are loaded at the beginning of the text segment and the text immediately follows the headers in the user address space. The first text address will equal 0x20000 plus the size of the headers, and varies depending upon the number of section headers in the **a.out** file. (The first 128k of user address space is unused; see *ld(1)*.) In an **a.out** file with three sections (.text, .data, and .bss), the first text address is at 0x200B8 on the M88000 Family processors. The text segment is not writable by the program; if other processes are executing the same **a.out** file, the processes will share a single text segment.

The data segment starts at the next 4Mb boundary past the last text address. The first data address is determined by the following: If an **a.out** file were split into 64K chunks, one of the chunks would contain both the end of text and the beginning of data. When the core image is created, that chunk will appear twice; once at the end of text and once at the beginning of data (with some unused space in between). The duplicated chunk of text that appears at the beginning of data is never executed; it is duplicated so that the operating system may bring in pieces of the file in multiples of the page size without having to realign the beginning of the data section to a page boundary. Therefore, the first data address is the sum of the next segment boundary past the end of text plus the remainder of the last text address divided by 64K. If the last text address is a multiple of 64K no duplication is necessary.

On M88000 Family processors, the stack begins at location 0xF0000000 and grows toward lower addresses. The stack is automatically extended as required. The data segment is extended only as requested by the *brk*(2) system call.

For relocatable files, 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 involves a reference to an undefined external symbol, there will be a relocation entry for the word, the storage class of the symbol-table entry for the symbol will be marked as an "external symbol", and the value and section number of the symbol-table entry will be undefined. 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.

File Header

The format of the **filehdr** header is:

```
struct filehdr
{
        unsigned short  f_magic;     /* magic number */
        unsigned short  f_nscns;     /* number of sections */
        long            f_timdat;    /* time and date stamp */
        long            f_symptr;    /* file ptr to symtab */
        long            f_nsyms;     /* # symtab entries */
        unsigned short  f_opthdr;    /* sizeof(opt hdr) */
        unsigned short  f_flags;     /* flags */
};
```

SYSTEM V/88 System Header

The format of the SYSTEM V/88 system header is:

```
typedef struct aouthdr
{
        short  magic;               /* magic number */
        unsigned int  tsize;        /* text size in bytes, padded */
        unsigned int  dsize;        /* initialized data (.data) */
        unsigned int  bsize;        /* uninitialized data (.bss) */
        unsigned int  nsyms;        /* size of symbol table */
        unsigned int  entry;        /* entry point */
} AOUTHDR;
```

Section Header

The format of the section header is:

```
struct scnhdr
{
        char            s_name[8];   /* section name */
        long            s_paddr;     /* physical address */
        long            s_vaddr;     /* virtual address */
        long            s_size;      /* section size */
        long            s_scnptr;    /* file ptr to raw data */
        long            s_relptr;    /* file ptr to relocation */
```

```
    long            s_lnnoptr;  /* file ptr to line numbers */
    long            s_nreloc;   /* # reloc entries */
    long            s_nlnno;    /* # line number entries */
    long            s_flags;    /* flags */
};
```

Relocation

Object files have one relocation entry for each relocatable reference in the text or data. If relocation information is present, it will be in the following format:

```
struct reloc
{
    long            r_vaddr;    /* (virtual) address of reference */
    long            r_symndx;   /* index into symbol table */
    ushort          r_type;     /* relocation type */
    char            r_pad1;     /* Pad to 4 byte multiple */
    char            r_pad2;     /* Pad to 4 byte multiple */
};
```

The start of the relocation information is *s_relptr* from the section header. If there is no relocation information, *s_relptr* is 0.

Symbol Table

The format of each symbol in the symbol table is:

```
#define         SYMNMLEN 8
#define         FILNMLEN 14
#define         DIMNUM   4

struct syment
{
 union                  /* all ways to get a symbol name */
  {
    char           _n_name[SYMNMLEN];  /* name of symbol */
    struct
    {
      long         _n_zeroes;  /* == OL if in string table */
      long         _n_offset;  /* location in string table */
    } _n_n;
    char           *_n_nptr[2];  /* allows overlaying */
  } _n;
  unsigned long  n_value;   /* value of symbol */
  short          n_scnum;   /* section number */
  unsigned short n_type;   /* type and derived type */
  char           n_sclass;   /* storage class */
  char           n_numaux;   /* number of aux entries */
  char           n_pad1;   /* Pad to 4 byte multiple */
  char           n_pad2;   /* Pad to 4 byte multiple */
};

#define  n_name   _n._n_name
#define  n_zeroes _n._n_n._n_zeroes
#define  n_offset _n._n_n._n_offset
#define  n_nptr   _n._n_nptr[1]
```

Some symbols require more information than a single entry; they are followed by *auxiliary entries* that are the same size as a symbol entry. The format follows.

```
union auxent {
   struct {
                  long    x_tagndx;
                  union {
                      struct {
                           unsigned short x_lnno;
                           unsigned short x_size;
                      } x_lnsz;
                      long    x_fsize;
                  } x_misc;
                  union {
                      struct {
                              long    x_lnnoptr;
                              long    x_endndx;
                      }  x_fcn;
                      struct {
                              unsigned short x_dimen[DIMNUM];
                      } x_ary;
                  } x_fcnary;
                  unsigned short  x_tvndx;
                  char    x_pad1; /* Pad to 4 byte multiple */
                  char    x_pad2; /* Pad to 4 byte multiple */
                  } x_sym;
    struct {
                  char    x_fname[FILNMLEN];
    } x_file;
    struct {
                  long  x_scnlen;
                  unsigned short  x_nreloc;
                  unsigned short  x_nlinno;
    } x_scn;
    struct {
                  long                              x_tvfill;
                  unsigned short  x_tvlen;
                  unsigned short  x_tvran[2];
    } x_tv;
};
```

Indexes of symbol table entries begin at *zero*. The start of the symbol table is *f_symptr* (from the file header) bytes from the beginning of the file. If the symbol table is stripped, *f_symptr* is 0. The string table (if one exists) begins at *f_symptr* + (*f_nsyms* ∗ SYMESZ) bytes from the beginning of the file.

SEE ALSO

as(1), cc(1), ld(1), brk(2), filehdr(4), ldfcn(4), linenum(4), reloc(4), scnhdr(4), syms(4).

NAME

acct – per-process accounting file format

SYNOPSIS

#include <sys/acct.h>

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 */
};

extern   struct   acct     acctbuf;
extern   struct   inode    *acctp;  /* inode of accounting file *,

#define   AFORK 01     /* has executed fork, but no exec */
#define   ASU   02     /* used super-user privileges */
#define   ACCTF        /* 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)

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 acct, 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 */
      unsigned short  ta_fee;        /* fee for special services */
};
```

SEE ALSO

acct(2), exec(2), fork(2) in the *Programmer's Reference Manual.*
acct(1M) in the *System Administrator's Reference Manual.*
acctcom(1) in the *User's Reference Manual.*

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

pathalias – alias file for FACE

DESCRIPTION

The pathalias files contain lines of the form "alias=path" where "path" can be one or more colon (:) separated directories. Whenever a FACE user references a path not beginning with a "/", this file is checked. If the first component of the pathname matches the left-hand side of the equals sign, the right-hand side is searched much like $PATH variable in the UNIX System. This allows users to reference the folder "$HOME/FILECABINET" by typing "filecabinet".

There is a system-wide pathalias file called $VMSYS/pathalias, and each user can also have local alias file called $HOME/pref/pathalias. Settings in the user alias file override settings in the system-wide file. The system-wide file is shipped with several standard FACE aliases, such as filecabinet, wastebasket, preferences, other_users, etc.

NOTES

Unlike command keywords, partial matching of a path alias is not permitted, however, path aliases are case insensitive. The name of an alias should be alphabetic, and in no case can it contain special characters like "/", " or "=". There is no particular limit on the number of aliases allowed. Alias files are read once, at login, and are held in core until logout. Thus, if an alias file is modified during a session, the change will not take effect until the next session.

FILES

$HOME/pref/pathalias
$VMSYS/pathalias

NAME

ar – common archive file format

SYNOPSIS

#include <ar.h>

DESCRIPTION

The archive command *ar*(1) is used to combine several files into one. 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 common object files (see *a.out*(4)) 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. Conversion tools such as *convert*(1) exist to aid in the transportation of 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 file are:

- The number of symbols. Length: 4 bytes.
- The array of offsets into the archive file. Length: 4 bytes * "the number of symbols".
- The name string table. Length: *ar_size* – (4 bytes * ("the number of symbols" + 1)).

The number of symbols and the array of offsets are managed with *sgetl* and *sputl*. The string table contains exactly as many null terminated strings as there are elements in the offsets array. Each offset from the array is associated with the corresponding name from the string table (in order). The names in the string table are all the defined global symbols found in the common object files in the archive. Each offset is the location of the archive header for the associated symbol.

SEE ALSO

ar(1), ld(1), strip(1), sputl(3X), a.out(4)

WARNINGS

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

cftime – language specific strings for converting times and dates to ASCII

DESCRIPTION

The programmer can create one printable file per language. These files must be kept in a special directory **/lib/cftime**. If this directory does not exist, the programmer should create it. The contents of these files are:

- abbreviated month names (in order)
- month names (in order)
- abbreviated weekday names (in order)
- weekday names (in order)
- default strings that specify formats for local time (**%x**) and local date (**%X**).
- default format for cftime, if the argument for cftime is zero or null.
- AM (ante meridiem) string
- PM (post meridiem) string

Each string is on a line by itself. All white space is significant. The order of the strings in the above list is the same order in which the strings appear in the file shown below.

EXAMPLE

/lib/cftime/usa_english

```
Jan
Feb
 ...
January
February
 ...
Sun
Mon
 ...
Sunday
Monday
 ...
%H:%M:%S
%m/%d/%y
%a %b %d %T %Z %Y
AM
PM
```

FILES

/lib/cftime – directory that contains the language specific printable files (create it if it does not exist)

SEE ALSO

ctime(3C) in the *Programmer's Reference Manual.*

NAME

checklist – list of file systems processed by fsck and ncheck

DESCRIPTION

checklist resides in directory **/etc** and contains a list of *specialfile* names. Each *specialfile* name is contained on a separate line and corresponds to a file system. Each file system will then be automatically processed by the *fsck*(1M) command.

FILES

/etc/checklist

SEE ALSO

fsck(1M), ncheck(1M) in the *System Administrator's Reference Manual.*

NAME

core – format of core image file

DESCRIPTION

The system writes out a core image of a terminated process when any of various errors occur. *signal*(2) describes reasons for errors. The most common errors are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called **core** and is written in the working directory of the process (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 copy of the system's per-user data for the process, including the registers as they were at the time of the fault. The size of this section depends on the parameter *usize*, which is defined in **/usr/include/sys/param.h**. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is read-only and shared, or separated from data space, it is not dumped.

The format of the information in the first section is described by the user structure of the system, defined in **/usr/include/sys/user.h**. The locations of the registers are outlined in **/usr/include/sys/reg.h**.

SEE ALSO

crash(1M),sdb(1),setuid(2),signal(2)

NAME

cpio – format of cpio archive

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,"%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%11lo%s",
   &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,Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file are recorded in an element of the array of varying length structures, *archive*, 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!!!. Special files, directories, and the trailer are recorded with *h_filesize* equal to zero.

SEE ALSO

stat(2)
cpio(1), find(1) in the *User's Reference Manual*.

NAME

dfile – device information file

DESCRIPTION

The description file, **dfile**, contains device information for the user's system. The file is divided into three parts. The first part contains physical device specifications. The second part contains system-dependent information. The third part contains microprocessor-specific information. The first two parts are required; the third part is optional. A line with an asterisk (*) in column 1 is a comment. Any kernel can be used to generate the **dfile** used to configure that kernel. Refer to the utility *sysdef*(1M).

FIRST PART OF dfile

Each line contains four or five fields, delimited by blanks and/or tabs in the following format:

devname vector address bus number

The first field, *devname*, is the name of the device as it appears in the **/etc/master** device table. The device name is Field 1 of Part 1 and has a maximum of eight characters (refer to *master*(4). The second field, *vector*, is the interrupt vector location (hexadecimal), which can be calculated as the vector number times 4; this value is also used in the interrupt vector array created by setting the 004000 bit of Field 4 in the *master*(4) file. The third field, *address*, is the device address (hexadecimal); the array for device addresses is automatically created (e.g., vm323_addr[ ]). The fourth field, *bus*, is the bus request level, or interrupt level (1 through 7), and is used in the interrupt level array (e.g., vm323_ilev[ ]) that is created by the 001000 bit in Field 4 of *master*. The fifth field, *number*, is the number (decimal) of devices associated with the corresponding controller; *number* is optional, and if omitted, a default value which is the maximum value for that controller is used. This field is the same as Field 9 in Part 1 of the *master*(4) file and overrides the *master* field if specified in **dfile.**

There are certain drivers which may be provided with the system that are actually pseudo-device drivers; that is, there is no real hardware associated with the driver. Drivers of this type are identified on their respective manual entries. When these devices are specified in the description file, the interrupt *vector*, device *address*, and *bus* request level must all be zero.

SECOND PART OF dfile

The second part contains three different types of lines. Note that all specifications of this part are required, although their order is arbitrary.

1. *Root/pipe/dump device specification*

   Three lines of three fields each:

   root *devnameminor* [,*minor*]...
   pipe *devnameminor* [,*minor*]...
   dump *devnameminor* [,*minor*]...

   where *minor* is the minor device number (in octal). For certain Motorola Inc. disk controllers, it is possible to have a single operating system capable of executing on any device on the controller. For such devices, *minor* can be repeated (separated by commas). The first reference to *minor* specifies the **root** (**pipe**, **dump**) to be used for disk 0, the second *minor* for disk 1, etc. The same number of *minor* references must be present for **root**, **pipe**, **dump**, and **swap**. Currently, eight *minor* numbers may be specified, with the restriction that they must fit on the 100-character line given for each of **root**, **pipe**, **dump**, and **swap**.

2. *Swap device specification*

   One line that contains five fields as follows:

   swap *devnameminor* *swplo* *nswap* [,*minor swplo nswap*]...

   where *swplo* is the lowest disk block (decimal) in the swap area and *nswap* is the number of disk blocks (decimal) in the swap area. Multiple *minor*, *swplo*, and *nswap* specifications can be given; refer to the restrictions described above for multiple *minor* specifications.

3. *Parameter specification*

   Several lines of two fields each as follows (number is decimal):

buffers	number
inodes	number
files	number
mounts	number
coremap	number
swapmap	number
calls	number
procs	number
maxproc	number
texts	number

clists	number
hashbuf	number
physbuf	number
power	0 or 1
mesg	0 or 1
sema	0 or 1
shmem	0 or 1

THIRD PART OF dfile

The third part contains lines identified by a keyword. The format of each line differs for each keyword. The ordering of the third part is significant.

1. *Non-unique driver specifications*

   Several lines of two fields:

   force identifier

   where *identifier* is the name of a unique identifier defined within a driver, located in the kernel I/O library file. This forces the correct linking of non-table driven drivers, such as those for the clock, console, and MMU.

2. *Memory probe specifications*

   Several lines of three fields:

   probe *address* *value*

   where *address* is the hexadecimal number specifying a memory-mapped I/O location that must be reset for the operating system to execute properly. The intent is to provide a means by which non-standard (or unsupported) devices can be set to a harmless state. *Value* is a hexadecimal number (0x00-0xff) to be written in *address*, or **-1**, indicating that the address is to be "read only".

3. *Alien handler entry specifications*

   Several lines of three fields:

   alien *vector_address* *alien_address*

   where *vector_address* is the hexadecimal address of the normal exception vector for the alien entry point, and *alien_address* is the hexadecimal entry point for the handler. If no handler is associated with the *vector_address*, then *alien_address* is entered into the vector. Otherwise, code is produced in **low.s** (for the 68K) or in **conf.c** (for the 88K) so that the alien handler is entered only when the exception occurs in the processor's supervisor state (refer to *config*(1M)).

4. *Multiple handler specifications*

   Several lines of four or five fields:

   dup *flag* *vector_address* *handler* [*argument*]

   where *flag* is a bit mask. The bits are interpreted as:

   1 - if *handler* returns 0, go to the normal interrupt return point (''intret'').

   2 - if *handler* returns 0, go to the normal trap return point (''alltraps'').

   4 - if *handler* returns 0, go to the branch equal return point (''beq return'').

   10 - *argument* is to be passed to *handler*.

   Vector_address is the hexadecimal address of the exception vector. *Handler* is the name of an exception handling routine, with the optional *argument* passed to it. The intent is to provide a means of specifying multiple handlers for a single exception. These handlers are called in the order given in *dfile*(4); then the normal handler is called. If bits 1, 2 or 4 of *flag* are set and the handler returns zero, then the remainder of the handlers are not called.

5. *Memory configuration specifications*

Several lines of four or five fields:

ram *flag* *low* *high* [*size*]

where *flag* is an octal bit mask, which is interpreted as follows:

1 - memory has no parity check and, therefore, need not be initialized after power up.

2 - a single memory block may exist, ranging from *low* through *high*–1.

4 - multiple memory blocks may be located in the range and are of *size* bytes.

10 - private memory will not be used for general purpose ram.

20 - cache inhibited memory will not be cached like general purpose ram.

Low and *high* are hexadecimal memory addresses, and *size* is a hexadecimal number. The intent is to provide information to the operating system about noncontiguous memory. *Low* specifies the low memory address where memory may be located, and which may extend through *high*–1. If the range consists of multiple boards, which may or may not be present, they are of *size* bytes.

For flag 2 ranges, the operating system writes sequential memory locations, starting at low, until a memory fault occurs. For flag 4 ranges, the operating system performs a test for each *size*-sized subrange. If memory need not be initialized, only the first byte of the range (flag 2) or subrange (flag 4) is tested to determine the presence of the memory.

It is essential that *ram* lines be ordered in ascending *low* addresses.

If no **ram** specifier is present, the default is:

ram **2** **0** **F00000**

6. *Header file specifications*

Several lines of two fields:

include *include_file*

where *include_file* is the name of a file to be inserted into the C program, *conf.c,* at the time it is generated by *config* (config.68(1M)). It is inserted after all pre-generated *#include* text, creating a line of the form:

#include include_file

Because the line is inserted exactly as typed, bracketing characters (such as " " and < >) must be a part of the string.

For example:

include < sys/space/newdevspace.h >

SEE ALSO

master(4), config(1M), sysgen(1M).

NAME

dir – format of directories

SYNOPSIS

#include <sys/dir.h>

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*(4)]. The structure of a directory entry as given in the include file is:

```
#ifndef    DIRSIZ
#define    DIRSIZ14
#endif
structdirect
{
        ushort    d_ino;
        char d_name[DIRSIZ];
};
```

By convention, the first two entries in each directory are for "." and "..". 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 ".." has the same meaning as ".".

SEE ALSO

fs(4).

NAME

dirent – file system independent directory entry

SYNOPSIS

#include <sys/dirent.h>
#include <sys/types.h>

DESCRIPTION

Different file system types may have different directory entries. The *dirent* structure defines a file system independent directory entry, which contains information common to directory entries in different file system types. A set of these structures is returned by the *getdents*(2) system call.

The *dirent* structure is defined below.

```
struct dirent {
        ino_t              d_ino;
        off_t              d_off;
        unsigned short     d_reclen;
        char               d_name[1];
    };
```

The *d_ino* is a number which is unique for each file in the file system. The field *d_off* is the offset of that directory entry in the actual file system directory. The field *d_name* is the beginning of the character array giving the name of the directory entry. This name is **NULL** terminated and may have at most MAXNAMLEN characters. This results in file system independent directory entries being variable length entities. The value of *d_reclen* is the record length of this entry. This length is defined to be the number of bytes between the current entry and the next one, so that it will always result in the next entry being on a long boundary.

FILES

/usr/include/sys/dirent.h

SEE ALSO

getdents(2)

NAME

.environ – system-wide FACE environment variables
.pref - default preferences for WASTEBASKET and FILECABINET
.variables

DESCRIPTION

The .environ, .pref, and .variables files contain variables that indicate user preferences for a variety of operations. The .environ and .variables files are located under the user's $HOME/pref directory. The .pref files are found under $HOME/FILECABINET, $HOME/WASTEBASKET, and any directory were preferences were set via the *organize* command. Names and descriptions for each variable are presented below. Variables are listed one per line and are of the form "variable=value".

Variables found in .environ include:

LOGINWIN1 - Windows that are opened when FACE is initialized
...
LOGINWIN4

SORTMODE - Sort mode for file folder listings.
Values include the following hexadecimal digits:
1 sorted alphabetically by name
2 files most recently modified first
800 sorted alphabetically by object type

The values above may be listed in reverse order by "ORing" the following value:
1000 list objects in reverse order

For example, a value of 1002 will produce a folder listing with files LEAST recently modified displayed first. A value of 1001 would produce a "reverse" alphabetical by name listing of the folder

DISPLAYMODE - Display mode for file folders
Values include the following hexadecimal digits:
0 file names only
4 file names and brief description
8 file names, description, plus additional information

WASTEPROMPT - Prompt before emptying wastebasket (yes/no)?
WASTEDAYS - # days before emptying wastebasket
PRINCMD1 - print command defined to print files.
...
PRINCMD3
UMASK - holds default permissions that files will be created with.

Variables found in .pref are SORTMODE and DISPMODE, which have the same values as the SORTMODE and DISPLAYMODE variables described in .environ above.

Variables found in .variables include:

EDITOR - Default editor
PS1 - UNIX shell prompt

FILES

$HOME/pref/.environ
$HOME/pref/.variables
$HOME/FILECABINET/.pref
$HOME/WASTEBASKET/.pref

NAME

errfile – error-log file format

DESCRIPTION

When hardware errors are detected by the system, an error record is generated and passed to the error-logging daemon for recording in the error log for later analysis. The default error log is **/usr/adm/errfile**.

The format of an error record depends on the type of error that was encountered. Every record, however, has a header with the following format:

```
struct errhdr {
 short      e_type;    /* record type */
  short      e_len;     /* bytes in record (inc hdr) */
  time_t    e_time;     /* time of day */
};
```

The permissible record types are as follows:

```
#define E_GOTS       010    /* start */
#define E_GORT       011    /* start for RT */
#define E_STOP       012    /* stop */
#define E_TCHG       013    /* time change */
#define E_CCHG       014    /* configuration change */
#define E_BLK        020    /* block device error */
#define E_STRAY       030   /* stray interrupt */
#define E_PRTY       031    /* memory parity */
```

Some records in the error file are of an administrative nature. These include the startup record that is entered into the file when logging is activated, the stop record that is written if the daemon is terminated "gracefully", and the time-change record that is used to account for changes in the system's time-of-day. These records have the following formats:

```
struct  estart
                short               e_cpu; /* CPU type */
                 struct utsname     e_name; /* system names */
};
#define eend errhdr          /* record header */
struct   etimchg {
         time_t        e_ntime;    /* new time */
};
```

Stray interrupts cause a record with the following format to be logged:

```
struct estray {
        uint e_saddr;   /* stray loc or device addr */
};
```

Generation of memory subsystem errors is not supported in this release.

Error records for block devices have the following format:

```
struct eblock {
dev_t       e_dev;              /* "true" major + minor dev no */
physadr     e_regloc;           /* controller address */
short       e_bacty;            /* other block I/O activity */
struct iostat {
   long     io_ops;             /* number read/writes */
   long     io_misc;            /* number "other" operations */
   ushort   io_unlog;           /* number unlogged errors */
 }          e_stats;
short       e_bflags;           /* read/write, error, etc */
short       e_cyloff;           /* logical dev start cyl */
dadd        r_te_bnum;          /* logical block number */
ushort      e_bytes;            /* number bytes to transfer */
padd        r_te_memadd;        /* buffer memory address */
ushort      e_rtry;             /* number retries */
short       e_nreg;             /* number device registers */
};
```

The following values are used in the *e_bflags* word:

```
#define E_WRITE     0     /* write operation */
#define E_READ      1     /* read operation */
#define E_NOIO      02    /* no I/O pending */
#define E_PHYS      04    /* physical I/O */
#define E_FORMAT    010   /* Formatting Disk*/
#define E_ERROR     020   /* I/O failed */
```

SEE ALSO

errdemon(1M)

NAME

filehdr – file header for common object files

SYNOPSIS

#include <filehdr.h>

DESCRIPTION

Every common object file begins with a 20-byte header. The following C **struct** declaration is used:

```
struct filehdr
{
        unsigned short   f_magic ;    /* magic number */
        unsigned short   f_nscns ;    /* number of sections */
        long             f_timdat ;   /* time & date stamp */
        long             f_symptr ;   /* file ptr to symtab */
        long             f_nsyms ;    /* # symtab entries */
        unsigned short   f_opthdr ;   /* sizeof(opt hdr) */
        unsigned short   f_flags ;    /* flags */
} ;
```

f_symptr is the byte offset into the file at which the symbol table can be found. Its value can be used as the offset in *fseek*(3S) to position an I/O stream to the symbol table. The system optional header is 28-bytes. The valid magic numbers are:

```
#define  FBOMAGIC     0560  /* 3B2 and 3B5 computers */
#define  N3BMAGIC     0550  /* 3B20 computer */
#define  NTVMAGIC     0551  /* 3B20 computer */

#define  VAXWRMAGIC   0570  /* VAX writable text segments */
#define  VAXROMAGIC   0575  /* VAX read only sharable
                               text segments */
```

The value in *f_timdat* is obtained from the *time*(2) system call. Flag bits currently defined are:

```
#define F_RELFLG   0000001 /* relocation entries stripped */
#define F_EXEC     0000002 /* file is executable */
#define F_LNNO     0000004 /* line numbers stripped */
#define F_LSYMS    0000010 /* local symbols stripped */
#define F_MINMAL   0000020 /* minimal object file */
#define F_UPDATE   0000040 /* update file, ogen produced */
#define F_SWABD    0000100 /* file is "pre-swabbed" */
#define F_AR16WR   0000200 /* 16-bit DEC host */
#define F_AR32WR   0000400 /* 32-bit DEC host */
#define F_AR32W    0001000 /* non-DEC host */
```

```
#define F_PATCH   0002000 /* "patch" list in opt hdr */
#define F_BM32ID  0160000 /* WE32000 family ID field */
#define F_BM32B   0020000 /* file contains WE 32100 code */
#define F_BM32MAU 0040000 /* file reqs MAU to execute */
#define F_BM32RST 0010000 /* this object file contains restore
                             work around [3B5/3B2 only] */
```

SEE ALSO

time(2), fseek(3S), a.out(4)

NAME

filesys – permissions file used by the value-added disk access utilities

DESCRIPTION

The file **/etc/filesys** contains information used by the value-added disk access utilities to determine if a user has access permission to certain disks.

Each entry has the following format:

slice *alias* *fsize* *perms* *mnt_pt* *'format_pgm'*

The fields are:

slice
: This is the block device to be accessed by value-added disk access utilities. Some of these utilities, such as *dcpy*(1M) may actually use the raw device. The utility *fmt*(1) uses the raw device slice 7.

alias
: This is a nickname for the entry. When a user asks to access a specific device, the *slice* or the *alias* may be requested. Note that if a user does not specify a device, the first line with the *alias* of **floppy** will be used.

fsize
: The maximum and/or default size of a file system on this device as created by *fs*(1). This field may contain a ':' separated subfield which is the number of inodes to allocate (see *mkfs*(1M)).

perms
: The permissions field actually contains two subfields. The first subfield is optional and is used only for the *tt*(1) command. This field is the largest amount of data that may be transferred to or from the disk. Note that this number may actually be larger than the disk capacity, to allow a larger and therefore faster block size to be used in the transfer. The size is specified as a number of bytes. A number may end with **k**, **b**, or **w** to specify multiplication by 1024, 512, or 2, respectively; a pair of numbers may be separated by **x** to indicate a product. If the first subfield is present, the semicolon character is used to delimit the first and second subfields.

The second subfield specifies which actions are allowed, for each specific disk. Note that if a flag is uppercase as shown in the following table), any user has permission; if a flag is lowercase, only the superuser may execute.

Flag	Permissible Action
M	Make a file system.
R	The disk may be mounted read only, or read from.
W	The disk may be mounted read/write, or written to.
F	The disk may be formatted.
C	Check a file system.

slices may be grouped for mounting and unmounting (using *mnt*(1) and *umnt*(1)) by specifying a set identifier in the permissions field for the desired entries. Valid set names are: **a, A, b, B, 1, 2,** and **3**. Note that sets **A, B, 1, 2,** and **3** may be accessed by any user and that sets **a** and **b** are accessible only to the superuser. In adddition, sets **a** and **b** are defined to include sets **A** and **B**, respectively. Care should be taken not to specify a numeric set identifier immediately following the format **(F)** *perms* flag.

mnt_pt
: The mount directory used when no directory is specified on the command line.

format_pgm
: This field, containing a utility name and options, is combined with the options given to *fmt*(1) and passed on to the shell to be executed. *fmt*(1) uses the raw device slice 7 unless the format slice is specified. An entry of NONE will prohibit formatting.

EXAMPLE

```
#slice    alias  fsize   perms mnt_pt        'format_pgm'

#       Floppy drives: 5.25 in on the MVME327

m327_d70s0 floppy 1264 RWMF /mnt  '/etc/dinit -b /stand/m88k/boots/vmeboot -f m327dsdd5'
m327_s70s0 pcfl   2370 RWMF /mnt  '/etc/dinit -b /stand/m88k/boots/vmeboot -f m327pcat'
```

FILES

/etc/filesys permissions file

SEE ALSO

dcpy(1M), fmt(1), fs(1), mnt(1), getnum(3X), getperms(3X), real(1), tt(1)

NAME

fs: file system – format of system volume

SYNOPSIS

#include <sys/fs/s5filsys.h>
#include <sys/types.h>
#include <sys/s5param.h>

DESCRIPTION

Every file system storage volume has a common format for certain vital information. Every such volume is divided into a certain number of 512-byte long sectors. Sector 0 is unused and is available to contain a bootstrap program or other information.

Sector 1 is the *super-block*. The format of a super-block is:

```
struct  filsys
{
        ushort     s_isize;              /* size in blocks of i-list */
        daddr_t    s_fsize;              /* size in blocks of entire volume */
        short      s_nfree;              /* number of addresses in s_free */
        daddr_t    s_free[NICFREE];      /* free block list */
        short      s_ninode;             /* number of i-nodes in s_inode */
        ushort     s_inode[NICINOD];     /* free i-node list */
        char       s_flock;              /* lock during free list manipulation
        char       s_ilock;              /* lock during i-list manipulation */
        char       s_fmod;               /* super block modified flag */
        char       s_ronly;              /* mounted read-only flag */
        time_t     s_time;               /* last super block update */
        short      s_dinfo[4];           /* device information */
        daddr_t    s_tfree;              /* total free blocks*/
        ushort     s_tinode;             /* total free i-nodes */
        char       s_fname[6];           /* file system name */
        char       s_fpack[6];           /* file system pack name */
        long       s_fill[14];           /* ADJUST to make sizeof filsys
                                         be 512 */
        long       s_state;              /* file system state */
        long       s_magic;              /* magic number to denote new
                                         file system */
        long       s_type;               /* type of new file system */
};
```

```
#define     FsMAGIC     0xfd187e20     /* s_magic number */

#define     Fs1b        1              /* 512-byte block (no longer support
#define     Fs2b        2              /* 1024-byte block (option) */
#define     Fs4b        3              /* 2048-byte block (option) */
#define     Fs8b        4              /* 4096-byte block (default) */
#define     Fs16b       5              /* 8192-byte block (option)*/
#define     FsOKAY      0x7c269d38     /* s_state: clean */
#define     FsACTIVE    0x5e72d81a     /* s_state: active */
#define     FsBAD       0xcb096f43     /* s_state: bad root */
#define     FsBADBLK    0xbadbc14b     /* s_state: bad block corrupted it */
```

S_type indicates the file system type. Currently, four types of file systems are supported: 1024-byte, 2048-byte, 4096-byte, and 8192-byte logical blocks. *S_magic* distinguishes the original 512-byte oriented file systems, which are no longer supported, from the newer file systems. If this field is not equal to the magic number, *fsMAGIC*, the type is assumed to be *fs1b*; otherwise the *s_type* field is used. A logical block is therefore determined by the type. The 1024-byte, 2048-byte, 4096-byte, and 8192-byte logical block file system will use two, four, eight, or sixteen physical blocks, respectively. The operating system takes care of all conversions from logical block numbers to physical block numbers.

S_state indicates the state of the file system. A cleanly unmounted, not damaged file system is indicated by the FsOKAY state. After a file system has been mounted for update, the state changes to FsACTIVE. A special case is used for the root file system. If the root file system appears damaged at boot time, it is mounted but marked FsBAD. Lastly, after a file system has been unmounted, the state reverts to FsOKAY.

S_isize is the address of the first data block after the i-list; the i-list starts just after the super-block, namely in block 2; thus the i-list is *s_isize*–2 blocks long. *S_fsize* is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers; if an "impossible" block number is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *s_free* array contains, in *s_free*[1], . . ., *s_free*[*s_nfree*–1], up to 49 numbers of free blocks. *S_free*[0] is the block number of the head of a chain of blocks constituting the free list. The first long in each free-chain block is the number (up to 50) of free-block numbers listed in the next 50 longs of this chain member. The first of these 50 blocks is the link to the next member of the chain. To allocate a block: decrement *s_nfree*, and the new block is *s_free*[*s_nfree*]. If the new block number is 0, there are no blocks left, so give an error. If *s_nfree* became 0, read in the block named by the new block number, replace *s_nfree* by its first word, and copy the block numbers in the next 50 longs into the *s_free* array. To free a block, check if *s_nfree* is 50; if so, copy *s_nfree* and the *s_free* array into it, write it out, and set *s_nfree* to 0. In any event set *s_free*[*s_nfree*] to the freed block's number and increment *s_nfree*.

S_tfree is the total free blocks available in the file system.

S_ninode is the number of free i-numbers in the *s_inode* array. To allocate an i-node: if *s_ninode* is greater than 0, decrement it and return *s_inode*[*s_ninode*]. If it was 0, read the i-list and place the numbers of all free i-nodes (up to 100) into the *s_inode* array, then try again. To free an i-node, provided *s_ninode* is less than 100, place its number into *s_inode*[*s_ninode*] and increment *s_ninode*. If *s_ninode* is already 100, do not bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the i-node is really free or not is maintained in the i-node itself.

S_tinode is the total free i-nodes available in the file system.

S_flock and *s_ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *s_fmod* on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

S_ronly is a read-only flag to indicate write-protection.

S_time is the last time the super-block of the file system was changed, and is the number of seconds that have elapsed since 00:00 Jan. 1, 1970 (GMT). During a reboot, the *s_time* of the super-block for the root file system is used to set the system's idea of the time.

S_fname is the name of the file system and *s_fpack* is the name of the pack.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 64 bytes long. I-node 1 is reserved for future use. I-node 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. For the format of an i-node and its flags, see *inode*(4).

SEE ALSO

mount(2), inode(4).
fsck(1M), fsdb(1M), mkfs(1M) in the *System Administrator's Reference Manual*.

NAME

fspec – format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files 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 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.

SEE ALSO

ed(1), newform(1), tabs(1) in the *User's Reference Manual*.

NAME

fstab – file-system-table

DESCRIPTION

The **/etc/fstab** file contains information about file systems for use by *mount*(1M) and *mountall*(1M). Each entry in **/etc/fstab** has the following format:

column 1	block special file name of file system or advertised remote resource
column 2	mount-point directory
column 3	"–r" if to be mounted read-only; "–d[r]" if remote
column 4	(optional) file system type string
column 5+	ignored

White-space separates columns. Lines beginning with "# " are comments. Empty lines are ignored.

A file-system-table might read:

```
/dev/dsk/m323_0s2 /usr  S51K
/dev/dsk/m323_1s2 /usr/src -r
adv_resource /mnt -d
```

FILES

/etc/fstab

SEE ALSO

mount(1M), mountall(1M), rmountall(1M) in the *System Administrator's Reference Manual.*

NAME

gettydefs – speed and terminal settings used by getty

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*(1) 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.

NOTE: Customers who need to support terminals that pass 8 bits to the system (as is typical outside the U.S.A.) must modify the entries in **/etc/gettydefs** as described in the **WARNINGS** section.

Each entry in **/etc/gettydefs** has the following format:

label# initial-flags # final-flags # login-prompt #next-label

Each entry is followed by a blank line. 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*(1M) 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 if a terminal type is not specified to *getty*(1M). The flags that *getty*(1M) understands are the same as the ones listed in **/usr/include/sys/termio.h** [see *termio*(7)]. Normally only the speed flag is required in the *initial-flags*. *getty*(1M) 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*(1M) executes *login*(1).

final-flags These flags take the same values as the *initial-flags* and are set just before *getty*(1M) executes *login*(1). 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.

next-label If this entry does not specify the desired speed, indicated by the user typing a <*break*> character, then *getty*(1M) will search for the entry with *next-label* as its *label* field and set 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*(1M) 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*(1M) can not find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into *getty*(1M) which will bring up a terminal at **300** baud.

After making or modifying **/etc/gettydefs**, it is strongly recommended that the file be run through *getty*(1M) with the check option to be sure there are no errors.

FILES

/etc/gettydefs

SEE ALSO

getty(1M), termio(7) in the *System Administrator's Reference Manual*.
ioctl(2) in the *Programmer's Reference Manual*.
login(1), stty(1) in the *User's Reference Manual*.

WARNINGS

To support terminals that pass 8 bits to the system (also, see the **BUGS** section), modify the entries in the **/etc/gettydefs** file for those terminals as follows: add **CS8** to *initial-flags* and replace all occurrences of **SANE** with the values: **BRKINT IGNPAR ICRNL IXON OPOST ONLCR CS8 ISIG ICANON ECHO ECHOK**

An example of changing an entry in **/etc/gettydefs** is illustrated below. All the information for an entry must be on one line in the file.

Original entry:

CONSOLE # B9600 HUPCL OPOST ONLCR # B9600 SANE IXANY TAB3 HUPCL # Console Login: # console

Modified entry:

CONSOLE # B9600 CS8 HUPCL OPOST ONLCR # B9600 BRKINT IGNPAR ICNRL IXON OPOST ONLCR CS8 ISIG ICANON ECHO ECHOK IXANY TAB3 HUPCL # Console Login: # console

This change will permit terminals to pass 8 bits to the system so long as the system is in MULTI-USER state. When the system changes to SINGLE-USER state, the *getty*(1M) is killed and the terminal attributes are lost. So to permit a terminal to pass 8 bits to the system in SINGLE-USER state, after you are in SINGLE-USER state, type (see *stty*(1)):

```
stty -istrip cs8
```

BUGS

8-bit with parity mode is not supported.

NAME

group – group file

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 newline. If the password field is **NULL**, no password is demanded.

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 group IDs to names.

FILES

/etc/group

SEE ALSO

passwd(4)
passwd(1) in the *User's Reference Manual.*
newgrp(1M) in the *System Administrator's Reference Manual.*

NAME

host - system host name.

DESCRIPTION

The file **/etc/host** contains the system host name as an ASCII string. It is read by *gethostname*(3N) to determine the system host name when *uname*(3N) fails.

FILES

/etc/host

SEE ALSO

gethostname(3N)

NAME

inittab – script for the init process

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, i.e., 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**. For 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 that 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* because *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 the first time *init* goes from single-user to multi-user state after the system is booted. (If **initdefault** is set to **2**, the process will run right after the boot.) *init* starts the process, waits for its termination and, when it dies, does 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 (i.e., before the `Console Login:` prompt). 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

exec(2), open(2), signal(2)
getty(1M), init(1M) in the *System Administrator's Reference Manual*.
sh(1), who(1) in the *User's Reference Manual*.

NAME

inode – format of an i-node

SYNOPSIS

#include <sys/types.h>
#include <sys/ino.h>

DESCRIPTION

An *i-node* for a plain file or directory in a file system has the following structure defined by <sys/ino.h>.

```
/* Inode structure as it appears on a disk block. */
struct   dinode
{
    ushort di_mode;      /* mode and type of file */
    short  di_nlink;     /* number of links to file */
    ushort di_uid;       /* owner's user id */
    ushort di_gid;       /* owner's group id */
    off_t  di_size;      /* number of bytes in file */
    char   di_addr[39]; /* disk block addresses */
    char   di_gen;       /* file generation number */
    time_t di_atime;     /* time last accessed */
    time_t di_mtime;     /* time last modified */
    time_t di_ctime;     /* time of last file status change */
};
/*
 * the address bytes:
 *       39 used; 13 addresses
 *       of 3 bytes each.
 */
```

For the meaning of the defined types *off_t* and *time_t* see *types*(5).

SEE ALSO

stat(2), fs(4), types(5)

NAME

issue – issue identification file

DESCRIPTION

The file **/etc/issue** contains the *issue* or project identification to be printed as a login prompt. This is an ASCII file that is read by program *getty* and then written to any terminal spawned or respawned from the *lines* file.

FILES

/etc/issue

SEE ALSO

login(1) in the *User's Reference Manual.*

NAME

ldfcn – common object file access routines

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

DESCRIPTION

The common object file access routines are a collection of functions for reading common object files and archives containing common object files. Although the calling program must know the detailed structure of the parts of the object file that it processes, the routines effectively insulate the calling program from knowledge of the overall structure of the object file.

The interface between the calling program and the object file access routines is based on the defined type **LDFILE,** defined as **struct ldfile,** declared in the header file **ldfcn.h**. The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function *ldopen*(3X) allocates and initializes the **LDFILE** structure and returns a pointer to the structure to the calling program. The fields of the **LDFILE** structure may be accessed individually through macros defined in **ldfcn.h** and contain the following information:

LDFILE
*ldptr;

TYPE(ldptr)
The file magic number used to distinguish between archive members and simple object files.

IOPTR(ldptr)
The file pointer returned by *fopen* and used by the standard input/output functions.

OFFSET(ldptr)
The file address of the beginning of the object file; the offset is non-zero if the object file is a member of an archive file.

HEADER(ldptr)
The file header structure of the object file.

The object file access functions themselves may be divided into four categories:

(1) functions that open or close an object file

ldopen(3X) and *ldaopen*[see *ldopen*(3X)]
open a common object file
ldclose(3X) and *ldaclose*[see *ldclose*(3X)]
close a common object file

(2) functions that read header or symbol table information

ldahread(3X)
read the archive header of a member of an archive file
ldfhread(3X)
read the file header of a common object file
ldshread(3X) and *ldnshread*[see *ldshread*(3X)]
read a section header of a common object file
ldtbread(3X)
read a symbol table entry of a common object file
ldgetname(3X)
retrieve a symbol name from a symbol table entry or from the string table

(3) functions that position an object file at (seek to) the start of the section, relocation, or line number information for a particular section.

ldohseek(3X)
seek to the optional file header of a common object file
ldsseek(3X) and *ldnsseek*[see *ldsseek*(3X)]
seek to a section of a common object file
ldrseek(3X) and *ldnrseek*[see *ldrseek*(3X)]
seek to the relocation information for a section of a common object file
ldlseek(3X) and *ldnlseek*[see *ldlseek*(3X)]
seek to the line number information for a section of a common object file
ldtbseek(3X)
seek to the symbol table of a common object file

(4) the function *ldtbindex*(3X) which returns the index of a particular common object file symbol table entry.

These functions are described in detail on their respective manual pages.

All the functions except *ldopen*(3X), *ldgetname*(3X), *ldtbindex*(3X) return either **SUCCESS** or **FAILURE,** both constants defined in **ldfcn.h**. *Ldopen*(3X) and *ldaopen*[(see *ldopen*(3X)] both return pointers to an **LDFILE** structure.

Additional access to an object file is provided through a set of macros defined in **ldfcn.h**. These macros parallel the standard input/output file reading and manipulating functions, translating a reference of the **LDFILE** structure into a reference to its file descriptor field.

The following macros are provided:

GETC(ldptr)
FGETC(ldptr)
GETW(ldptr)
UNGETC(c, ldptr)
FGETS(s, n, ldptr)
FREAD((char *) ptr, sizeof (*ptr), nitems, ldptr)
FSEEK(ldptr, offset, ptrname)
FTELL(ldptr)
REWIND(ldptr)
FEOF(ldptr)
FERROR(ldptr)
FILENO(ldptr)
SETBUF(ldptr, buf)
STROFFSET(ldptr)

The STROFFSET macro calculates the address of the string table. See the manual entries for the corresponding standard input/output library functions for details on the use of the rest of the macros.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fseek(3S), ldahread(3X), ldclose(3X), ldgetname(3X), ldfhread(3X), ldlread(3X), ldlseek(3X), ldohseek(3X), ldopen(3X), ldrseek(3X), ldlseek(3X), ldshread(3X), ldtbindex(3X), ldtbread(3X), ldtbseek(3X), stdio(3S), intro(5)

WARNING

The macro **FSEEK** defined in the header file **ldfcn.h** translates into a call to the standard input/output function *fseek*(3S). **FSEEK** should not be used to seek from the end of an archive file since the end of an archive file may not be the same as the end of one of its object file members!

NAME

limits – file header for implementation-specific constants

SYNOPSIS

#include <limits.h>

DESCRIPTION

The header file **<limits.h>** is a list of magnitude limitations imposed by a specific implementation of the operating system. All values are specified in decimal.

```
#define  CHAR_BIT      8              /* # of bits in a "char" */
#define  CHAR_MAX      255            /* max value of a "char" */
#define  CHAR_MIN      0              /* min value of a "char" */
#define  INT_MAX       2147483647     /* max value of an "int" */
#define  INT_MIN       -2147483648    /* min value of an "int" */
#define  LONG_MAX      2147483647     /* max value of a "long" */
#define  LONG_MIN      -2147483648    /* min value of a "long" */
#define MB_LEN_MAX     1              /* max number of bytes in a multibyte
                                      /* character */
#define SCHAR_MAX      127            /* max value for signed char */
#define SCHAR_MIN      -127           /* min value for signed char */
#define  SHRT_MAX      32767          /* max value of a "short" */
#define  SHRT_MIN      -32768         /* min value of a "short" */
#define UCHAR_MAX      255            /* max value of an "unsigned char" */
#define UINT_MAX       4294967295     /* max value of an "unsigned int" */
#define ULONG_MAX      4294967295     /* max value of an "unsigned long" */
#define USHRT_MAX      65535          /* max value of an "unsigned short" */

/*
 * POSIX Minimum Values (set by POSIX)
 */
#define _POSIX_ARG_MAX        4096    /* length of the arguments for exec */
                                      /* in bytes including environment   */
                                      /* data */
#define _POSIX_CHILD_MAX      6       /* The number of simultaneous procs */
                                      /* per user ID */
#define _POSIX_LINK_MAX       8       /* The value of a file's link count */
#define _POSIX_MAX_CANON      255     /* The number of bytes in a terminal */
                                      /* canonical input queue */
#define _POSIX_MAX_INPUT      255     /* The number of bytes for which    */
                                      /* space is guaranteed to be        */
                                      /* available for terminal input queue */
#define _POSIX_NAME_MAX       14      /* Number of bytes in a filename */
#define _POSIX_NGROUPS_MAX    0       /* Number of simultaneous supp.  */
                                      /* group ID's per process */
#define _POSIX_OPEN_MAX       16      /* The number of files that one proc */
                                      /* can have open at a time */
#define _POSIX_PATH_MAX       255     /* Max number of bytes in a pathname */
```

```
#define _POSIX_PIPE_BUF        512     /* Number of bytes guaranteed to be  */
                                       /* written atomically to a pipe */

/*
 * POSIX Run-Time Increasable Values (set by BCS)
 */
#define NGROUPS_MAX            0       /* Minimum maximum number of */
                                       /* simultanueous supplementary */
                                       /* group IDs per process */

/*
 * POSIX Run-Time Invariant Values (unset by BCS)
 */
#undef ARG_MAX                         /* traditional max for exec args */
#undef CHILD_MAX                       /* max # of processes per user id */
#undef OPEN_MAX                        /* max # of open files per process */

/*
 * POSIX Pathname Variable Values (unset by BCS)
 */
#undef LINK_MAX                        /* max # of links to a single file */
#undef MAX_CANON                       /* Maximum number of bytes in a */
                                       /* terminal canonical input line */
#undef MAX_INPUT                       /* Minimum number of bytes for which */
                                       /* for which space is guaranteed in */
                                       /* a terminal input queue. */
#undef NAME_MAX                        /* max # of characters in a file name */
#undef PATH_MAX                        /* max # of characters in a path name */
#undef PIPE_BUF                        /* max # bytes atomic in write to a
                                       /* pipe */


#ifndef _POSIX_SOURCE
/*
 * Non-POSIX symbols must be hidden by _POSIX_SOURCE
 */
#define   DBL_DIG   16             /* digits of precision of a "double" */
#define   DBL_MAX   1.79769313486231470e+308 /* max decimal value of a "double" */
#define   DBL_MIN   ((double)4.4501477170144023e-308) /*min decimal value of a "double"*/
#define   FCHR_MAX  1048576        /* max size of a file in bytes */
#define   FLT_DIG   7              /* digits of precision of a "float" */
#define   FLT_MAX   3.40282346638528860e+38 /*max decimal value of a "float" */
#define   FLT_MIN   1.40129846432481707e-45  /*min decimal value of a "float" */
#define   HUGE_VAL  FLT_MAX        /* error value returned by math lib */
#define   PASS_MAX  8              /* max # of characters in a password */
#define   PID_MAX   30000          /* max value for a process ID */
#define   PIPE_MAX  8192           /* max # bytes written to a pipe in a write */
#define   STD_BLK   1024           /* # bytes in a physical I/O block */
#define   SYS_NMLN  9              /* # of chars in uname-returned strings */
```

```
#define  UID_MAX   60000        /* max value for a user or group ID */
#define  USI_MAX   UINT_MAX     /* max decimal value of an "unsigned" */
#define  WORD_BIT  32           /* # of bits in a "word" or "int" */

#endif   /* _POSIX_SOURCE */
```

NAME

linenum – line number entries in a common object file

SYNOPSIS

#include <linenum.h>

DESCRIPTION

The *cc* command generates an entry in the object file for each C source line on which a breakpoint is possible (when invoked with the **–g** option; see *cc*(1)). Users can then reference line numbers when using the appropriate software test system (see *sdb*(1)). The structure of these line number entries is:

```
struct      lineno
{
        union
        {
                long    l_symndx ;
                long    l_paddr ;
        }               l_addr ;
        unsigned shortl_lnno ;
#if defined (m88k)
        char    l_pad1;
        char    l_pad2;
#endif
} ;
```

Numbering starts with one for each function. The initial line number entry for a function has *l_lnno* equal to zero, and the symbol table index of the function's entry is in *l_symndx*. Otherwise, *l_lnno* is non-zero, and *l_paddr* is the physical address of the code for the referenced line. Thus, the overall structure is:

```
l_addr                     l_lnno

function symtab index0
physical address     line
physical address     line
...

function symtab index0
physical address     line
physical address     line
...
```

SEE ALSO

cc(1), sdb(1), a.out(4)

NAME

loginlog – log of failed login attempts

DESCRIPTION

After five unsuccessful login attempts, all the attempts are logged in the **loginlog** file. This file contains one record for each failed attempt. Each record contains the following information:

login name
tty specification
time

This is an ASCII file. Each field within each entry is separated from the next by a colon. Each entry is separated from the next by a new-line.

By default, **loginlog** does not exist, so no logging is done. To enable logging, the log file must be created with read and write permission for owner only. Owner must be **root** and group must be **sys**.

FILES

/usr/adm/loginlog

SEE ALSO

login(1), passwd(1) in the *User's Reference Manual*.
passwd(1M) in the *System Administrator's Reference Manual*.

NAME

master – master device information table

DESCRIPTION

The *master* file is used by the *config*(1M) program to obtain device information that enables it to generate the configuration file **conf.c**. *config* reads *dfile* and places information from each Part 1 entry into the arrays provided by *master*. Refer to *config*(1M) for information about the file produced and to *dfile*(4) for information about the fields in the first part of the user-supplied *dfile*.

master has 5 parts, each separated by a line with a dollar sign (**$**) in column 1. Any line with an asterisk (*) in column 1 is treated as a comment. Part 1 contains device information; part 2 contains names of devices that have aliases; part 3 contains tunable parameter information. Parts 4 and 5 contain information related to configuring the M88000 family systems. Part 4, the microprocessor specification, must appear in *master* and cannot be in the user-specified *dfile*. Part 5 contains lines exactly like those for the M88000-specific portion of *dfile*. See *dfile*(4) for a description of these lines.

The following paragraphs describe the 5 parts of the *master* file. In this description, the VME323 disk controller is used as an example.

PART 1

Part 1 contains lines consisting of at least 10 fields and at most 13 fields; the fields are delimited by tabs and/or blanks:

Field 1: device name (8 characters maximum).

Field 2: interrupt vectors size (decimal); the size is the number of vectors multiplied by four. Refer to Table 6-2 in the *M88100 32-Bit Microprocessor User's Manual* (M88100UM/AD) for information on the memory map for exception vectors.

Field 3: device mask (octal); each "on" bit indicates that the handler exists.

002000	device has a select handler
001000	device has a stream handler

000400	separate open and close for block and character devices; setting the 000400 bit and the 000020 bit results, for example, in **m323bopen** for opening the block device and **m323copen** for opening the character device.
000200	device has a tty structure
000100	initialization handler
000040	power-failure handler
000020	open handler
000010	close handler
000004	read handler
000002	write handler
000001	ioctl handler

Field 4: device type indicator (octal):

004000	create interrupt vector array; e.g., **m323_ivec[ ]**; each vector (hexadecimal) specified in *dfile* (vector number multiplied by 4) is placed in this array.
002000	create character major number or block major number for the device (e.g., **m323_cmaj** or **m323_bmaj**).
001000	create interrupt level array; e.g., **m323_ilev[ ]**; interrupt levels are specified in the fourth field ("bus") of each line in the first part of the *dfile*.
000200	allow only one of these devices
000100	suppress count field in the **conf.c** file
000040	suppress interrupt vector
000020	required device
000010	block device
000004	character device
000002	interrupt driven device other than block or char. device
000001	allow for a single vector definition with multiple addresses

Field 5: handler prefix (4 chars. maximum); e.g., m323.

Field 6: page registers size (decimal); the span of memory for all the device registers on the device page, starting at the dfile address.

Field 7: major device number for block-type device.

Field 8: major device number for character-type device.

Field 9: maximum number of devices per controller (decimal); e.g., **m323_cnt**; *cnt* is the optional fifth field on each line in the first part of *dfile*. If more than one controller is listed in *dfile*, however, then example will be the sum of the devices for all the controllers (e.g., A number specified in *dfile* overrides this field in *master*.

Field 10: maximum bus request level (1 through 7).

Fields 11-13: optional configuration table structure declarations (8 chars. maximum)

Devices that are not interrupt-driven have an interrupt vector size of zero. The 040 bit in Field 4 causes *config*(1M) to record the interrupt vectors although the **m88kvec.s** file will show no interrupt vector assignment at those locations (interrupts here will be treated as strays).

PART 2

Part 2 contains lines with 2 fields each:

Field 1: alias name of device (8 chars. maximum).

Field 2: reference name of device (8 chars. maximum; specified in part 1)

PART 3

Part 3 contains lines with 2 or 3 fields each:

Field 1: parameter name (as it appears in *dfile*; 30 chars. maximum)

Field 2: parameter name (as it appears in the **conf.c** file; 30 chars. maximum)

Field 3: default parameter value (30 chars. maximum; parameter specification is required if this field is omitted)

PART 4

Part 4 contains one line with two fields for the microprocessor specification.

Field 1	mpu	
Field 2	*number*	where *number* is 88100, or 88110. The default is 88100.

PART 5

Part 5 contains M88000-specific lines exactly like those for the M88000-specific portion of the *dfile*. See the *dfile*(4) for a description of these lines.

FILES

/etc/master

SEE ALSO

config(1M), sysdef(1M), dfile(4)

NAME

mnttab – mounted file system table

SYNOPSIS

#include <mnttab.h>

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[32];
         char      mt_filsys[32];
         short     mt_ro_flg;
         time_t    mt_time;
         char      mt_fstyp[16];
         char      mt_mntopts[64];
};
```

Each entry is 150 bytes; the first 32 bytes are the null-padded name of the place where the *special file* is mounted; the next 32 bytes represent the null-padded root name of the mounted special file; the next 6 bytes contain the mounted *special files* read/write permissions and the date on which it was mounted; the following 16 bytes are the null-padded name of file system type; and the remaining 64 bytes are the null-ppadded string of mount options. The mount options are only used for an NFS file system.

The maximum number of entries in *mnttab* is based on the system parameter NMOUNT located in **/usr/src/uts/mot/sysgen/descriptions/kernel,** which defines the number of allowable mounted special files.

FILES

/etc/mnttab

SEE ALSO

mount(1M), setmnt(1M) in the *System Administrator's Reference Manual.*

NAME

.ott – files that hold object architecture information

DESCRIPTION

The FACE object architecture stores information about object-types in an ASCII file named **.ott** (object type table) that is contained in each directory. This file describes all of the objects in that directory. Each line of the **.ott** file contains information about one object in pipe separated fields. The fields are (in order):

name
: the name of the actual System file.

dname
: the name that should be displayed to the user, or a dot if it is the same as the name of the file.

description
: the description of the object, or a dot if the description is the default (the same as object-type).

object-type
: the FACE internal object type name.

flags
: object specific flags.

mod time
: the time that FACE last modified the object. The time is given as number of seconds since 1/1/1970, and is in hexadecimal notation.

object information
: an optional field, contains a set of semi-colon separated "name=value" fields that can be used by FACE to store any other information necessary to describe this object.

FILES

.ott is created in any directory opened by FACE.

NAME

passwd – password file

DESCRIPTION

/etc/passwd contains for each user the following information:

login name
password and (optional) aging
numerical user ID
numerical group ID
GCOS job number, box number, optional GCOS user ID
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. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user is separated from the next by a new-line. If the shell field is `NULL`, **/bin/sh** is used.

This file has user login information, and has general read permission. It can therefore be used, for example, to map numerical user IDs to names.

The password field contains of the character **x** if there is a **/etc/shadow** file. If **/etc/shadow** does not exist and the login does have a password, this field will contain an encrypted copy of the password. This field remains only for compatibility reasons when **/etc/shadow** exists.

The encrypted password consists of 13 characters chosen from a 64-character alphabet (**.**, **/**, **0–9**, **A–Z**, **a–z**), except when the password is null, in which case the encrypted password is also `NULL`. Password aging is effected for a particular user if his encrypted password in the password file is followed by a comma and a non-null string of characters from the above alphabet. (Such a string must be introduced in the first instance by the superuser.)

The first character of the age, e.g., *M*, denotes the maximum number of weeks for which a password is valid. A user who attempts to login after the password has expired will be forced to supply a new one. The next character, e.g., *M*, denotes the minimum period in weeks that must expire before the password may be changed. The remaining one or two characters define the week (counted from the beginning of 1970) when the password was last changed. (A null string is equivalent to zero.) *M* and *m* have numerical values in the range 0–63 that correspond to the 64-character alphabet shown above (i.e., / = 1 week; z = 63 weeks). If *m* = *M* = 0 (derived from the string . or ..) the user will be forced to change his password the next time he logs in (and the "age" will disappear from his entry in the password file). If *m* > *M* (signified, e.g., by the string ./) only the superuser will be able to change the password.

The **passwd** file can also have line beginning with a plus (+), which means to incorporate entries from the yellow pages. There are three styles of + entries: all by itself, + means to insert the entire contents of the yellow pages password file at that point; +*name* means to insert the entry (if any) for *name* from the yellow pages at that point; +@*name* means to insert the entries for all members of the network group *name* at that point. If a + entry has a nonnull password, directory, GCOS, or shell field, they will override what is contained in the yellow pages. The numeric user ID and group ID fields cannot be overridden.

EXAMPLE

The following is a sample **/etc/passwd** file:

```
root:q.mJzTnu8icF.:0:10:God:/:/bin/sh
ja:6k/7KCFRPNVXg:508:10:Jerry Asher:/usr2/ja:/bin/sh
+melissa:
+@documentation:no-login:
+:::Guest
```

In this example there are specific entries for users **root** and **ja**, in case the Yellow Pages (YP) are out of order. The user **melissa** has her password entry in the YP incorporated without change; anyone in the netgroup **documentation** has their password field disabled, and anyone else will be able to log in with their usual password, shell, and home directory, but with a GCOS field of **Guest**.

FILES

/etc/passwd
/etc/shadow

SEE ALSO

group(4)
getpwent(3C) in the *Programmer's Reference Manual*.
login(1), passwd(1) in the *User's Reference Manual*.
passwd(1M) in the *System Administrator's Reference Manual*.

NAME

profile – setting up an environment at login time

SYNOPSIS

/etc/profile
$HOME/.profile

DESCRIPTION

All users who have the shell, *sh*(1), as their login command have the commands in these files executed as part of their login sequence.

/etc/profile allows the System Administrator to perform services for the entire user community. Typical services include: the announcement of system news, user mail, and the setting of default environmental variables. It is not unusual for **/etc/profile** to execute special actions for the **root** login or the *su*(1) command. Computers running outside the Eastern time zone should have the line

. /etc/TIMEZONE

included early in **/etc/profile** (see timezone(4)).

The file **$HOME/.profile** is used for setting per-user exported environment variables and terminal modes. The following example is typical (except for the comments):

```
# Make some environment variables global
export MAIL PATH TERM
# Set file creation mask
umask 027
# Tell me when new mail comes in
MAIL=/usr/mail/$LOGNAME
#  Add my /bin directory to the shell search sequence
PATH=$PATH:$HOME/bin
#  Set terminal type
while :
do   echo "terminal: \c"
     read TERM
     if [-f ${TERMINFO:-/usr/lib/terminfo}/?/$TERM]
     then break
     elif [ -f /usr/lib/terminfo/?/$TERM ]
     then break
     else echo "invalid term $TERM" 1>&2
     fi
done
# Initialize the terminal and set tabs
# The environmental variable TERM must have been exported
```

```
# before the "tput init" command is executed.
tput init
#  Set the erase character to backspace
stty erase '^H' echoe
```

FILES

/etc/TIMEZONE timezone environment
$HOME/.profile user-specific environment
/etc/profile system-wide environment

SEE ALSO

terminfo(4), timezone(4), environ(5), term(5)
env(1), login(1), mail(1), sh(1), stty(1), su(1), tput(1) in the *User's Reference Manual*.
su(1M) in the *System Administrator's Reference Manual*.
User's Guide.
Chapter 10 in the *Programmer's Guide*.

NOTES

Care must be taken in providing system-wide services in **/etc/profile**. Personal **.profile** files are better for serving all but the most global needs.

NAME

reloc – relocation information for a common object file

SYNOPSIS

#include <reloc.h>

DESCRIPTION

Object files have one relocation entry for each relocatable reference in the text or data. If relocation information is present, it will be in the following format:

```
struct reloc
{
    long r_vaddr;         /* (virtual) address of reference */
    long r_symndx;        /* index into symbol table */
    unsigned short        r_type;   /* relocation type */
    char r_pad1;          /* pad to 4 byte multiple */
    char r_pad2;          /* pad to 4 byte multiple */

} ;

#define R_ABS             0
#define R_PCR16L          128
#define R_PCR26L          129
#define R_VRT16           130
#define R_HVRT16          131
#define R_LVRT16          132
#define R_VRT32           133
```

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated.

R_ABS
: The reference is absolute and no relocation is necessary. The entry will be ignored.

R_PCR16L
: A "PC-Relative" 16-bit reference to a symbol's virtual address. The actual address is calculated by adding a constant to the PC value.

R_PCR26L
: A "PC-Relative" 26-bit reference to a symbol's virtual address. The actual address is calculated by adding a constant to the PC value.

R_VRT16
: Direct 16-bit reference to the symbol's virtual address.

R_HVRT16
: Same as R_VRT16, except, only the high 16-bits are used in the relocation.

R_LVRT16
: Same as R_VRT16, except, only the low 16-bits are used in the relocation.

R_VRT32
: Direct 32-bit reference to the symbol's virtual address.

More relocation types exist for other processors. Equivalent relocation types on different processors have equal values and meanings. New relocation types will be defined (with new values) as they are needed.

Relocation entries are generated automatically by the assembler and automatically used by the link editor. Link editor options exist for both preserving and removing the relocation entries from object files.

SEE ALSO

as(1), ld(1), a.out(4), syms(4)

NAME

rfmaster – Remote File Sharing name server master file

DESCRIPTION

The **rfmaster** file is an ASCII file that identifies the hosts that are responsible for providing primary and secondary domain name service for Remote File Sharing domains. This file contains a series of records, each terminated by a newline; a record may be extended over more than one line by escaping the newline character with a backslash ("\"). The fields in each record are separated by one or more tabs or spaces. Each record has three fields:

name *type* *data*

The *type* field, which defines the meaning of the *name* and *data* fields, has three possible values:

p The **p** type defines the primary domain name server. For this type, *name* is the domain name and *data* is the full host name of the machine that is the primary name server. The full host name is specified as *domain.nodename*. There can be only one primary name server per domain.

s The **s** type defines a secondary name server for a domain. *Name* and *data* are the same as for the **p** type. The order of the **s** entries in the **rfmaster** file determines the order in which secondary name servers take over when the current domain name server fails.

a The **a** type defines a network address for a machine. *Name* is the full domain name for the machine and *data* is the network address of the machine. The network address can be in plain ASCII text or it can be preceded by a \x to be interpreted as hexadecimal notation. (See the documentation for the particular network you are using to determine the network addresses you need.)

There are at least two lines in the **rfmaster** file per domain name server: one **p** and one **a** line, to define the primary and its network address. There should also be at least one secondary name server in each domain.

This file is created and maintained on the primary domain name server. When a machine other than the primary tries to start Remote File Sharing, this file is read to determine the address of the primary. If **rfmaster** is missing, the **-p** option of **rfstart** must be used to identify the primary. After that, a copy of the primary's **rfmaster** file is automatically placed on the machine.

Domains not served by the primary can also be listed in the **rfmaster** file. By adding primary, secondary, and address information for other domains on a network, machines served by the primary will be able to share resources with machines in other domains.

A primary name server may be a primary for more than one domain. However, the secondaries must then also be the same for each domain served by the primary.

EXAMPLE

An example of an **rfmaster** file is shown below. (The network address examples, *comp1.serve* and *comp2.serve*, are STARLAN network addresses.)

```
ccs         p    ccs.comp1
ccs         s    ccs.comp2
ccs.comp2   a    comp2.serve
ccs.comp1   a    comp1.serve
```

NOTE: If a line in the **rfmaster** file begins with a # character, the entire line will be treated as a comment.

FILES

/usr/nserve/rfmaster

SEE ALSO

rfstart(1M) in the *System Administrator's Reference Manual*.

NAME

.rhosts – user-specified file of equivalent hosts and users.

DESCRIPTION

The *.rhosts* file resides in a user's login directory. It contains entries, one per line, which are of the form:

hostname

or

hostname username

It allows a user to specify a set of users of other systems who are allowed equivalent capabilities to himself on this system.

In an environment where a single organization might have many systems used by a common set of users, it is often the case that a single user will have a login account on many different systems. In the common case where the login names are the same for each user on all systems, then user authentication is provided by the list of host names in **/etc/hosts.equiv**. In the case where a host is not in **/etc/hosts.equiv**, or the user has a different name on another system, the user can provide individual authentication by adding entries in his personal **.rhosts** file. Users who connect to the system, via *rcp*, *remsh*, or *rlogin* and are authorized via **user granting authorization.**

remshd, used to support *remsh* and *rcp* requests, uses **.rhosts** in the following way. When the connection is made, *remshd* gets the name of the user on the remote (calling) system. It then looks up the remote user in the local **/etc/passwd** file. If the remote user is not the superuser, then **/etc/hosts.equiv** is checked for the name of the remote host. If it is found, the user is considered to be equivalent to the user of the same local name, and the command proceeds. If the host name is not found, or if the remote user is the superuser, then *remshd* checks the file **.rhosts** in the login directory found in **/etc/passwd**. If an entry is found for the remote host, or this local user name and remote host combination, then the user is considered equivalent and the command proceeds. If this test fails, the command is terminated. *rlogin* uses these files in an analogous fashion.

The host name here must match the first name listed for a host in **/etc/hosts**, not one of its aliases. As a convenience, the RFS setup menu, **transpmgmt tcpip**, adds and deletes entries in **/etc/rhosts**. RFS commands, however, do not make use of the entries in this file.

FILES

$HOME/.rhosts

SEE ALSO

rcp(1), rlogin(1), remsh(1), rlogind(1M), remshd(1M), hosts.equiv(4)

NOTES

On most systems, users are required to enter their password (if they have one) on the remote system. This is due to the operation of login on these systems.

NAME

sccsfile – format of SCCS file

DESCRIPTION

An SCCS (Source Code Control System) 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) 064001.

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. Any line starting with a **!** prohibits the succeeding group or user from making deltas.

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	<keyword string>
@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>
@f z	<reserved for use in interfaces>

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. The **z** flag is used in certain specialized interface programs.

Comments

Arbitrary text is 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 do not 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).

NAME

scnhdr – section header for a common object file

SYNOPSIS

#include <scnhdr.h>

DESCRIPTION

Every common object file has a table of section headers to specify the layout of the data within the file. Each section within an object file has its own header. The C structure appears below.

```
struct      scnhdr
{
        char            s_name[SYMNMLEN]; /* section name */
        long            s_paddr;  /* physical address */
        long            s_vaddr;  /* virtual address */
        long            s_size;   /* section size */
        long            s_scnptr; /* file ptr to raw data */
        long            s_relptr; /* file ptr to relocation */
        long            s_lnnoptr;/* file ptr to line numbers */
        unsigned                s_nreloc;/* # reloc entries */
        unsigned                s_nlnno;/* # line number entries */
        long            s_flags;  /* flags */
} ;
```

File pointers are byte offsets into the file; they can be used as the offset in a call to FSEEK [see *ldfcn*(4)]. If a section is initialized, the file contains the actual bytes. An uninitialized section is somewhat different. It has a size, symbols defined in it, and symbols that refer to it. But it can have no relocation entries, line numbers, or data. Consequently, an uninitialized section has no raw data in the object file, and the values for *s_scnptr*, *s_relptr*, *s_lnnoptr*, *s_nreloc*, and *s_nlnno* are zero.

SEE ALSO

ld(1), fseek(3S), a.out(4).

NAME

scr_dump – format of curses screen image file

SYNOPSIS

scr_dump(file)

DESCRIPTION

The *curses*(3X) function *scr_dump*() will copy the contents of the screen into a file. The format of the screen image is as described below.

The name of the tty is 20 characters long and the modification time (the *mtime* of the tty that this is an image of) is of the type *time_t*. All other numbers and characters are stored as *chtype* (see **<curses.h>**). No newlines are stored between fields.

<magic number: octal 0433>
<name of tty>
<mod time of tty>
<columns> <lines>
<line length> <chars in line> for each line on the screen
<line length> <chars in line>
.
.
.
<labels?> **1**, if soft screen labels are present
<cursor row> <cursor column>

Only as many characters as are in a line will be listed. For example, if the *<line length>* is 0, there will be no characters following *<line length>*. If *<labels?>* is TRUE, following it will be:

<number of labels>
<label width>
<chars in label 1>
<chars in label 2>
.
.
.

SEE ALSO

curses(3X).

NAME

syms – common object file symbol table format

SYNOPSIS

#include <syms.h>

DESCRIPTION

Common object files contain information to support symbolic software testing (see *sdb*(1)). Line number entries, *linenum*(4), and extensive symbolic information permit testing at the C *source* level. Every object file's symbol table is organized as:

```
File name 1.
        Function 1.
                Local symbols for function 1.
        Function 2.
                Local symbols for function 2.
        ...
        Static externs for file 1.

File name 2.
        Function 1.
                Local symbols for function 1.
        Function 2.
                Local symbols for function 2.
        ...
        Static externs for file 2.
...

Defined global symbols.
Undefined global symbols.
```

The entry for a symbol is a fixed-length structure. The members of the structure hold the name (null padded), its value, and other information. The following is the C structure:

```
#define   SYMNMLEN   8
#define   FILNMLEN   14
#define   DIMNUM     4

struct  syment
{
  union    /* all ways to get symbol name */
    {
```

```
        char    _n_name[SYMNMLEN]; /* symbol name */
        struct
        {
          long  _n_zeroes; /* == OL when in string table */
          long  _n_offset; /* location of name in table */
        } _n_n;
        char    *_n_nptr[2];/* allows overlaying */
      } _n;
      long       n_value     /* value of symbol */
      short      n_scnum;    /* section number */
      unsigned short n_type;/* type and derived type */
      char       n_sclass;   /* storage class */
      char       n_numaux;   /* number of aux entries */
    #if defined (m88k)
      char       n_pad1;     /* pad to 4 byte multiple */
      char       n_pad2;     /* pad to 4 byte multiple */
    #endif
    };

    #define  n_name    _n._n_name
    #define  n_zeroes  _n._n_n._n_zeroes
    #define  n_offset  _n._n_n._n_offset
    #define  n_nptr    _n._n_nptr[1]
```

Meaningful values and explanations for them are given in both **syms.h** and COFF. Anyone who needs to interpret the entries should seek more information in these sources. Some symbols require more information than a single entry; they are followed by *auxiliary entries* that are the same size as a symbol entry. The format follows.

```
union auxent
{
        struct
        {
                long            x_tagndx;
                union
                {
                        struct
                        {
                                unsigned shortx_lnno;
                                unsigned shortx_size;
                        } x_lnsz;
                        long    x_fsize;
                } x_misc;
                union
```

```
                {
                        struct
                        {
                                long    x_lnnoptr;
                                long    x_endndx;
                        }       x_fcn;
                        struct
                        {
                                unsigned shortx_dimen[DIMNUM];
                        }       x_ary;
                }               x_fcnary;
                unsigned short  x_tvndx;
#if defined (m88k)
        char r_pad1 ;   /* pad to 4 byte multiple */
        char r_pad2 ;   /* pad to 4 byte multiple */
#endif
        }       x_sym;
        struct
        {
                char    x_fname[FILNMLEN];
        }       x_file;
           struct
           {
                   long     x_scnlen;
                   unsigned short  x_nreloc;
                   unsigned short  x_nlinno;
           }       x_scn;

        struct
        {
                long            x_tvfill;
                unsigned short      x_tvlen;
                unsigned short      x_tvran[2];
        }       x_tv;
};
```

Indexes of symbol table entries begin at *zero*.

SEE ALSO

sdb(1), a.out(4), linenum(4)
Common Object File Format in the *Programming Guide.*

WARNINGS

On machines on which **ints** are equivalent to **longs**, all **longs** have their type changed to **int.** Thus, the information about which symbols are declared as **longs** and which, as **ints**, does not show up in the symbol table.

NAME

system – system configuration information table

DESCRIPTION

This file is used by the **boot** program to obtain configuration information that cannot be obtained from the equipped device table (EDT) at system boot time. This file generally contains a list of software drivers to include in the load, the assignment of system devices such as *pipedev* and *swapdev*, as well as instructions for manually overriding the drivers selected by the self-configuring boot process.

The syntax of the system file is given below. The parser for the **/etc/system** file is case sensitive. All uppercase strings in the syntax below should be uppercase in the **/etc/system** file as well. Nonterminal symbols are enclosed in angle brackets "<>" while optional arguments are enclosed in square brackets "[ ]". Ellipses "..." indicate optional repetition of the argument for that line.

<fname>	::=	pathname
<string>	::=	driver file name from **/boot** or EDT entry name
<device>	::=	special device name | DEV(<major>,<minor>)
<major>	::=	<number>
<minor>	::=	<number>
<number>	::=	decimal, octal or hex literal

The lines listed below may appear in any order. Blank lines may be inserted at any point. Comment lines must begin with an asterisk. Entries for EXCLUDE and INCLUDE are cumulative. For all other entries, the last line to appear in the file is used; any earlier entries are ignored.

BOOT: <fname>
: specifies the kernel a.out file to be booted; if the file is fully resolved [such as that produced by the *mkunix*(1M) program] then all other lines in the *system* file have no effect.

EXCLUDE: [<string>] ...
: specifies drivers to exclude from the load even if the device is found in the EDT.

INCLUDE: [<string>[(<number>)]] ...
specifies software drivers or loadable modules to be included in the load. This is necessary to include the drivers for software "devices". The optional <number> (parenthesis required) specifies the number of "devices" to be controlled by the driver (defaults to 1). This number corresponds to the builtin variable *#c* which may be referred to by expressions in part one of the **/etc/master** file.

ROOTDEV: <device>
identifies the device containing the root file system.

SWAPDEV: <device> <number> <number>
identifies the device to be used as swap space, the block number the swap space starts at, and the number of swap blocks available.

PIPEDEV: <device>
identifies the device to be used for pipe space.

FILES

/etc/system

SEE ALSO

master(4)
crash(1M), mkunix(1M), mkboot(1M) in the *System Administrator's Reference Manual*.

NAME

term – format of compiled term file.

SYNOPSIS

/usr/lib/terminfo/?/*

DESCRIPTION

Compiled *terminfo*(4) descriptions are placed under the directory **/usr/lib/terminfo**. In order to avoid a linear search of a huge 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, **att4425** can be found in the file **/usr/lib/terminfo/a/att4425**. 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-bit byte is assumed, but no assumptions about byte ordering or sign extension are made. Thus, these binary *terminfo*(4) files can be transported to other hardware with 8-bit bytes.

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**, and the value **–2** is represented by **0376,0377**; other negative values are illegal. Computers where this does not correspond to the hardware read the integers as two bytes and compute the result, making the compiled entries portable between machine types. The **–1** generally means that a capability is missing from this terminal. The **–2** means that the capability has been cancelled in the *terminfo*(4) source and also is to be considered missing.

The compiled file is created from the source file descriptions of the terminals (see the **–I** option of *infocmp*(1M)) by using the *terminfo*(4) compiler, *tic*(1M), and read by the routine **setupterm**( ). (See *curses*(3X).) The file is divided into 6 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.

The terminal names section comes next. It contains the first line of the *terminfo*(4) description, listing the various names for the terminal, separated by the bar (|) character (see *term*(5)). 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 value of **2** means that the flag has been cancelled. 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 boolean flags section. Each capability takes up two bytes, and is stored as a short integer. If the value represented is **–1** or **–2**, 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** or **–2** 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 AT&T Model 37 KSR is included:

```
37|tty37|AT&T model 37 teletype,
        hc, os, xon,
        bel=^G, cr=\r, cub1=\b, cud1=\n, cuu1=\E7, hd=\E9,
        hu=\E8, ind=\n,

0000000 032 001      \0 032  \0 013  \0 021 001   3  \0   3   7   |   t
0000020   t   y   3   7   |   A   T   &   T       m   o   d   e   l
0000040   3   7       t   e   l   e   t   y   p   e  \0  \0  \0  \0  \0
0000060  \0  \0  \0 001  \0  \0  \0  \0  \0  \0  \0 001  \0  \0  \0  \0
0000100 001  \0  \0  \0  \0  \0 377 377 377 377 377 377 377 377 377 377
0000120 377 377 377 377 377 377 377 377 377 377 377 377 377 377   &  \0
0000140      \0 377 377 377 377 377 377 377 377 377 377 377 377 377 377
0000160 377 377   "  \0 377 377 377 377   (  \0 377 377 377 377 377 377
0000200 377 377   0  \0 377 377 377 377 377 377 377 377   -  \0 377 377
0000220 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0000520 377 377 377 377 377 377 377 377 377 377 377 377 377 377   $  \0
0000540 377 377 377 377 377 377 377 377 377 377 377 377 377 377   *  \0
0000560 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0001160 377 377 377 377 377 377 377 377 377 377 377 377 377 377   3   7
0001200   |   t   t   y   3   7   |   A   T   &   T       m   o   d   e
0001220   l       3   7       t   e   l   e   t   y   p   e  \0  \r  \0
0001240  \n  \0  \n  \0 007  \0  \b  \0 033   8  \0 033   9  \0 033   7
0001260  \0  \0
0001261
```

Some limitations: total compiled entries cannot exceed 4096 bytes; all entries in the name field cannot exceed 128 bytes.

FILES

/usr/lib/terminfo/?/*	compiled terminal description database
/usr/include/term.h	*terminfo*(4) header file

SEE ALSO

curses(3X), terminfo(4), term(5).
infocmp(1M) in the *System Administrator's Reference Manual*.
Chapter 10 of the *Programmer's Guide*.

NAME

terminfo – terminal capability data base

SYNOPSIS

/usr/lib/terminfo/?/*

DESCRIPTION

terminfo is a database, produced by *tic*(1M), that describes the capabilities of devices, e.g., terminals and printers. Devices are described in *terminfo* source files by specifying a set of capabilities, quantifying certain aspects of the device, and specifying character sequences that effect particular results. This database is often used by screen oriented applications, e.g., *vi*(1) and *curses*(3X), as well as by some operating system commands, e.g., *ls*(1) and *pg*(1). This usage allows them to work with a variety of devices without changes to the programs. To obtain the source description for a device, use the *infocmp*(1M) command.

terminfo source files consist of one or more device descriptions. Each description consists of a header (beginning in column 1) and one or more lines that list the features for that particular device. Every line in a *terminfo* source file must end in a comma (,). Every line in a *terminfo* source file except the header must be indented with one or more white spaces (either spaces or tabs).

Entries in *terminfo* source files consist of a number of comma-separated fields. White space after each comma is ignored. Embedded commas must be escaped by using a backslash. The following example shows the format of a *terminfo* source file:

Column 1

↓

alias$_1$ | *alias*$_2$ | ... | *alias*$_n$ | *longname*,
<white space> am, lines #24,
<white space> home=Eeh,

The first line, commonly referred to as the header line, must begin in column one and it must contain at least two aliases, separated by vertical bars. The last field in the header line must be the long name of the device and it may contain any string. Alias names must be unique in the *terminfo* database and they must conform to the operating system file naming conventions (see *tic*(1M)); they cannot, for example, contain white space or slashes.

Every device must be assigned a name, e.g., "att5425" (for the AT&T model 5425 device). Choose device names (except the long name) that do not contain hyphens; hyphens are reserved for use when adding suffixes that indicate special modes.

These special modes may be modes that the hardware can be in, or user preferences. To assign a special mode to a particular device, append a suffix, consisting of a hyphen and an indicator of the mode, to the device name. For example, the **-w** suffix means "wide mode"; when specified, it allows for a width of 132 columns instead of the standard 80 columns. Therefore, if you want to use an AT&T 5425 device set to wide mode, name the device "att5425-w." Use the following suffixes where possible:

Suffix	Meaning	Example
-w	Wide mode (more than 80 columns)	5410-w
-am	With auto. margins (usually default)	vt100-am
-nam	Without automatic margins	vt100-nam
-*n*	Number of lines on the screen	2300-40
-na	No arrow keys (leave them in local)	c100-na
*n*p	Number of pages of memory	c100-4p
-rv	Reverse video	4415-rv

The *terminfo* reference manual page is organized in two sections: *DEVICE CAPABILITIES* and *PRINTER CAPABILITIES*.

PART 1: DEVICE CAPABILITIES

Capabilities in *terminfo* are of three types: Boolean capabilities (show that a device has or does not have a particular feature), numeric capabilities (quantify particular features of a device), and string capabilities (provide sequences that can be used to perform particular operations on devices).

In the following tables, a **Variable** is the name by which a C programmer accesses a capability (at the *terminfo* level). A **Capname** is the short name for a capability specified in the *terminfo* source file. It is used by a person updating the source file and by the *tput*(1) command. A **Termcap Code** is a two-letter sequence that corresponds to the *termcap* capability name. (Note that *termcap* is no longer supported.)

Capability names have no hard length limit, but an informal limit of five characters has been adopted to keep them short. Whenever possible, capability names are chosen to be the same as or similar to those specified by the ANSI X3.64-1979 standard. Semantics are also intended to match those of the ANSI standard.

All the following string capabilities may have padding specified, with the exception of those used for input. Input capabilities, listed under the **Strings** section in the following tables, have names beginning with **key_**. The *#i* symbol in the description field of the following tables refers to the i^{th} parameter.

Booleans:

Variable	Cap-name	Termcap Code	Description
auto_left_margin	bw	bw	**cub1** wraps from column 0 to last column
auto_right_margin	am	am	Terminal has automatic margins
back_color_erase	bce	be	Screen erased with background color
can_change	ccc	cc	Terminal can re-define existing color
ceol_standout_glitch	xhp	xs	Standout not erased by overwriting (hp)
col_addr_glitch	xhpa	YA	Only positive motion for **hpa/mhpa** caps
cpi_changes_res	cpix	YF	Changing character pitch changes resolution
cr_cancels_micro_mode	crxm	YB	Using **cr** turns off micro mode
eat_newline_glitch	xenl	xn	Newline ignored after 80 columns (*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
hard_cursor	chts	HC	Cursor is hard to see
has_meta_key	km	km	Has a meta key (shift, sets parity bit)
has_print_wheel	daisy	YC	Printer needs operator to change character set
has_status_line	hs	hs	Has extra "status line"
hue_lightness_saturation	hls	hl	Terminal uses only HLS color notation (Tektronix)
insert_null_glitch	in	in	Insert mode distinguishes nulls
lpi_changes_res	lpix	YG	Changing line pitch changes resolution
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
needs_xon_xoff	nxon	nx	Padding won't work, xon/xoff required
no_esc_ctlc	xsb	xb	Beehive (f1=escape, f2=ctrl C
non_rev_rmcup	nrrmc	NR	**smcup** does not reverse **rmcup**
no_pad_char	npc	NP	Pad character doesn't exist
over_strike	os	os	Terminal overstrikes on hard-copy terminal
prtr_silent	mc5i	5i	Printer won't echo on screen
row_addr_glitch	xvpa	YD	Only positive motion for **vpa/mvpa** caps
semi_auto_right_margin	sam	YE	Printing in last column causes **cr**

status_line_esc_ok	eslok	es	Escape can be used on the status line
dest_tabs_magic_smso	xt	xt	Destructive tabs, magic **smso** char (t1061)
tilde_glitch	hz	hz	Hazeltine; can't print tilde (~)
transparent_underline	ul	ul	Underline character overstrikes
xon_xoff	xon	xo	Terminal uses xon/xoff handshaking

Numbers:

Variable	Cap-name	Termcap Code	Description
buffer_capacity	bufsz	Ya	Number of bytes buffered before printing
columns	cols	co	Number of columns in a line
dot_vert_spacing	spinv	Yb	Spacing of pins vertically in pins per inch
dot_horz_spacing	spinh	Yc	Spacing of dots horizontally in dots per inch
init_tabs	it	it	Tabs initially every # spaces
label_height	lh	lh	Number of rows in each label
label_width	lw	lw	Number of columns in each label
lines	lines	li	Number of lines on a screen or a page
lines_of_memory	lm	lm	Lines of memory if > **lines**; 0 means varies
magic_cookie_glitch	xmc	sg	Number of blank characters left by **smso** or **rmso**
max_colors	colors	Co	Maximum number of colors on the screen
max_micro_address	maddr	Yd	Maximum value in **micro_..._address**
max_micro_jump	mjump	Ye	Maximum value in **parm_..._micro**
max_pairs	pairs	pa	Maximum number of color-pairs on the screen
micro_col_size	mcs	Yf	Character step size when in micro mode
micro_line_size	mls	Yg	Line step size when in micro mode
no_color_video	ncv	NC	Video attributes that can't be used with colors
number_of_pins	npins	Yh	Number of pins in print-head
num_labels	nlab	Nl	Number of labels on screen (start at 1)
output_res_char	orc	Yi	Horizontal resolution in units per character
output_res_line	orl	Yj	Vertical resolution in units per line
output_res_horz_inch	orhi	Yk	Horizontal resolution in units per inch
output_res_vert_inch	orvi	Yl	Vertical resolution in units per inch
padding_baud_rate	pb	pb	Lowest baud rate where padding needed
virtual_terminal	vt	vt	Virtual terminal number (SYSTEM V/88)
wide_char_size	widcs	Yn	Character step size when in double wide mode
width_status_line	wsl	ws	Number of columns in status line

Strings:

Variable	Cap-name	Termcap Code	Description
acs_chars	acsc	ac	Graphic charset pairs aAbBcC - def=vt100
back_tab	cbt	bt	Back tab
bell	bel	bl	Audible signal (bell)
carriage_return	cr	cr	Carriage return
change_char_pitch	cpi	ZA	Change number of characters per inch
change_line_pitch	lpi	ZB	Change number of lines per inch
change_res_horz	chr	ZC	Change horizontal resolution
change_res_vert	cvr	ZD	Change vertical resolution
change_scroll_region	csr	cs	Change to lines #1 through #2 (vt100)
char_padding	rmp	rP	Like ip but when in replace mode
char_set_names	csnm	Zy	List of character set names
clear_all_tabs	tbc	ct	Clear all tab stops
clear_margins	mgc	MC	Clear all margins (top, bottom, and sides)
clear_screen	clear	cl	Clear screen and home cursor
clr_bol	el1	cb	Clear to beginning of line, inclusive
clr_eol	el	ce	Clear to end of line
clr_eos	ed	cd	Clear to end of display
column_address	hpa	ch	Horizontal position absolute
command_character	cmdch	CC	Terminal settable cmd character in prototype
cursor_address	cup	cm	Move to row #1 col #2
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 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 or carriage 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
define_char	defc	ZE	Define a character in a character set†
delete_character	dch1	dc	Delete character
delete_line	dl1	dl	Delete line
dis_status_line	dsl	ds	Disable status line
down_half_line	hd	hd	Half-line down (forward 1/2 linefeed)
ena_acs	enacs	eA	Enable alternate character set
enter_alt_charset_mode	smacs	as	Start alternate character set

enter_am_mode	smam	SA	Turn on automatic margins
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_doublewide_mode	swidm	ZF	Enable double wide printing
enter_draft_quality	sdrfq	ZG	Set draft quality print
enter_insert_mode	smir	im	Insert mode (enter)
enter_italics_mode	sitm	ZH	Enable italics
enter_leftward_mode	slm	ZI	Enable leftward carriage motion
enter_micro_mode	smicm	ZJ	Enable micro motion capabilities
enter_near_letter_quality	snlq	ZK	Set near-letter quality print
enter_normal_quality	snrmq	ZL	Set normal quality print
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 (characters invisible)
enter_shadow_mode	sshm	ZM	Enable shadow printing
enter_standout_mode	smso	so	Begin standout mode
enter_subscript_mode	ssubm	ZN	Enable subscript printing
enter_superscript_mode	ssupm	ZO	Enable superscript printing
enter_underline_mode	smul	us	Start underscore mode
enter_upward_mode	sum	ZP	Enable upward carriage motion
enter_xon_mode	smxon	SX	Turn on xon/xoff handshaking
erase_chars	ech	ec	Erase #1 characters
exit_alt_charset_mode	rmacs	ae	End alternate character set
exit_am_mode	rmam	RA	Turn off automatic margins
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_doublewide_mode	rwidm	ZQ	Disable double wide printing
exit_insert_mode	rmir	ei	End insert mode
exit_italics_mode	ritm	ZR	Disable italics
exit_leftward_mode	rlm	ZS	Enable rightward (normal) carriage motion
exit_micro_mode	rmicm	ZT	Disable micro motion capabilities
exit_shadow_mode	rshm	ZU	Disable shadow printing
exit_standout_mode	rmso	se	End standout mode
exit_subscript_mode	rsubm	ZV	Disable subscript printing
exit_superscript_mode	rsupm	ZW	Disable superscript printing
exit_underline_mode	rmul	ue	End underscore mode

exit_upward_mode	rum	ZX	Enable downward (normal) carriage motion
exit_xon_mode	rmxon	RX	Turn off xon/xoff handshaking
flash_screen	flash	vb	Visible bell (may not move cursor)
form_feed	ff	ff	Hardcopy terminal page eject
from_status_line	fsl	fs	Return from status line
init_1string	is1	i1	Terminal or printer initialization string
init_2string	is2	is	Terminal or printer initialization string
init_3string	is3	i3	Terminal or printer initialization string
init_file	if	if	Name of initialization file
init_prog	iprog	iP	Path name of program for initialization
initialize_color	initc	Ic	Initialize the definition of color
initialize_pair	initp	Ip	Initialize color-pair
insert_character	ich1	ic	Insert character
insert_line	il1	al	Add new blank line
insert_padding	ip	ip	Insert pad after character inserted
key_a1	ka1	K1	KEY_A1, 0534, upper left of keypad
key_a3	ka3	K3	KEY_A3, 0535, upper right of keypad
key_b2	kb2	K2	KEY_B2, 0536, center of keypad
key_backspace	kbs	kb	KEY_BACKSPACE, 0407, sent by backspace key
key_beg	kbeg	@1	KEY_BEG, 0542, sent by beg(inning) key
key_btab	kcbt	kB	KEY_BTAB, 0541, sent by back-tab key
key_c1	kc1	K4	KEY_C1, 0537, lower left of keypad
key_c3	kc3	K5	KEY_C3, 0540, lower right of keypad
key_cancel	kcan	@2	KEY_CANCEL, 0543, sent by cancel key
key_catab	ktbc	ka	KEY_CATAB, 0526, sent by clear-all-tabs key
key_clear	kclr	kC	KEY_CLEAR, 0515, sent by clear-screen or erase key
key_close	kclo	@3	KEY_CLOSE, 0544, sent by close key
key_command	kcmd	@4	KEY_COMMAND, 0545, sent by cmd (command) key
key_copy	kcpy	@5	KEY_COPY, 0546, sent by copy key
key_create	kcrt	@6	KEY_CREATE, 0547, sent by create key
key_ctab	kctab	kt	KEY_CTAB, 0525, sent by clear-tab key
key_dc	kdch1	kD	KEY_DC, 0512, sent by delete-character key
key_dl	kdl1	kL	KEY_DL, 0510, sent by delete-line key
key_down	kcud1	kd	KEY_DOWN, 0402, sent by terminal down-arrow key
key_eic	krmir	kM	KEY_EIC, 0514, sent by **rmir** or **smir** in insert mode
key_end	kend	@7	KEY_END, 0550, sent by end key
key_enter	kent	@8	KEY_ENTER, 0527, sent by enter/send key
key_eol	kel	kE	KEY_EOL, 0517, sent by clear-to-end-of-line key
key_eos	ked	kS	KEY_EOS, 0516, sent by clear-to-end-of-screen key
key_exit	kext	@9	KEY_EXIT, 0551, sent by exit key

key_f0	kf0	k0	KEY_F(0), 0410, sent by function key f0
key_f1	kf1	k1	KEY_F(1), 0411, sent by function key f1
key_f2	kf2	k2	KEY_F(2), 0412, sent by function key f2
key_f3	kf3	k3	KEY_F(3), 0413, sent by function key f3
key_f4	kf4	k4	KEY_F(4), 0414, sent by function key f4
key_f5	kf5	k5	KEY_F(5), 0415, sent by function key f5
key_f6	kf6	k6	KEY_F(6), 0416, sent by function key f6
key_f7	kf7	k7	KEY_F(7), 0417, sent by function key f7
key_f8	kf8	k8	KEY_F(8), 0420, sent by function key f8
key_f9	kf9	k9	KEY_F(9), 0421, sent by function key f9
key_f10	kf10	k;	KEY_F(10), 0422, sent by function key f10
key_f11	kf11	F1	KEY_F(11), 0423, sent by function key f11
key_f12	kf12	F2	KEY_F(12), 0424, sent by function key f12
key_f13	kf13	F3	KEY_F(13), 0425, sent by function key f13
key_f14	kf14	F4	KEY_F(14), 0426, sent by function key f14
key_f15	kf15	F5	KEY_F(15), 0427, sent by function key f15
key_f16	kf16	F6	KEY_F(16), 0430, sent by function key f16
key_f17	kf17	F7	KEY_F(17), 0431, sent by function key f17
key_f18	kf18	F8	KEY_F(18), 0432, sent by function key f18
key_f19	kf19	F9	KEY_F(19), 0433, sent by function key f19
key_f20	kf20	FA	KEY_F(20), 0434, sent by function key f20
key_f21	kf21	FB	KEY_F(21), 0435, sent by function key f21
key_f22	kf22	FC	KEY_F(22), 0436, sent by function key f22
key_f23	kf23	FD	KEY_F(23), 0437, sent by function key f23
key_f24	kf24	FE	KEY_F(24), 0440, sent by function key f24
key_f25	kf25	FF	KEY_F(25), 0441, sent by function key f25
key_f26	kf26	FG	KEY_F(26), 0442, sent by function key f26
key_f27	kf27	FH	KEY_F(27), 0443, sent by function key f27
key_f28	kf28	FI	KEY_F(28), 0444, sent by function key f28
key_f29	kf29	FJ	KEY_F(29), 0445, sent by function key f29
key_f30	kf30	FK	KEY_F(30), 0446, sent by function key f30
key_f31	kf31	FL	KEY_F(31), 0447, sent by function key f31
key_f32	kf32	FM	KEY_F(32), 0450, sent by function key f32
key_f33	kf33	FN	KEY_F(13), 0451, sent by function key f13
key_f34	kf34	FO	KEY_F(34), 0452, sent by function key f34
key_f35	kf35	FP	KEY_F(35), 0453, sent by function key f35
key_f36	kf36	FQ	KEY_F(36), 0454, sent by function key f36
key_f37	kf37	FR	KEY_F(37), 0455, sent by function key f37
key_f38	kf38	FS	KEY_F(38), 0456, sent by function key f38
key_f39	kf39	FT	KEY_F(39), 0457, sent by function key f39

key_f40	kf40	FU	KEY_F(40), 0460, sent by function key f40
key_f41	kf41	FV	KEY_F(41), 0461, sent by function key f41
key_f42	kf42	FW	KEY_F(42), 0462, sent by function key f42
key_f43	kf43	FX	KEY_F(43), 0463, sent by function key f43
key_f44	kf44	FY	KEY_F(44), 0464, sent by function key f44
key_f45	kf45	FZ	KEY_F(45), 0465, sent by function key f45
key_f46	kf46	Fa	KEY_F(46), 0466, sent by function key f46
key_f47	kf47	Fb	KEY_F(47), 0467, sent by function key f47
key_f48	kf48	Fc	KEY_F(48), 0470, sent by function key f48
key_f49	kf49	Fd	KEY_F(49), 0471, sent by function key f49
key_f50	kf50	Fe	KEY_F(50), 0472, sent by function key f50
key_f51	kf51	Ff	KEY_F(51), 0473, sent by function key f51
key_f52	kf52	Fg	KEY_F(52), 0474, sent by function key f52
key_f53	kf53	Fh	KEY_F(53), 0475, sent by function key f53
key_f54	kf54	Fi	KEY_F(54), 0476, sent by function key f54
key_f55	kf55	Fj	KEY_F(55), 0477, sent by function key f55
key_f56	kf56	Fk	KEY_F(56), 0500, sent by function key f56
key_f57	kf57	Fl	KEY_F(57), 0501, sent by function key f57
key_f58	kf58	Fm	KEY_F(58), 0502, sent by function key f58
key_f59	kf59	Fn	KEY_F(59), 0503, sent by function key f59
key_f60	kf60	Fo	KEY_F(60), 0504, sent by function key f60
key_f61	kf61	Fp	KEY_F(61), 0505, sent by function key f61
key_f62	kf62	Fq	KEY_F(62), 0506, sent by function key f62
key_f63	kf63	Fr	KEY_F(63), 0507, sent by function key f63
key_find	kfnd	@0	KEY_FIND, 0552, sent by find key
key_help	khlp	%1	KEY_HELP, 0553, sent by help key
key_home	khome	kh	KEY_HOME, 0406, sent by home key
key_ic	kich1	kI	KEY_IC, 0513, sent by ins-char/enter ins-mode key
key_il	kil1	kA	KEY_IL, 0511, sent by insert-line key
key_left	kcub1	kl	KEY_LEFT, 0404, sent by terminal left-arrow key
key_ll	kll	kH	KEY_LL, 0533, sent by home-down key
key_mark	kmrk	%2	KEY_MARK, 0554, sent by mark key
key_message	kmsg	%3	KEY_MESSAGE, 0555, sent by message key
key_move	kmov	%4	KEY_MOVE, 0556, sent by move key
key_next	knxt	%5	KEY_NEXT, 0557, sent by next-object key
key_npage	knp	kN	KEY_NPAGE, 0522, sent by next-page key
key_open	kopn	%6	KEY_OPEN, 0560, sent by open key
key_options	kopt	%7	KEY_OPTIONS, 0561, sent by options key
key_ppage	kpp	kP	KEY_PPAGE, 0523, sent by previous-page key
key_previous	kprv	%8	KEY_PREVIOUS, 0562, sent by previous-object key

key_print	kprt	%9	KEY_PRINT, 0532, sent by print or copy key
key_redo	krdo	%0	KEY_REDO, 0563, sent by redo key
key_reference	kref	&1	KEY_REFERENCE, 0564, sent by ref(erence) key
key_refresh	krfr	&2	KEY_REFRESH, 0565, sent by refresh key
key_replace	krpl	&3	KEY_REPLACE, 0566, sent by replace key
key_restart	krst	&4	KEY_RESTART, 0567, sent by restart key
key_resume	kres	&5	KEY_RESUME, 0570, sent by resume key
key_right	kcuf1	kr	KEY_RIGHT, 0405, sent by terminal right-arrow key
key_save	ksav	&6	KEY_SAVE, 0571, sent by save key
key_sbeg	kBEG	&9	KEY_SBEG, 0572, sent by shifted beginning key
key_scancel	kCAN	&0	KEY_SCANCEL, 0573, sent by shifted cancel key
key_scommand	kCMD	*1	KEY_SCOMMAND, 0574, sent by shifted command key
key_scopy	kCPY	*2	KEY_SCOPY, 0575, sent by shifted copy key
key_screate	kCRT	*3	KEY_SCREATE, 0576, sent by shifted create key
key_sdc	kDC	*4	KEY_SDC, 0577, sent by shifted delete-char key
key_sdl	kDL	*5	KEY_SDL, 0600, sent by shifted delete-line key
key_select	kslt	*6	KEY_SELECT, 0601, sent by select key
key_send	kEND	*7	KEY_SEND, 0602, sent by shifted end key
key_seol	kEOL	*8	KEY_SEOL, 0603, sent by shifted clear-line key
key_sexit	kEXT	*9	KEY_SEXIT, 0604, sent by shifted exit key
key_sf	kind	kF	KEY_SF, 0520, sent by scroll-forward/down key
key_sfind	kFND	*0	KEY_SFIND, 0605, sent by shifted find key
key_shelp	kHLP	#1	KEY_SHELP, 0606, sent by shifted help key
key_shome	kHOM	#2	KEY_SHOME, 0607, sent by shifted home key
key_sic	kIC	#3	KEY_SIC, 0610, sent by shifted input key
key_sleft	kLFT	#4	KEY_SLEFT, 0611, sent by shifted left-arrow key
key_smessage	kMSG	%a	KEY_SMESSAGE, 0612, sent by shifted message key
key_smove	kMOV	%b	KEY_SMOVE, 0613, sent by shifted move key
key_snext	kNXT	%c	KEY_SNEXT, 0614, sent by shifted next key
key_soptions	kOPT	%d	KEY_SOPTIONS, 0615, sent by shifted options key
key_sprevious	kPRV	%e	KEY_SPREVIOUS, 0616, sent by shifted prev key
key_sprint	kPRT	%f	KEY_SPRINT, 0617, sent by shifted print key
key_sr	kri	kR	KEY_SR, 0521, sent by scroll-backward/up key
key_sredo	kRDO	%g	KEY_SREDO, 0620, sent by shifted redo key
key_sreplace	kRPL	%h	KEY_SREPLACE, 0621, sent by shifted replace key
key_sright	kRIT	%i	KEY_SRIGHT, 0622, sent by shifted right-arrow key
key_srsume	kRES	%j	KEY_SRSUME, 0623, sent by shifted resume key
key_ssave	kSAV	!1	KEY_SSAVE, 0624, sent by shifted save key
key_ssuspend	kSPD	!2	KEY_SSUSPEND, 0625, sent by shifted suspend key
key_stab	khts	kT	KEY_STAB, 0524, sent by set-tab key

key_sundo	kUND	!3	KEY_SUNDO, 0626, sent by shifted undo key
key_suspend	kspd	&7	KEY_SUSPEND, 0627, sent by suspend key
key_undo	kund	&8	KEY_UNDO, 0630, sent by undo key
key_up	kcuu1	ku	KEY_UP, 0403, 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_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
lab_f10	lf10	la	Labels on function key f10 if not f10
label_off	rmln	LF	Turn off soft labels
label_on	smln	LO	Turn on soft labels
meta_off	rmm	mo	Turn off "meta mode"
meta_on	smm	mm	Turn on "meta mode" (8th bit)
micro_column_address	mhpa	ZY	Like **column_address** for micro adjustment
micro_down	mcud1	ZZ	Like **cursor_down** for micro adjustment
micro_left	mcub1	Za	Like **cursor_left** for micro adjustment
micro_right	mcuf1	Zb	Like **cursor_right** for micro adjustment
micro_row_address	mvpa	Zc	Like **row_address** for micro adjustment
micro_up	mcuu1	Zd	Like **cursor_up** for micro adjustment
newline	nel	nw	Newline (behaves like cr followed by lf)
order_of_pins	porder	Ze	Matches software bits to print-head pins
orig_colors	oc	oc	Set all color(-pair)s to the original ones
orig_pair	op	op	Set default color-pair to the original one
pad_char	pad	pc	Pad character (instead of null)
parm_dch	dch	DC	Delete #1 chars
parm_delete_line	dl	DL	Delete #1 lines
parm_down_cursor	cud	DO	Move down #1 lines
parm_down_micro	mcud	Zf	Like **parm_down_cursor** for micro adjust
parm_ich	ich	IC	Insert #1 blank chars
parm_index	indn	SF	Scroll forward #1 lines
parm_insert_line	il	AL	Add #1 new blank lines
parm_left_cursor	cub	LE	Move cursor left #1 spaces

parm_left_micro	mcub	Zg	Like **parm_left_cursor** for micro adjust
parm_right_cursor	cuf	RI	Move right #1 spaces
parm_right_micro	mcuf	Zh	Like **parm_right_cursor** for micro adjust
parm_rindex	rin	SR	Scroll backward #1 lines
parm_up_cursor	cuu	UP	Move cursor up #1 lines
parm_up_micro	mcuu	Zi	Like **parm_up_cursor** for micro adjust
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
plab_norm	pln	pn	Prog label #1 to show string #2
print_screen	mc0	ps	Print contents of the screen
prtr_non	mc5p	pO	Turn on the printer for #1 bytes
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
req_for_input	rfi	RF	Send next input char (for ptys)
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
save_cursor	sc	sc	Save cursor position
scroll_forward	ind	sf	Scroll text up
scroll_reverse	ri	sr	Scroll text down
select_char_set	scs	Zj	Select character set
set_attributes	sgr	sa	Define the video attributes #1-#9
set_background	setb	Sb	Set current background color
set_bottom_margin	smgb	Zk	Set bottom margin at current line
set_bottom_margin_parm	smgbp	Zl	Set bottom margin at line #1 or #2 lines from bottom
set_color_pair	scp	sp	Set current color-pair
set_foreground	setf	Sf	Set current foreground color1
set_left_margin	smgl	ML	Set left margin at current line
set_left_margin_parm	smglp	Zm	Set left (right) margin at column #1 (#2)
set_right_margin	smgr	MR	Set right margin at current column
set_right_margin_parm	smgrp	Zn	Set right margin at column #1
set_tab	hts	st	Set a tab in all rows, current column
set_top_margin	smgt	Zo	Set top margin at current line
set_top_margin_parm	smgtp	Zp	Set top (bottom) margin at line #1 (#2)
set_window	wind	wi	Current window is lines #1-#2 cols #3-#4

start_bit_image	sbim	Zq	Start printing bit image graphics
start_char_set_def	scsd	Zr	Start definition of a character set
stop_bit_image	rbim	Zs	End printing bit image graphics
stop_char_set_def	rcsd	Zt	End definition of a character set
subscript_characters	subcs	Zu	List of "subscript-able" characters
superscript_characters	supcs	Zv	List of "superscript-able" characters
tab	ht	ta	Tab to next 8-space hardware tab stop
these_cause_cr	docr	Zw	Printing any of these chars causes cr
to_status_line	tsl	ts	Go to status line, col #1
underline_char	uc	uc	Underscore one char and move past it
up_half_line	hu	hu	Half-line up (reverse 1/2 linefeed)
xoff_character	xoffc	XF	X-off character
xon_character	xonc	XN	X-on character
zero_motion	zerom	Zx	No motion for the subsequent characte

Booleans:

Cap-Name	Variable	Termcap Code	Description
am	auto_right_margin	am	Terminal has automatic margins
bw	auto_left_margin	bw	cub1 wraps from column 0 to last column
ccc	can_change	cc	Terminal can re-define existing color
chts	hard_cursor	HC	Cursor is hard to see
cpix	cpi_changes_res	YF	Changing character pitch changes resolution
crxm	cr_cancels_micro_modem	YB	Using cr turns off micro mode
da	memory_above	da	Display may be retained above the screen
daisy	has_print_wheel	YC	Printer needs operator to change character set
db	memory_below	db	Display may be retained below the screen
eo	erase_overstrike	eo	Can erase overstrikes with a blank
eslok	status_line_esc_ok	es	Escape can be used on the status line
gn	generic_type	gn	Generic line type (e.g., dialup, switch)
hc	hard_copy	hc	Hardcopy terminal
hls	hue_lightness_saturation	hl	Terminal uses only HLS color notation (Tektronix)
hs	has_status_line	hs	Has extra "status line"
hz	tilde_glitch	hz	Hazeltine; can't print tilde (~)
in	insert_null_glitch	in	Insert mode distinguishes nulls
km	has_meta_key	km	Has a meta key (shift, sets parity bit)
lpix	lpi_changes_res	YG	Changing line pitch changes resolution
mc5i	prtr_silent	5i	Printer won't echo on screen
mir	move_insert_mode	mi	Safe to move while in insert mode
msgr	move_standout_mode	ms	Safe to move in standout modes

npc	no_pad_char	NP	Pad character doesn't exist
nrrmc	non_rev_rmcup	NR	**smcup** does not reverse **rmcup**
nxon	needs_xon_xoff	nx	Padding won't work, xon/xoff required
os	over_strike	os	Terminal overstrikes on hard-copy terminal
sam	semi_auto_right_margin	YE	Printing in last column causes **cr**
ul	transparent_underline	ul	Underline character overstrikes
xenl	eat_newline_glitch	xn	Newline ignored after 80 columns (*Concept*)
xhp	ceol_standout_glitch	xs	Standout not erased by overwriting (hp)
xhpa	col_addr_glitch	YA	Only positive motion for **hpa/mhpa** caps
xon	xon_xoff	xo	Terminal uses xon/xoff handshaking
xsb	no_esc_ctlc	xb	Beehive (f1=escape, f2=ctrl C)
xt	dest_tabs_magic_smso	xt	Destructive tabs, magic **smso** char (t1061)
xvpa	row_addr_glitch	YD	Only positive motion for **vpa/mvpa** caps

Numbers:

Cap-name	Variable	Termcap Code	Description
bufsz	buffer_capacity	Ya	Number of bytes buffered before printing
colors	max_colors	Co	Maximum number of colors on the screen
cols	columns	co	Number of columns in a line
cps	print_rate	Ym	Average print rate in characters per second
it	init_tabs	it	Tabs initially every # spaces
lh	label_height	lh	Number of rows in each label
lines	lines	li	Number of lines on a screen or a page
lm	lines_of_memory	lm	Lines of memory if > **lines**; 0 means varies
lw	label_width	lw	Number of columns in each label
maddr	max_micro_address	Yd	Maximum value in **micro_..._address**
mcs	micro_col_size	Yf	Character step size when in micro mode
mjump	max_micro_jump	Ye	Maximum value in **parm_..._micro**
mls	micro_line_size	Yg	Line step size when in micro mode
ncv	no_color_video	NC	Video attributes that can't be used with colors
nlab	num_labels	Nl	Number of labels on screen (start at 1)
npins	number_of_pins	Yh	Number of pins in print-head
orc	output_res_char	Yi	Horizontal resolution in units per character
orhi	output_res_horz_inch	Yk	Horizontal resolution in units per inch
orl	output_res_line	Yj	Vertical resolution in units per line
orvi	output_res_vert_inch	Yl	Vertical resolution in units per inch
pairs	max_pairs	pa	Maximum number of color-pairs on the screen
pb	padding_baud_rate	pb	Lowest baud rate where padding needed
spinh	dot_horz_spacing	Yc	Spacing of dots horizontally in dots per inch

spinv	dot_vert_spacing	Yb	Spacing of pins vertically in pins per inch
vt	virtual_terminal	vt	Virtual terminal number (SYSTEM V/88)
widcs	wide_char_size	Yn	Character step size when in double wide mode
wsl	width_status_line	ws	Number of columns in status line
xmc	magic_cookie_glitch	sg	Number of blank characters left by **smso** or **rmso**

Strings:

Cap-name	Variable	Termcap Code	Description
acsc	acs_chars	ac	Graphic charset pairs aAbBcC - def=vt100
bel	bell	bl	Audible signal (bell)
blink	enter_blink_mode	mb	Turn on blinking
bold	enter_bold_mode	md	Turn on bold (extra bright) mode
cbt	back_tab	bt	Back tab
chr	change_res_horz	ZC	Change horizontal resolution
civis	cursor_invisible	vi	Make cursor invisible
clear	clear_screen	cl	Clear screen and home cursor
cmdch	command_character	CC	Terminal settable cmd character in prototype
cnorm	cursor_normal	ve	Make cursor appear normal (undo **vs/vi**)
cpi	change_char_pitch	ZA	Change number of characters per inch
cr	carriage_return	cr	Carriage return
csnm	char_set_names	Zy	List of character set names
csr	change_scroll_region	cs	Change to lines #1 through #2 (vt100)
cub	parm_left_cursor	LE	Move cursor left #1 spaces
cub1	cursor_left	le	Move left one space.
cud	parm_down_cursor	DO	Move down #1 lines.
cuf	parm_right_cursor	RI	Move right #1 spaces.
cuf1	cursor_right	nd	Non-destructive space (cursor or carriage right)
cup	cursor_address	cm	Move to row #1 col #2
cuu	parm_up_cursor	UP	Move cursor up #1 lines.
cvr	change_res_vert	ZD	Change vertical resolution
cvvis	cursor_visible	vs	Make cursor very visible
dch	parm_dch	DC	Delete #1 chars
dch1	delete_character	dc	Delete character
defc	define_char	ZE	Define a character in a character set
dim	enter_dim_mode	mh	Turn on half-bright mode
dl	delete_line	dl1	Delete line
dl	parm_delete_line	DL	Delete #1 lines
do	cursor_down	do	Down one line
docr	these_cause_cr	Zw	Printing any of these chars causes **cr**

dsl	dis_status_line	ds	Disable status line
ech	erase_chars	ec	Erase #1 characters
ed	clr_eos	cd	Clear to end of display
el	clr_eol	ce	Clear to end of line
el1	clr_bol	cb	Clear to beginning of line, inclusive
enacs	ena_acs	eA	Enable alternate character set
ff	form_feed	ff	Hardcopy terminal page eject
flash	flash_screen	vb	Visible bell (may not move cursor)
fsl	from_status_line	fs	Return from status line
hd	down_half_line	hd	Half-line down (forward 1/2 linefeed)
home	cursor_home	ho	Home cursor (if no cup)
hpa	column_address	ch	Horizontal position absolute
ht	tab	ta	Tab to next 8-space hardware tab stop
hts	set_tab	st	Set a tab in all rows, current column
hu	up_half_line	hu	Half-line up (reverse 1/2 linefeed)
ich	parm_ich	IC	Insert #1 blank chars
ich1	insert_character	ic	Insert character
if	init_file	if	Name of initialization file
il	parm_insert_line	AL	Add #1 new blank lines
il1	insert_line	al	Add new blank line
ind	scroll_forward	sf	Scroll text up
indn	parm_index	SF	Scroll forward #1 lines
initc	initialize_color	Ic	Initialize the definition of color
initp	initialize_pair	Ip	Initialize color-pair
invis	enter_secure_mode	mk	Turn on blank mode (characters invisible)
ip	insert_padding	ip	Insert pad after character inserted
iprog	init_prog	iP	Path name of program for initialization
is1	init_1string	i1	Terminal or printer initialization string
is2	init_2string	is	Terminal or printer initialization string
is3	init_3string	i3	Terminal or printer initialization string
kBEG	key_sbeg	&9	KEY_SBEG, 0572, sent by shifted beginning key
kCAN	key_scancel	&0	KEY_SCANCEL, 0573, sent by shifted cancel key
kCMD	key_scommand	*1	KEY_SCOMMAND, 0574, sent by shifted command key
kCPY	key_scopy	*2	KEY_SCOPY, 0575, sent by shifted copy key
kCRT	key_screate	*3	KEY_SCREATE, 0576, sent by shifted create key
kDC	key_sdc	*4	KEY_SDC, 0577, sent by shifted delete-char key
kDL	key_sdl	*5	KEY_SDL, 0600, sent by shifted delete-line key
kEND	key_send	*7	KEY_SEND, 0602, sent by shifted end key
kEOL	key_seol	*8	KEY_SEOL, 0603, sent by shifted clear-line key
kEXT	key_sexit	*9	KEY_SEXIT, 0604, sent by shifted exit key

kFND	key_sfind	*0	KEY_SFIND, 0605, sent by shifted find key
kHLP	key_shelp	#1	KEY_SHELP, 0606, sent by shifted help key
kHOM	key_shome	#2	KEY_SHOME, 0607, sent by shifted home key
kIC	key_sic	#3	KEY_SIC, 0610, sent by shifted input key
kLFT	key_sleft	#4	KEY_SLEFT, 0611, sent by shifted left-arrow key
kMOV	key_smove	%b	KEY_SMOVE, 0613, sent by shifted move key
kMSG	key_smessage	%a	KEY_SMESSAGE, 0612, sent by shifted message key
kNXT	key_snext	%c	KEY_SNEXT, 0614, sent by shifted next key
kOPT	key_soptions	%d	KEY_SOPTIONS, 0615, sent by shifted options key
kPRT	key_sprint	%f	KEY_SPRINT, 0617, sent by shifted print key
kPRV	key_sprevious	%e	KEY_SPREVIOUS, 0616, sent by shifted prev key
kRDO	key_sredo	%g	KEY_SREDO, 0620, sent by shifted redo key
kRES	key_srsume	%j	KEY_SRSUME, 0623, sent by shifted resume key
kRIT	key_sright	%i	KEY_SRIGHT, 0622, sent by shifted right-arrow key
kRPL	key_sreplace	%h	KEY_SREPLACE, 0621, sent by shifted replace key
kSAV	key_ssave	!1	KEY_SSAVE, 0624, sent by shifted save key
kSPD	key_ssuspend	!2	KEY_SSUSPEND, 0625, sent by shifted suspend key
kUND	key_sundo	!3	KEY_SUNDO, 0626, sent by shifted undo key
ka1	key_a1	K1	KEY_A1, 0534, upper left of keypad
ka3	key_a3	K3	KEY_A3, 0535, upper right of keypad
kb2	key_b2	K2	KEY_B2, 0536, center of keypad
kbeg	key_beg	@1	KEY_BEG, 0542, sent by beg(inning) key
kbs	key_backspace	kb	KEY_BACKSPACE, 0407, sent by backspace key
kc1	key_c1	K4	KEY_C1, 0537, lower left of keypad
kc3	key_c3	K5	KEY_C3, 0540, lower right of keypad
kcan	key_cancel	@2	KEY_CANCEL, 0543, sent by cancel key
kcbt	key_btab	kB	KEY_BTAB, 0541, sent by back-tab key
kclo	key_close	@3	KEY_CLOSE, 0544, sent by close key
kclr	key_clear	kC	KEY_CLEAR, 0515, sent by clear-screen or erase key
kcmd	key_command	@4	KEY_COMMAND, 0545, sent by cmd (command) key
kcpy	key_copy	@5	KEY_COPY, 0546, sent by copy key
kcrt	key_create	@6	KEY_CREATE, 0547, sent by create key
kctab	key_ctab	kt	KEY_CTAB, 0525, sent by clear-tab key
kcub1	key_left	kl	KEY_LEFT, 0404, sent by terminal left-arrow key
kcud1	key_down	kd	KEY_DOWN, 0402, sent by terminal down-arrow key
kcuf1	key_right	kr	KEY_RIGHT, 0405, sent by terminal right-arrow key
kcuu1	key_up	ku	KEY_UP, 0403, sent by terminal up-arrow key
kdch1	key_dc	kD	KEY_DC, 0512, sent by delete-character key
kdl1	key_dl	kL	KEY_DL, 0510, sent by delete-line key
ked	key_eos	ked	KEY_EOS, 0516, sent by clear-to-end-of-screen key

kel	key_eol	kE	KEY_EOL, 0517, sent by clear-to-end-of-line key
kend	key_end	@7	KEY_END, 0550, sent by end kee
kent	key_enter	@8	KEY_ENTER, 0527, sent by enter/send key
kext	key_exit	@9	KEY_EXIT, 0551, sent by exit key
kf0	key_f0	k0	KEY_F(0), 0410, sent by function key f0
kf1	key_f1	k1	KEY_F(1), 0411, sent by function key f1
kf10	key_f10	k;	KEY_F(10), 0422, sent by function key f10
kf11	key_f11	F1	KEY_F(11), 0423, sent by function key f11
kf12	key_f12	F2	KEY_F(12), 0424, sent by function key f12
kf13	key_f13	F3	KEY_F(13), 0425, sent by function key f13
kf14	key_f14	F4	KEY_F(14), 0426, sent by function key f14
kf15	key_f15	F5	KEY_F(15), 0427, sent by function key f15
kf16	key_f16	F6	KEY_F(16), 0430, sent by function key f16
kf17	key_f17	F7	KEY_F(17), 0431, sent by function key f17
kf18	key_f18	F8	KEY_F(18), 0432, sent by function key f18
kf19	key_f19	F9	KEY_F(19), 0433, sent by function key f19
kf2	key_f2	k2	KEY_F(2), 0412, sent by function key f2
kf20	key_f20	FA	KEY_F(20), 0434, sent by function key f20
kf21	key_f21	FB	KEY_F(21), 0435, sent by function key f21
kf22	key_f22	FC	KEY_F(22), 0436, sent by function key f22
kf23	key_f23	FD	KEY_F(23), 0437, sent by function key f23
kf24	key_f24	FE	KEY_F(24), 0440, sent by function key f24
kf25	key_f25	FF	KEY_F(25), 0441, sent by function key f25
kf26	key_f26	FG	KEY_F(26), 0442, sent by function key f26
kf27	key_f27	FH	KEY_F(27), 0443, sent by function key f27
kf28	key_f28	FI	KEY_F(28), 0444, sent by function key f28
kf29	key_f29	FJ	KEY_F(29), 0445, sent by function key f29
kf3	key_f3	k3	KEY_F(3), 0413, sent by function key f3
kf30	key_f30	FK	KEY_F(30), 0446, sent by function key f30
kf31	key_f31	FL	KEY_F(31), 0447, sent by function key f31
kf32	key_f32	FM	KEY_F(32), 0450, sent by function key f32
kf33	key_f33	FN	KEY_F(13), 0451, sent by function key f13
kf34	key_f34	FO	KEY_F(34), 0452, sent by function key f34
kf35	key_f35	FP	KEY_F(35), 0453, sent by function key f35
kf36	key_f36	FQ	KEY_F(36), 0454, sent by function key f36
kf37	key_f37	FR	KEY_F(37), 0455, sent by function key f37
kf38	key_f38	FS	KEY_F(38), 0456, sent by function key f38
kf39	key_f39	FT	KEY_F(39), 0457, sent by function key f39
kf4	key_f4	k4	KEY_F(4), 0414, sent by function key f4
kf40	key_f40	FU	KEY_F(40), 0460, sent by function key f40

kf41	key_f41	FV	KEY_F(41), 0461, sent by function key f41
kf42	key_f42	FW	KEY_F(42), 0462, sent by function key f42
kf43	key_f43	FX	KEY_F(43), 0463, sent by function key f43
kf44	key_f44	FY	KEY_F(44), 0464, sent by function key f44
kf45	key_f45	FZ	KEY_F(45), 0465, sent by function key f45
kf46	key_f46	Fa	KEY_F(46), 0466, sent by function key f46
kf47	key_f47	Fb	KEY_F(47), 0467, sent by function key f47
kf48	key_f48	Fc	KEY_F(48), 0470, sent by function key f48
kf49	key_f49	Fd	KEY_F(49), 0471, sent by function key f49
kf5	key_f5	k5	KEY_F(5), 0415, sent by function key f5
kf50	key_f50	Fe	KEY_F(50), 0472, sent by function key f50
kf51	key_f51	Ff	KEY_F(51), 0473, sent by function key f51
kf52	key_f52	Fg	KEY_F(52), 0474, sent by function key f52
kf53	key_f53	Fh	KEY_F(53), 0475, sent by function key f53
kf54	key_f54	Fi	KEY_F(54), 0476, sent by function key f54
kf55	key_f55	Fj	KEY_F(55), 0477, sent by function key f55
kf56	key_f56	Fk	KEY_F(56), 0500, sent by function key f56
kf57	key_f57	Fl	KEY_F(57), 0501, sent by function key f57
kf58	key_f58	Fm	KEY_F(58), 0502, sent by function key f58
kf59	key_f59	Fn	KEY_F(59), 0503, sent by function key f59
kf6	key_f6	k6	KEY_F(6), 0416, sent by function key f6
kf60	key_f60	Fo	KEY_F(60), 0504, sent by function key f60
kf61	key_f61	Fp	KEY_F(61), 0505, sent by function key f61
kf62	key_f62	Fq	KEY_F(62), 0506, sent by function key f62
kf63	key_f63	Fr	KEY_F(63), 0507, sent by function key f63
kf7	key_f7	k7	KEY_F(7), 0417, sent by function key f7
kf8	key_f8	k8	KEY_F(8), 0420, sent by function key f8
kf9	key_f9	k9	KEY_F(9), 0421, sent by function key f9
kfnd	key_find	@0	KEY_FIND, 0552, sent by find key
khlp	key_help	%1	KEY_HELP, 0553, sent by help key
khome	key_home	kh	KEY_HOME, 0406, sent by home key
khts	key_stab	kT	KEY_STAB, 0524, sent by set-tab key
kich1	key_ic	kI	KEY_IC, 0513, sent by ins-char/enter ins-mode key
kil1	key_il	kA	KEY_IL, 0511, sent by insert-line key
kind	key_sf	kF	KEY_SF, 0520, sent by scroll-forward/down key
kll	key_ll	kH	KEY_LL, 0533, sent by home-down key
kmov	key_move	%4	KEY_MOVE, 0556, sent by move key
kmrk	key_mark	%2	KEY_MARK, 0554, sent by mark key
kmsg	key_message	%3	KEY_MESSAGE, 0555, sent by message key
knp	key_npage	kN	KEY_NPAGE, 0522, sent by next-page key

knxt	key_next	%5	KEY_NEXT, 0557, sent by next-object key
kopn	key_open	%6	KEY_OPEN, 0560, sent by open key
kopt	key_options	%7	KEY_OPTIONS, 0561, sent by options key
kpp	key_ppage	kP	KEY_PPAGE, 0523, sent by previous-page key
kprt	key_print	%9	KEY_PRINT, 0532, sent by print or copy key
kprv	key_previous	%8	KEY_PREVIOUS, 0562, sent by previous-object key
krdo	key_redo	%0	KEY_REDO, 0563, sent by redo key
kref	key_reference	&1	KEY_REFERENCE, 0564, sent by ref(erence) key
kres	key_resume	&5	KEY_RESUME, 0570, sent by resume key
krfr	key_refresh	&2	KEY_REFRESH, 0565, sent by refresh key
kri	key_sr	kR	KEY_SR, 0521, sent by scroll-backward/up key
krmir	key_eic	kM	KEY_EIC, 0514, sent by **rmir** or **smir** in insert mode
krpl	key_replace	&3	KEY_REPLACE, 0566, sent by replace key
krst	key_restart	&4	KEY_RESTART, 0567, sent by restart key
ksav	key_save	&6	KEY_SAVE, 0571, sent by save key
kslt	key_select	*6	KEY_SELECT, 0601, sent by select key
kspd	key_suspend	&7	KEY_SUSPEND, 0627, sent by suspend key
ktbc	key_catab	ka	KEY_CATAB, 0526, sent by clear-all-tabs key
kund	key_undo	&8	KEY_UNDO, 0630, sent by undo key
lf0	lab_f0	l0	Labels on function key f0 if not f0
lf1	lab_f1	l1	Labels on function key f1 if not f1
lf10	lab_f10	la	Labels on function key f10 if not f10
lf2	lab_f2	l2	Labels on function key f2 if not f2
lf3	lab_f3	l3	Labels on function key f3 if not f3
lf4	lab_f4	l4	Labels on function key f4 if not f4
lf5	lab_f5	l5	Labels on function key f5 if not f5
lf6	lab_f6	l6	Labels on function key f6 if not f6
lf7	lab_f7	l7	Labels on function key f7 if not f7
lf8	lab_f8	l8	Labels on function key f8 if not f8
lf9	lab_f9	l9	Labels on function key f9 if not f9
ll	cursor_to_ll	ll	Last line, first column (if no **cup**)
lpi	change_line_pitch	ZB	Change number of lines per inch
mc0	print_screen	ps	Print contents of the screen
mc4	prtr_off	pf	Turn off the printer
mc5	prtr_on	po	Turn on the printer
mc5p	prtr_non	pO	Turn on the printer for #1 bytes
mcub	parm_left_micro	Zg	Like **parm_left_cursor** for micro adjust
mcub1	micro_left	Za	Like **cursor_left** for micro adjustment
mcud	parm_down_micro	Zf	Like **parm_down_cursor** for micro adjust
mcud1	micro_down	ZZ	Like **cursor_down** for micro adjustment

mcuf	parm_right_micro	Zh	Like **parm_right_cursor** for micro adjust
mcuf1	micro_right	Zb	Like **cursor_right** for micro adjustment
mcuu	parm_up_micro	Zi	Like **parm_up_cursor** for micro adjust
mcuu1	micro_up	Zd	Like **cursor_up** for micro adjustment)
mgc	clear_margins	MC	Clear all margins (top, bottom, and sides)
mhpa	micro_column_address	ZY	Like **column_address** for micro adjustment
mrcup	cursor_mem_address	CM	Memory relative cursor addressing
mvpa	micro_row_address	Zc	Like **row_address** for micro adjustment
nel	newline	nw	Newline (behaves like **cr** followed by **lf**)
oc	orig_colors	oc	Set all color(-pair)s to the original ones
op	orig_pair	op	Set default color-pair to the original one
pad	pad_char	pc	Pad character (instead of null)
pfkey	pkey_key	pk	Prog funct key #1 to type string #2
pfloc	pkey_local	pl	Prog funct key #1 to execute string #2
pfx	pkey_xmit	px	Prog funct key #1 to xmit string #2
pln	plab_norm	pn	Prog label #1 to show string #2
porder	order_of_pins	Ze	Matches software bits to print-head pins
prot	enter_protected_mode	mp	Turn on protected mode
rbim	stop_bit_image	Zs	End printing bit image graphics
rc	restore_cursor	rc	Restore cursor to position of last sc
rcsd	stop_char_set_def	Zt	End definition of a character set
rep	repeat_char	rp	Repeat char #1 #2 times
rev	enter_reverse_mode	mr	Turn on reverse video mode
rf	reset_file	rf	Name of file containing reset string
rfi	req_for_input	RF	Send next input char (for ptys)
ri	scroll_reverse	sr	Scroll text down
rin	parm_rindex	SR	Scroll backward #1 lines
ritm	exit_italics_mode	ZR	Disable italics
rlm	exit_leftward_mode	ZS	Enable rightward (normal) carriage motion
rmacs	exit_alt_charset_mode	ae	End alternate character set
rmam	exit_am_mode	RA	Turn off automatic margins
rmcup	exit_ca_mode	te	String to end programs that use **cup**
rmdc	exit_delete_mode	ed	End delete mode
rmicm	exit_micro_mode	ZT	Disable micro motion capabilities
rmir	exit_insert_mode	ei	End insert mode
rmkx	keypad_local	ke	Out of "keypad-transmit" modey
rmln	label_off	LF	Turn off soft labels
rmm	meta_off	mo	Turn off "meta mode"
rmp	char_padding	rP	Like **ip** but when in replace mode
rmso	exit_standout_mode	se	End standout mode

rmul	exit_underline_mode	ue	End underscore mode
rmxon	exit_xon_mode	RX	Turn off xon/xoff handshaking
rs1	reset_1string	r1	Reset terminal completely to sane modes
rs2	reset_2string	r2	Reset terminal completely to sane modes
rs3	reset_3string	r3	Reset terminal completely to sane modes
rshm	exit_shadow_mode	ZU	Disable shadow printing
rsubm	exit_subscript_mode	ZV	Disable subscript printing
rsupm	exit_superscript_mode	ZW	Disable superscript printing
rum	exit_upward_mode	ZX	Enable downward (normal) carriage motion
rwidm	exit_doublewide_mode	ZQ	Disable double wide printing
sbim	start_bit_image	Zq	Start printing bit image graphics
sc	save_cursor	sc	Save cursor position
scp	set_color_pair	sp	Set current color-pair
scs	select_char_set	Zj	Select character set
scsc	start_char_set_def	Zr	Start definition of a character set
sdrfq	enter_draft_quality	ZG	Set draft quality print
setb	set_background	Sb	Set current background color
setf	set_foreground	Sf	Set current foreground color
sgr	set_attributes	sa	Define the video attributes #1-#9
sgr0	exit_attribute_mode	me	Turn off all attributes
sitm	enter_italics_mode	ZH	Enable italics
slm	enter_leftward_mode	ZI	Enable leftward carriage motion
smacs	enter_alt_charset_mode	as	Start alternate character set
smam	enter_am_mode	SA	Turn on automatic margins
smcup	enter_ca_mode	ti	String to begin programs that use cup
smdc	enter_delete_mode	dm	Delete mode (enter)
smgb	set_bottom_margin	Zk	Set bottom margin at current line
smgbp	set_bottom_margin_parm	Zl	Set bottom margin at line #1 or #2 lines from bottom
smgl	set_left_margin	ML	Set left margin at current line
smglp	set_left_margin_parm	Zm	Set left (right) margin at column #1 (#2)
smgr	set_right_margin	MR	Set right margin at current column
smgrp	set_right_margin_parm	Zn	Set right margin at column #1
smgt	set_top_margin	Zo	Set top margin at current line
smgtp	set_top_margin_parm	Zp	Set top (bottom) margin at line #1 (#2)
smicm	enter_micro_mode	ZJ	Enable micro motion capabilities
smir	enter_insert_mode	im	Insert mode (enter)
smkx	keypad_xmit	ks	Put terminal in "keypad-transmit" mode
smln	label_on	LO	Turn on soft labels
smm	meta_on	mm	Turn on "meta mode" (8th bit)
smso	enter_standout_mode	so	Begin standout mode

smul	enter_underline_mode	us	Start underscore mode
smxon	enter_xon_mode	SX	Turn on xon/xoff handshaking
snlq	enter_near_letter_quality	ZK	Set near-letter quality print
snrmq	enter_normal_quality	ZL	Set normal quality print
sshm	enter_shadow_mode	ZM	Enable shadow printing
ssubm	enter_subscript_mode	ZN	Enable subscript printing
ssupm	enter_superscript_mode	ZO	Enable superscript printing
subcs	subscript_characters	Zu	List of "subscript-able" characters
sum	enter_upward_mode	ZP	Enable upward carriage motion
supcs	superscript_characters	Zv	List of "superscript-able" characters
swidm	enter_doublewide_mode	ZF	Enable double wide printing
tbc	clear_all_tabs	ct	Clear all tab stops
tsl	to_status_line	ts	Go to status line, col #1
uc	underline_char	uc	Underscore one char and move past it
up	cursor_up	cuu1	Upline (cursor up)
vpa	row_address	cv	Vertical position absolute
wind	set_window	wi	Current window is lines #1-#2 cols #3-#4
xoffc	xoff_character	XF	X-off character
xonc	xon_character	XN	X-on character
zerom	zero_motion	Zx	No motion for the subsequent character

SAMPLE ENTRY

The following entry, which describes the AT&T 610 terminal, is among the more complex entries in the *terminfo* file as of this writing:

```
610|610bct|ATT610|att610|AT&T 610; 80 column; 98key keyboard
    am, eslok, hs, mir, msgr, xenl, xon,
    cols#80, it#8, lh#2, lines#24, lw#8, nlab#8, wsl#80,
    acsc=``aaffggjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
    bel=^G, blink=\E[5m, bold=\E[1m, cbt=\E[Z,
    civis=\E[?25l, clear=\E[H\E[J, cnorm=\E[?25h\E[?12l,
    cr=\r, csr=\E[%i%p1%d;%p2%dr, cub=\E[%p1%dD, cub1=\b,
    cud=\E[%p1%dB, cud1=\E[B, cuf=\E[%p1%dC, cuf1=\E[C,
    cup=\E[%i%p1%d;%p2%dH, cuu=\E[%p1%dA, cuu1=\E[A,
    cvvis=\E[?12;25h, dch=\E[%p1%dP, dch1=\E[P, dim=\E[2m,
    dl=\E[%p1%dM, dl1=\E[M, ed=\E[J, el=\E[K, el1=\E[1K,
    flash=\E[?5h$<200>\E[?5l, fsl=\E8, home=\E[H, ht=\t,
    ich=\E[%p1%d@, il=\E[%p1%dL, il1=\E[L, ind=\ED,
    invis=\E[8m,
    is1=\E[8;0|\E[?3;4;5;13;15l\E[13;20l\E[?7h\E[12h\E(B\E)0,
    is2=\E[0m^O, is3=\E(B\E)0, kLFT=\E[\s@, kRIT=\E[\sA,
    kbs=\b, kcbt=\E[Z, kclr=\E[2J, kcub1=\E[D, kcud1=\E[B,
    kcuf1=\E[C, kcuu1=\E[A, kf1=\EOc, kf10=\ENp,
```

```
   kf11=\ENq, kf12=\ENr, kf13=\ENs, kf14=\ENt, kf2=\EOd,
   kf3=\EOe, kf4=\EOf, kf5=\EOg, kf6=\EOh, kf7=\EOi,
   kf8=\EOj, kf9=\ENo, khome=\E[H, kind=\E[S, kri=\E[T,
   ll=\E[24H, mc4=\E[?4i, mc5=\E[?5i, nel=\EE,
   pfx=\E[%p1%d;%p2%l%02dq\s\s\sF%p1%1d\s\s\s\s
\s\s\s\s\s\s%p2%s,
   pln=\E[%p1%d;0;0;0q%p2%:-16.16s, rc=\E8, rev=\E[7m,
   ri=\EM, rmacs=^O, rmir=\E[4l, rmln=\E[2p, rmso=\E[m,
   rmul=\E[m, rs2=\Ec\E[?3l, sc=\E7,
   sgr=\E[0%?%p6%t;1%;%?%p5%t;2%;%?%p2%t;4%;%?%p4%t;5%;
%?%p3%p1% | %t;7%;%?%p7%t;8%;m%?%p9%t^N%e^O%;,
   sgr0=\E[m^O, smacs=^N, smir=\E[4h, smln=\E[p,
   smso=\E[7m, smul=\E[4m, tsl=\E7\E[25;%i%p1%dx,
```

Types of Capabilities in the Sample Entry

The sample entry shows the formats for the three types of *terminfo* capabilities listed: Boolean, numeric, and string. All capabilities specified in the *terminfo* source file must be followed by commas, including the last capability in the source file. In *terminfo* source files, capabilities are referenced by their capability names (as shown in the previous tables).

Boolean capabilities are specified by their comma separated cap names.

Numeric capabilities are followed by the character '#', then a positive integer value. Thus, in the sample, **cols** (which shows the number of columns available on a device) is assigned the value **80** for the AT&T 610. (Values for numeric capabilities may be specified in decimal, octal or hexadecimal, using normal C conventions.)

Finally, string-valued capabilities, e.g., **el** (clear to end of line sequence) are listed by a two- to five-character capname, an '=', and a string ended by the next occurrence of a comma. A delay in milliseconds may appear anywhere in such a capability, enclosed in **$<..>** brackets, as in **el=\EK$<3>**. Padding characters are supplied by **tputs**( ).

The delay can be any of the following: a number (5), a number followed by an '*' (5*), a number followed by a '/' (5/), or a number followed by both (5*/). An '*' shows 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 characters, the factor is still the number of lines affected. This is always 1 unless the device has **in** and the software uses it.) When an '*' 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 '/' indicates that the padding is mandatory. If a device has **xon** defined, the padding information is advisory and will only be used for cost estimates or when the device is in raw mode. Mandatory padding will be transmitted regardless of the setting of **xon**. If padding (whether advisory or mandatory) is specified for bel or flash, however, it will always be used, regardless of whether **xon** is specified.

terminfo offers notation for encoding special characters. 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**, and **\s** give a newline, linefeed, return, tab, backspace, formfeed, and space, respectively. Other escapes include: **\^** for caret (^); **** for backslash (\); **\,** for comma (,); **\:** for colon (:); and **\0** for null. (**\0** will actually produce **\200**, which does not terminate a string but behaves as a null character on most devices, providing CS7 is specified. (See *stty*(1).) Finally, characters may be given as three octal digits after a backslash (e.g., \123).

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. Note that capabilities are defined in a left-to-right order and, therefore, a prior definition will override a later definition.

Preparing Descriptions

The most effective way to prepare a device description is by imitating the description of a similar device in *terminfo* and building up a description gradually, using partial descriptions with *vi*(1) to check that they are correct. Be aware that an unusual device may expose deficiencies in the ability of the *terminfo* file to describe it or the inability of *vi*(1) to work with that device.

To test a new device description, set the environment variable **TERMINFO** to the pathname of a directory containing the compiled description you are working on and programs will look there insted of in **/usr/lib/terminfo**. To get the padding for insert-line correct (if the device manufacturer did not document it), a severe test is to comment out **xon**, edit a large file at 9600 baud with **vi**(1), delete 16 or so lines from the middle of the screen, then hit the **u** key several times quickly. If the display is corrupted, more padding is usually needed. A similar test can be used for insert-character.

Section 1-1: Basic Capabilities

The number of columns on each line for the device is given by the **cols** numeric capability. If the device has a screen, the number of lines on the screen is given by the **lines** capability. If the device wraps around to the beginning of the next line when it reaches the right margin, 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 (instead of clearing a position when a character is struck over), it should have the **os** capability.

If the device is a printing terminal, with no soft copy unit, specify both **hc** and **os**. If there is a way to move the cursor to the left edge of the current row, specify this as **cr**. (Normally, this is carriage return, control m.) If there is a way to produce an audible signal (e.g., a bell or a beep), specify it as **bel**. If, like most devices, the device uses the xon-xoff flow-control protocol, specify **xon**.

If there is a way to move the cursor one position to the left (e.g., backspace), that capability should be given as **cub1**. Similarly, sequences to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**, respectively. These local cursor motions must not alter the text they pass over; for example, you would not normally use "**cuf1=\s**" 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 screen terminal. Programs should never attempt to backspace around the left edge, unless **bw** is specified, and should never attempt to go up locally off the top. To scroll text up, a program goes to the bottom left corner of the screen and sends 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**. These versions have the same semantics as **ind** and **ri**, except that they take one parameter and scroll the number of lines specified by that parameter. 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. Backward motion from the left edge of the screen is possible only when **bw** is specified. Here, **cub1** 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 device has switch selectable automatic margins, **am** should be specified in the *terminfo* source file. Here, initialization strings should turn on this option, if possible. If the device has a command that 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 device 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 screen terminals. Thus, the AT&T 5320 hardcopy terminal is described as:

```
5320|att5320|AT&T 5320 hardcopy terminal,
    am, hc, os,
    cols#132,
    bel=^G, cr=\r, cub1=\b, cnd1=\n,
    dch1=\E[P, dl1=\E[M,
    ind=\n,
```

The Lear Siegler ADM–3 is described as:

```
adm3 | lsi adm3,
  am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H,
  cud1=^J, ind=^J, lines#24,
```

Section 1-2: Parameterized Strings

Cursor addressing and other strings requiring parameters are described by a parameterized string capability, with **printf**(3S)-like escapes (**%x**) in it. For example, to address the cursor, the **cup**, 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, indicate that by **mrcup**.

The parameter mechanism uses a stack and special % codes to manipulate the stack in the manner of Reverse Polish Notation (postfix). Typically, a sequence will push one of the parameters onto the stack, then print it in some format. Often, more complex operations are necessary. Operations are in postfix form with the operands in the usual order. For example, to subtract 5 from the first parameter, use **%p1%{5}%–**.

The % encodings have the following meanings:

%%	outputs '%'
%[[:]*flags*][*width*[.*precision*]][**doxXs**]	
	as in printf, flags are **[–+#]** and space
%c	print pop() gives %c
%p[1-9]	push i^{th} parm
%P[a-z]	set variable [a-z] to pop()
%g[a-z]	get variable [a-z] and push it
%'*c*'	push char constant *c*
%{*nn*}	push decimal constant *nn*
%l	push strlen(pop())
%+ %– %* %/ %m	
	arithmetic (%m is mod): push(pop $integer_2$() op pop $integer_1$(
%& %| %^	bit operations: push(pop $integer_2$() op pop *integersub* 1())
%= %> %<	logical operations: push(pop() op pop())
%A %O	logical operations: and, or
%! %~	unary operations: push(op pop())
%i	(for ANSI terminals) add 1 to first parm, if one parm present, or first two parms, if more than one parm present

%? 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

b_5%

c_i are conditions, b_i are bodies.

If the "–" flag is used with "%[doxXs]", a colon (:) must be placed between the "%" and the "–" to differentiate the flag from the binary "%–" operator, .e.g., "%:–16.16s".

Consider the Hewlett-Packard 2645, 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 zero-padded as two digits. Thus, its **cup** capability is "**cup**=\E&a%p2%2.2dc%p1%2.2dY$<6>".

The Micro-Term ACT-IV needs the current row and column sent preceded by a **^T**, with the row and column encoded in binary, "**cup**=^T%p1%c%p2%c". Devices that use "%c" must be able to backspace the cursor (**cub1**), and 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**, because 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%'\s'%+%c%p2%'\s'%+%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.

Section 1-3: Cursor Motions

If the terminal has a fast way to home the cursor (to upper left corner of screen), 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 Hewlett-Packard terminals cannot be used for **home** without losing some of the other terminal features.)

If the device 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 Hewlett-Packard 2645) 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 device does not have **cup**, e.g., the Tektronix 4025.

If the device 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, e.g., the *Concept*, with more than one page of memory. If the device has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the device 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 the **smcup** sequence will not restore the screen after an **rmcup** sequence is output (to the state prior to outputting **rmcup**), specify **nrrmc**.

Section 1-4: 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 beginning of the line to the current position inclusive leaving the cursor where it is, this should be given as **el1**. If the terminal can clear from the current position to the end of the display, 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.)

Section 1-5: 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 that the cursor is on, 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** that take a single parameter and insert or delete that many lines can be given as **il** and **dl**.

If the terminal has a settable destructive 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 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.

To determine if a terminal has destructive scrolling regions or non-destructive scrolling regions, create a scrolling region in the middle of the screen, place data on the bottom line of the scrolling region, move the cursor to the top line of the scrolling region, and do a reverse index (**ri**) followed by a delete line (**dl1**) or index (**ind**). If the data that was originally on the bottom line of the scrolling region was restored into the scrolling region by the **dl1** or **ind**, the terminal has non-destructive scrolling regions. Otherwise, it has destructive scrolling regions. Do not specify **csr** if the terminal has non-destructive scrolling regions, unless **ind**, **ri**, **indn**, **rin**, **dl**, and **dl1** all simulate destructive scrolling.

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, the **da** capability should be given; if display memory can be retained below, **db** should be given. These indicate that deleting a line or scrolling a full screen may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Section 1-6: Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character operations thata 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, e.g., 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 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, 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 versus 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 that have an insert mode and terminals that 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 that 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 requires both to be used in combination.)

If post-insert padding is needed, give this as a number of milliseconds padding in **ip** (a string option). Any other sequence that 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, both **smir**/**rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n*, will insert *n* blanks.

If padding is necessary between characters typed while not in insert mode, give this as a number of milliseconds padding in **rmp**.

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.

Section 1-7: Highlighting, Underlining, and Visible Bells

Your device may have one or more kinds of display attributes that allow you to highlight selected characters when they appear on the screen. The following display modes (shown with the names by which they are set) may be available: a blinking screen (**blink**), bold or extra-bright characters (**bold**), dim or half-bright characters (**dim**), blanking or invisible text (**invis**), protected text (**prot**), a reverse-video screen (**rev**), and an alternate character set (**smacs** to enter this mode and **rmacs** to exit it). (If a command is necessary before you can enter alternate character set mode, give the sequence in **enacs** or "enable alternate-character-set" mode.) Turning on any of these modes singly may or may not turn off other modes.

sgr0 should be used to turn off all video enhancement capabilities. It should always be specified because it represents the only way to turn off some capabilities, e.g., **dim** or **blink**.

You should choose one display method as *standout mode* (see *curses*(3X)) and use it to highlight error messages and other kinds of text to which you want to draw attention. Choose a form of display that provides strong contrast but that is easy on the eyes. (We recommend reverse-video plus half-bright 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, give **xmc** to tell how many spaces are left.

Sequences to begin underlining and end underlining can be specified as **smul** and **rmul**, respectively. If the device has a sequence to underline the current character and to move the cursor one space to the right (e.g., the Micro-Term MIME), this sequence can be specified as **uc**.

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm instead of having extra bits for each character. Some terminals, e.g., the Hewlett-Packard 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout 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), this can be given as **flash**; it must not move the cursor. A good flash can be done by changing the screen into reverse video, pad for 200 ms, then return the screen to normal video.

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**. The boolean **chts** should also be given. 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 either of these modes.

If your terminal generates underlined characters by using the underline character (with no special sequences needed) even though it does not otherwise overstrike characters, you should specify the capability **ul**. For devices on which a character overstriking another leaves both characters on the screen, specify the capability **os**. If overstrikes are erasable with a blank, indicate this by specifying **eo**.

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking nine parameters. Each parameter is either **0** or non-zero, as the corresponding attribute is on or off. The nine parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need to be supported by **sgr**; only those for which corresponding separate attribute commands exist should be supported. For example, let's assume that the terminal in question needs the following escape sequences to turn on various modes.

tparm Parameter	Attribute	Escape Sequence
	none	\E[0m
p1	standout	\E[0;4;7m
p2	underline	\E[0;3m
p3	reverse	\E[0;4m
p4	blink	\E[0;5m
p5	dim	\E[0;7m
p6	bold	\E[0;3;4m
p7	invis	\E[0;8m
p8	protect	not available
p9	altcharset	^O (off) ^N(on)

Note that each escape sequence requires a **0** to turn off other modes before turning on its own mode. Also note that, as suggested above, *standout* is set up to be the combination of *reverse* and *dim*. Also, because this terminal has no *bold* mode, *bold* is set up as the combination of *reverse* and *underline*. In addition, to allow combinations, e.g., *underline+blink*, the sequence to use would be **\E[0;3;5m**. The terminal doesn't have *protect* mode, either, but that cannot be simulated in any way, so **p8** is ignored. The *altcharset* mode is different in that it is either **^O** or **^N**, depending on whether it is off or on. If all modes were to be turned on, the sequence would be **\E[0;3;4;5;7;8m^N**.

Now look at when different sequences are output. For example, **;3** is output when either **p2** or **p6** is true, i.e., if either *underline* or *bold* modes are turned on. Writing out the above sequences with their dependencies, gives the following:

Sequence	When to Output	Terminfo Translation
\E[0	always	\E[0
;3	if p2 or p6	%?%p2%p6%|%t;3%;
;4	if p1 or p3 or p6	%?%p1%p3%|%p6%|%t;4%;
;5	if p4	%?%p4%t;5%;
;7	if p1 or p5	%?%p1%p5%|%t;7%;
;8	if p7	%?%p7%t;8%;
m	always	m
^N or ^O	if p9 ^N, else ^O	%?%p9%t^N%e^O%;

Putting this all together into the **sgr** sequence gives:

```
sgr=\E[0%?%p2%p6%|%t;3%;%?%p1%p3%|%p6%|%t;4%;%?%p5%t;5%;%?%p1%p5%
        |%t;7%;%?%p7%t;8%;m%?%p9%t^N%e^O%;,
```

NOTE: **sgr** and **sgr0** must always be specified.

Section 1-8: Keypad

If the device has a keypad that transmits sequences when the keys are pressed, this information can also be specified. Note that it is not possible to handle devices where the keypad only works in local (this applies, for example, to the unshifted Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, specify these sequences as **smkx** and **rmkx**. Otherwise, the keypad is assumed to always transmit.

The sequences 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, e.g., f0, f1, ..., f63, specify the sequences they send as **kf0, kf1, ..., kf63.**

If the first 11 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. Further keys are defined above in the capabilities list.

Strings to program function keys can be specified as **pfkey, pfloc,** and **pfx.** A string to program screen labels should be specified as **pln.** Each of these strings takes two parameters: a function key identifier and a string with which to program it. **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 mode; **pfx** causes the string to be transmitted to the computer.

The capabilities **nlab**, **lw** and **lh** define the number of programmable screen labels and their width and height. If there are commands to turn the labels on and off, give them in **smln** and **rmln**. **smln** is normally output after one or more **pln** sequences to make sure that the change becomes visible.

Section 1-9: Tabs and Initialization

If the device has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control I). A "backtab" command that moves leftward to the next tab stop can be given as **cbt**. By convention, if tty modes show that tabs are being expanded by the computer instead of being sent to the device, programs should not use **ht** or **cbt** (even if they are present) because the user may not have the tab stops properly set.

If the device has hardware tabs that are initially set every *n* spaces when the device is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used by **tput init** (see *tput*(1)) to determine whether to set the mode for hardware tab expansion and whether to set the tab stops. If the device has tab stops that can be saved in nonvolatile memory, the *terminfo* description can assume that they are properly set. 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).

Other capabilities include: **is1**, **is2**, and **is3**, initialization strings for the device; **iprog**, the path name of a program to be run to initialize the device; and **if**, the name of a file containing long initialization strings. These strings are expected to set the device into modes consistent with the rest of the *terminfo* description. They must be sent to the device each time the user logs in and be output in the following order: run the program **iprog**; output **is1**; output **is2**; set the margins using **mgc**, **smgl** and **smgr**; set the tabs using **tbc** and **hts**; print the file **if**; and finally output **is3**. This is usually done using the **init** option of *tput*(1); see *profile*(4).

Most initialization is done with **is2**. Special device modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. Sequences that do a harder reset from a totally unknown state can be given as **rs1**, **rs2**, **rf**, and **rs3**, analogous to **is1**, **is2**, **is3**, and **if**. (The method using files, **if** and **rf**, is used for a few terminals, from */usr/lib/tabset/**; however, the recommended method is to use the initialization and reset strings.) These strings are output by **tput reset**, which is used when the terminal gets into a wedged state.

Commands are normally placed in **rs1, rs2, rs3,** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set a terminal into 80-column mode would normally be part of **is2**, but on some terminals it causes an annoying glitch on the screen and is not normally needed because the terminal is usually already in 80-column mode.

If a more complex sequence is needed to set the tabs than can be described by using **tbc** and **hts**, the sequence can be placed in **is2** or **if**.

Any margin can be cleared with **mgc**. (For instructions on how to specify commands to set and clear margins, see *Margins* under *Printer Capabilities*.)

Section 1-10: Delays

Certain capabilities control padding in the *tty*(7) driver. These are primarily needed by hard-copy terminals, and are used by **tput init** to set tty modes appropriately. Delays embedded in the capabilities **cr, ind, cub1, ff,** and **tab** can be used to set the appropriate delay bits to be set in the tty driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

Section 1-11: Status Lines

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 (e.g., the Heathkit h19's 25th line, or the 24th line of a VT100 that is set to a 23-line scrolling region), the capability **hs** should be given. Special strings that go to a given column of the status line and 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 capability **tsl** takes one parameter, which is the column number of the status line to which the cursor is to be moved.

If escape sequences and other special commands, e.g., tab, work while in the status line, the flag **eslok** can be given. A string that 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**.

Section 1-12: Line Graphics

If the device has a line drawing alternate character set, the mapping of glyph to character would be given in **acsc**. The definition of this string is based on the alternate character set used in the DEC VT100 terminal, extended slightly with some characters from the AT&T 4410v1 terminal.

glyph Name	vt100+ Character
arrow pointing right	+
arrow pointing left	,
arrow pointing down	.
solid square block	0
lantern symbol	I
arrow pointing up	-
diamond	`
checker board (stipple)	a
degree symbol	f
plus/minus	g
board of squares	h
lower right corner	j
upper right corner	k
upper left corner	l
lower left corner	m
plus	n
scan line 1	o
horizontal line	q
scan line 9	s
left tee (├)	t
right tee (┤)	u
bottom tee (┴)	v
top tee (┬)	w
vertical line	x
bullet	~

The best way to describe a new device's line graphics set is to add a third column to the above table with the characters for the new device that produce the appropriate glyph when the device is in the alternate character set mode. For example:

glyph Name	vt100+ Character	New tty Character
upper left corner	l	R
lower left corner	m	F
upper right corner	k	T
lower right corner	j	G
horizontal line	q	,
vertical line	x	.

Now write down the characters left to right, as in "**acsc**=lRmFkTjGq\,x.".

In addition, *terminfo* allows you to define multiple character sets (see Section 2-5 for details).

Section 1-13: Color Manipulation

Let us define two methods of color manipulation: the Tektronix method and the HP method. The Tektronix method uses a set of N predefined colors (usually 8) from which a user can select "current" foreground and background colors. Thus, a terminal can support up to N colors mixed into N*N color-pairs to display on the screen at the same time. When using an HP method, the user cannot define the foreground independently of the background or vice-versa. Instead, the user must define an entire color-pair at once. Up to M color-pairs, made from 2*M different colors, can be defined this way. Most existing color terminals belong to one of these two classes of terminals.

The numeric variables **colors** and **pairs** define the number of colors and color-pairs that can display on the screen at the same time. If a terminal can change the definition of a color (e.g., the Tektronix 4100 and 4200 series terminals), this should be specified with **ccc** (can change color). To change the definition of a color (Tektronix method), use **initc** (initialize color). It requires four arguments: color number (ranging from 0 to **colors**–1) and three RGB (red, green, and blue) values (ranging from 0 to 1000).

Tektronix 4100 series terminals use a type of color notation called HLS (Hue Lightness Saturation) instead of RGB color notation. For such terminals you must define a boolean variable **hls**. The last three arguments to the **initc** string would be HLS values: H, ranging from 0 to 360; and L and S, ranging from 0 to 100.

If a terminal can change the definitions of colors, but uses a color notation different from RGB and HLS, a mapping to either RGB or HLS must be developed.

To set current foreground or background to a given color, use **setf** (set foreground) and **setb** (set background). They require one parameter: the number of the color. To initialize a color-pair (HP method), use **initp** (initialize pair). It requires seven parameters: the number of a color-pair (range=0 to **pairs**–1), and six RGB values: three for the foreground followed by three for the background. (Each of these groups of three should be in the order RGB.) When **initc** or **initp** are used, RGB or HLS arguments should be in the order "red, green, blue" or "hue, lightness, saturation"), respectively. To make a color-pair current, use **scp** (set color-pair). It takes one parameter, the number of a color-pair.

Some terminals (e.g., most color terminal emulators for PCs) erase areas of the screen with current background color. In such cases, **bce** (background color erase) should be defined. The variable **op** (original pair) contains a sequence for setting the foreground and the background colors to what they were at the terminal start-up time. Similarly, **oc** (original colors) contains a control sequence for setting all colors (for the Tektronix method) or color-pairs (for the HP method) to the values they had at the terminal start-up time.

Some color terminals substitute color for video attributes. Such video attributes should not be combined with colors. Information about these video attributes should be packed into the **ncv** (no color video) variable. There is a one-to-one correspondence between the nine least significant bits of that variable and the video attributes. The following table depicts this correspondence:

Attribute	Bit Position	Decimal Value
A_STANDOUT	0	1
A_UNDERLINE	1	2
A_REVERSE	2	4
A_BLINK	3	8
A_DIM	4	16
A_BOLD	5	32
A_INVIS	6	64
A_PROTECT	7	128
A_ALTCHARSET	8	256

When a particular video attribute should not be used with colors, the corresponding **ncv** bit should be set to 1; otherwise, it should be set to zero. To determine the information to pack into the **ncv** variable, you must add together the decimal values corresponding to those attributes that cannot coexist with colors. For example, if the terminal uses colors to simulate reverse video (bit number 2 and decimal value 4) and bold (bit number 5 and decimal value 32), the resulting value for **ncv** is 36 (4 + 32).

Section 1-14: Miscellaneous

If the terminal requires other than a null (zero) character as a pad, this can be given as **pad**. Only the first character of the **pad** string is used. If the terminal does not have a pad character, specify **npc**.

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, e.g., Tektronix 4025, this can be indicated with **cmdch**. A prototype command character is chosen that is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some operating systems: if the environment variable CC exists, all occurrences of the prototype character are replaced with the character in CC.

Terminal descriptions that do not represent a specific kind of known terminal, e.g., **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 is one of those supported by the SYSTEM V/88 virtual terminal protocol, the terminal number can be given as **vt**. A line-turn-around sequence to be transmitted before doing reads should be specified in **rfi**.

If the device 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. Sequences to turn on and off xon/xoff handshaking may be given in **smxon** and **rmxon**. If the characters used for handshaking are not ^S and ^Q, they may be specified with **xonc** and **xoffc**.

If the terminal has a "meta key" that acts as a shift key setting the 8th bit of any character transmitted, indicate 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.

Media copy strings that 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. 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. If the text is not displayed on the terminal screen when the printer is on, specify **mc5i** (silent printer). All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Section 1-15: Special Cases

The working model used by *terminfo* fits most terminals reasonably well. However, some terminals do not completely match that model, requiring special support by *terminfo*. These are not meant to be construed as deficiencies in the terminals; they are just differences between the working model and the actual hardware. They may be unusual devices or, for some reason, do not have all the features of the *terminfo* model implemented.

Terminals that cannot display tilde (~) characters, e.g., certain Hazeltine terminals, should indicate **hz**.

Terminals that ignore a linefeed immediately after an **am** wrap, e.g., *Concept* 100, should indicate **xenl**. Those terminals whose cursor remains on the right-most column until another character has been received, instead of wrapping immediately upon receiving the right-most character, e.g., VT100, should also indicate **xenl**.

If **el** is required to get rid of standout (instead of writing normal text on top of it), **xhp** should be given.

The Teleray terminals whose tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This capability is also taken to mean that it is not possible to position the cursor on top of a "magic cookie." Therefore, to erase standout mode, it is necessary, instead, to use delete and insert line.

The Beehive Superbee terminals that do not transmit the escape or control–C characters, should specify **xsb**, indicating that the f1 key is to be used for escape and the f2 key for control–C.

Section 1-16: 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 canceled by placing *xx*@ to the left of the capability definition, where *xx* is the capability. For example, the following entry defines an AT&T 4424 terminal that does not have the **rev**, **sgr**, and **smul** capabilities, therefore, cannot do highlighting:

> att4424-2|Teletype 4424 in display function group ii, rev@, sgr@, smul@, use=att4424,

This is useful for different modes for a terminal, or for different user preferences. More than one **use** capability may be given.

PART 2: PRINTER CAPABILITIES

The *terminfo* database allows you to define capabilities of printers and terminals. To find out what capabilities are available for printers and terminals, see the two lists under *Device Capabilities* that list capabilities by variable and by capability name.

Section 2-1: Rounding Values

Because parameterized string capabilities work only with integer values, we recommend that *terminfo* designers create strings that expect numeric values that have been rounded. Application designers should note this and should always round values to the nearest integer before using them with a parameterized string capability.

Section 2-2: Printer Resolution

A printer's resolution is defined to be the smallest spacing of characters it can achieve. In general, printers have independent resolution horizontally and vertically. Thus, the vertical resolution of a printer can be determined by measuring the smallest achievable distance between consecutive printing baselines, while the horizontal resolution can be determined by measuring the smallest achievable distance between the left-most edges of consecutive printed, identical, characters.

All printers are assumed to be capable of printing with a uniform horizontal and vertical resolution. The view of printing that *terminfo* currently presents is one of printing inside a uniform matrix: All characters are printed at fixed positions relative to each "cell" in the matrix; furthermore, each cell has the same size given by the smallest horizontal and vertical step sizes dictated by the resolution. (The cell size can be changed as will be seen later.)

Many printers are capable of "proportional printing," where the horizontal spacing depends on the size of the character last printed. *terminfo* does not make use of this capability, although it does provide enough capability definitions to allow an application to simulate proportional printing.

A printer must not only be able to print characters as close together as the horizontal and vertical resolutions suggest, but also of "moving" to a position an integral multiple of the smallest distance away from a previous position. Thus, printed characters can be spaced apart a distance that is an integral multiple of the smallest distance, up to the length or width of a single page.

Some printers can have different resolutions depending on different "modes." In "normal mode," the existing *terminfo* capabilities are assumed to work on columns and lines, just like a video terminal. Thus, the old **lines** capability would give the length of a page in lines, and the **cols** capability would give the width of a page in columns. In "micro mode," many *terminfo* capabilities work on increments of lines and columns. With some printers the micro mode may be concomitant with normal mode, so that all the capabilities work at the same time.

Section 2-3: Specifying Printer Resolution'

The printing resolution of a printer is given in several ways. Each specifies the resolution as the number of smallest steps per distance:

Specification of Printer Resolution

Characteristic	Number of Smallest Steps
orhi	Steps per inch horizontally
orvi	Steps per inch vertically
orc	Steps per column
orl	Steps per line

When printing in normal mode, each character printed causes movement to the next column, except in special cases described later; the distance moved is the same as the per-column resolution. Some printers cause an automatic movement to the next line when a character is printed in the rightmost position; the distance moved vertically is the same as the per-line resolution. When printing in micro mode, these distances can be different, and may be zero for some printers.

Specification of Printer Resolution

Automatic Motion after Printing

Normal Mode:	
orc	Steps moved horizontally
orl	Steps moved vertically
Micro Mode:	
mcs	Steps moved horizontally
mls	Steps moved vertically

Some printers are capable of printing wide characters. The distance moved when a wide character is printed in normal mode may be different from when a regular width character is printed. The distance moved when a wide character is printed in micro mode may also be different from when a regular character is printed in micro mode, but the differences are assumed to be related: if the distance moved for a regular character is the same whether in normal mode or micro mode, (**mcs=orc**), the distance moved for a wide character is also the same whether in normal mode or micro mode. This doesn't mean the normal character distance is the same as the wide character distance, just that the distances don't change with a change in normal to micro mode. However, if the distance moved for a regular character is different in micro mode from the distance moved in normal mode (**mcs<orc**), the micro mode distance is assumed to be the same for a wide character printed in micro mode, as the following table shows:

Specification of Printer Resolution
Automatic Motion after Printing Wide Character

*Normal Mode or Micro Mode (***mcs = orc***):*	
widcs	Steps moved horizontally
*Micro Mode (***mcs < orc***):*	
mcs	Steps moved horizontally

There may be control sequences to change the number of columns per inch (the character pitch) and to change the number of lines per inch (the line pitch). If these are used, the resolution of the printer changes, but the type of change depends on the printer:

Specification of Printer Resolution
Changing the Character/Line Pitches

cpi	Change character pitch
cpix	If set, **cpi** changes **orhi**, otherwise changes **orc**
lpi	Change line pitch
lpix	If set, **lpi** changes **orvi**, otherwise changes **orl**
chr	Change steps per column
cvr	Change steps per line

The **cpi** and **lpi** string capabilities are each used with a single argument, the pitch in columns (or characters) and lines per inch, respectively. The **chr** and **cvr** string capabilities are each used with a single argument, the number of steps per column and line, respectively.

Using any of the control sequences in these strings implies a change in some of the values of **orc, orhi, orl,** and **orvi**. Also, the distance moved when a wide character is printed, **widcs,** changes in relation to **orc**. The distance moved when a character is printed in micro mode, **mcs,** changes similarly, with one exception: if the distance is 0 or 1, no change is assumed (see items marked with † in the following table).

Programs that use **cpi, lpi, chr,** or **cvr** should recalculate the printer resolution (and should recalculate other values —see *Effect of Changing Printing Resolution* under *Dot-Mapped Graphics*).

Specification of Printer Resolution
Effects of Changing the Character/Line Pitches

Before	*After*
Using **cpi** *with* **cpix** *clear:*	
orhi ′	**orhi**
orc ′	$\mathbf{orc} = \frac{\mathbf{orhi}}{V_{cpi}}$
Using **cpi** *with* **cpix** *set:*	
orhi ′	$\mathbf{orhi} = \mathbf{orc} \cdot V_{cpi}$
orc ′	**orc**
Using **lpi** *with* **lpix** *clear:*	
orvi ′	**orvi**
orl ′	$\mathbf{orl} = \frac{\mathbf{orvi}}{V_{lpi}}$
Using **lpi** *with* **lpix** *set:*	
orvi ′	$\mathbf{orvi} = \mathbf{orl} \cdot V_{lpi}$
orl ′	**orl**
Using **chr:**	
orhi ′	**orhi**
orc ′	V_{chr}
Using **cvr:**	
orvi ′	**orvi**
orl ′	V_{cvr}

Using **cpi** *or* **chr:**

widcs ′	$\mathbf{widcs} = \mathbf{widcs}' \frac{\mathbf{orc}}{\mathbf{orc}'}$
mcs ′†	$\mathbf{mcs} = \mathbf{mcs}' \frac{\mathbf{orc}}{\mathbf{orc}'}$

V_{cpi}, V_{lpi}, V_{chr}, and V_{cvr} are the arguments used with **cpi, lpi, chr,** and **cvr,** respectively. The † mark indicates the old value.

Section 2-4: Capabilities that Cause Movement

In the following descriptions, "movement" refers to the motion of the "current position." With video terminals this would be the cursor; with some printers this is the carriage position. Other printers have different equivalents. In general, the current position is where a character would be displayed if printed.

terminfo has string capabilities for control sequences that cause movement a number of full columns or lines. It also has equivalent string capabilities for control sequences that cause movement a number of smallest steps.

String Capabilities for Motion

mcub1	Move 1 step left
mcuf1	Move 1 step right
mcuu1	Move 1 step up
mcud1	Move 1 step down
mcub	Move *N* steps left
mcuf	Move *N* steps right
mcuu	Move *N* steps up
mcud	Move *N* steps down
mhpa	Move *N* steps from the left
mvpa	Move *N* steps from the top

The latter six strings are each used with a single argument, *N*.

Sometimes the motion is limited to less than the width or length of a page. Also, some printers don't accept absolute motion to the left of the current position. *terminfo* has capabilities for specifying these limits.

Limits to Motion

mjump	Limit on use of **mcub1, mcuf1, mcuu1, mcud1**
maddr	Limit on use of **mhpa, mvpa**
xhpa	If set, **hpa** and **mhpa** can't move left
xvpa	If set, **vpa** and **mvpa** can't move up

If a printer needs to be in a "micro mode" for the motion capabilities described above to work, there are string capabilities defined to contain the control sequence to enter and exit this mode. A boolean is available for those printers where using a carriage return causes an automatic return to normal mode.

Entering/Exiting Micro Mode	
smicm	Enter micro mode
rmicm	Exit micro mode
crxm	Using **cr** exits micro mode

The movement made when a character is printed in the rightmost position varies among printers. Some make no movement, some move to the beginning of the next line, others move to the beginning of the same line. *terminfo* has boolean capabilities for describing all three cases.

What Happens After Character Printed in Rightmost Position	
sam	Automatic move to beginning of same line

Some printers can be put in a mode where the normal direction of motion is reversed. This mode can be useful when there are no capabilities for leftward or upward motion, because those capabilities can be built from the motion reversal capability and the rightward or downward motion capabilities. It is best to leave it up to an application to build the leftward or upward capabilities and not enter them in the *terminfo* database. This allows several reverse motions to be strung together without intervening wasted steps that leave and re-enter reverse mode.

Entering/Exiting Reverse Modes	
slm	Reverse sense of horizontal motions
rlm	Restore sense of horizontal motions
sum	Reverse sense of vertical motions
rum	Restore sense of vertical motions
While sense of horizontal motions reversed:	
mcub1	Move 1 step right
mcuf1	Move 1 step left
mcub	Move *N* steps right
mcuf	Move *N* steps left
cub1	Move 1 column right
cuf1	Move 1 column left
cub	Move *N* columns right
cuf	Move *N* columns left
While sense of vertical motions reversed:	
mcuu1	Move 1 step down
mcud1	Move 1 step up
mcuu	Move *N* steps down
mcud	Move *N* steps up
cuu1	Move 1 line down
cud1	Move 1 line up
cuu	Move *N* lines down
cud	Move *N* lines up

The reverse motion modes should not affect the **mvpa** and **mhpa** absolute motion capabilities. The reverse vertical motion mode should, however, also reverse the action of the line "wrapping" that occurs when a character is printed in the right-most position. Thus printers that have the standard *terminfo* capability **am** defined should experience motion to the beginning of the previous line when a character is printed in the right-most position under reverse vertical motion mode.

The action when any other motion capabilities are used in reverse motion modes is not defined; thus, programs must exit reverse motion modes before using other motion capabilities.

Two miscellaneous capabilities complete the list of new motion capabilities. One of these is needed for printers that move the current position to the beginning of a line when certain control characters, e.g., "line-feed" or "form-feed," are used. The other is used for the capability of suspending the motion that normally occurs after printing a character.

Miscellaneous Motion Strings

docr	List of control characters causing **cr**
zerom	Prevent auto motion after printing next single character

Margins

terminfo provides two strings for setting margins on terminals: one for the left and one for the right margin. Printers, however, have two additional margins, for the top and bottom margins of each page. Furthermore, some printers require not using motion strings to move the current position to a margin and fixing the margin there, but require the specification of where a margin should be regardless of the current position. Therefore, *terminfo* offers six additional strings for defining margins with printers.

Setting Margins

smgl	Set left margin at current column
smgr	Set right margin at current column
smgb	Set bottom margin at current line
smgt	Set top margin at current line
smgbp	Set bottom margin at line *N*
smglp	Set left margin at column *N*
smgrp	Set right margin at column *N*
smgtp	Set top margin at line *N*

The last four strings are used with one or more arguments that give the position of the margin or margins to set. If both of **smglp** and **smgrp** are set, each is used with a single argument, *N* , that gives the column number of the left and right margin, respectively. If both **smgtp** and **smgbp** are set, each is used to set the top and bottom margin, respectively: **smgtp** is used with a single argument, *N* , the line number of the top margin; however, **smgbp** is used with two arguments, *N* and *M* , that give the line number of the bottom margin, the first counting from the top of the page and the second counting from the bottom. This accommodates the two styles of specifying the bottom margin in different manufacturers' printers.

When coding a *terminfo* entry for a printer that has a settable bottom margin, only the first or second parameter should be used, depending on the printer. When writing an application that uses **smgbp** to set the bottom margin, both arguments must be given.

If only one of **smglp** and **smgrp** is set, it is used with two arguments, the column number of the left and right margins, in that order. Likewise, if only one of **smgtp** and **smgbp** is set, it is used with two arguments that give the top and bottom margins, in that order, counting from the top of the page. Thus when coding a *terminfo* entry for a printer that requires setting both left and right or top and bottom margins simultaneously, only one of **smglp** and **smgrp** or **smgtp** and **smgbp** should be defined; the other should be left blank. When writing an application that uses these string capabilities, the pairs should be first checked to see if each in the pair is set or only one is set, and should then be used accordingly.

In counting lines or columns, line zero is the top line and column zero is the left-most column. A zero value for the second argument with **smgbp** means the bottom line of the page.

All margins can be cleared with **mgc**.

Shadows, Italics, Wide Characters, Superscripts, Subscripts

Five new sets of strings are used to describe the capabilities printers have of enhancing printed text.

Enhanced Printing

sshm	Enter shadow-printing mode
rshm	Exit shadow-printing mode
sitm	Enter italicizing mode
ritm	Exit italicizing mode
swidm	Enter wide character mode
rwidm	Exit wide character mode
ssupm	Enter superscript mode
rsupm	Exit superscript mode
supcs	List of characters available as superscripts
ssubm	Enter subscript mode
rsubm	Exit subscript mode
subcs	List of characters available as subscripts

If a printer requires the **sshm** control sequence before every character to be shadow-printed, the **rshm** string is left blank. Thus, programs that find a control sequence in **sshm** but none in **rshm** should use the **sshm** control sequence before every character to be shadow-printed. Otherwise, the **sshm** control sequence should be used once before the set of characters to be shadow-printed, followed by **rshm**. The same is also true of each **sitm/ritm**, **swidm/rwidm**, **ssupm/rsupm**, and **ssubm/rsubm** pairs.

Note that *terminfo* also has a capability for printing emboldened text (**bold**). While shadow printing and emboldened printing are similar in that they "darken" the text, many printers produce these two types of print in slightly different ways. Generally, emboldened printing is done by overstriking the same character one or more times. Shadow printing likewise usually involves overstriking, but with a slight movement up and/or to the side so that the character is "fatter."

It is assumed that enhanced printing modes are independent modes, so that it would be possible, for instance, to shadow print italicized subscripts.

As mentioned earlier, the amount of motion automatically made after printing a wide character should be given in **widcs**.

If only a subset of the printable ASCII characters can be printed as superscripts or subscripts, they should be listed in **supcs** or **subcs** strings, respectively. If the **ssupm** or **ssubm** strings contain control sequences, but the corresponding **supcs** or **subcs** strings are empty, it is assumed that all printable ASCII characters are available as superscripts or subscripts.

Automatic motion made after printing a superscript or subscript is assumed to be the same as for regular characters. Thus, for example, printing any of the following three examples will result in equivalent motion: Bi B_i B^i

Note that the existing **msgr** boolean capability describes whether motion control sequences can be used while in "tandout mode." This capability is extended to cover the enhanced printing modes added here. **msgr** should be set for those printers that accept any motion control sequences without affecting shadow, italicized, widened, superscript, or subscript printing. Conversely, if **msgr** is not set, a program should end these modes before attempting any motion.

Section 2-5: Alternate Character Sets

In addition to allowing you to define line graphics (described in Section 1-12), *terminfo* lets you define alternate character sets. The following capabilities cover printers and terminals with multiple selectable or definable character sets.

Alternate Character Sets

scs	Select character set *N*
scsd	Start definition of character set *N*, *M* characters
defc	Define character *A*, *B* dots wide, descender *D*
rcsd	End definition of character set *N*
csnm	List of character set names
daisy	Printer has manually changed print-wheels

The **scs**, **rcsd**, and **csnm** strings are used with a single argument, *N*, a number from 0 to 63 that identifies the character set. The **scsd** string is also used with the argument *N* and another, *M*, that gives the number of characters in the set. The **defc** string is used with three arguments: *A* gives the ASCII code representation for the character, *B* gives the width of the character in dots, and *D* is zero or one depending on whether the character is a "descender" or not. The **defc** string is also followed by a string of "image-data" bytes that describe how the character looks (described later).

Character set 0 is the default character set present after the printer has been initialized. Not every printer has 64 character sets, of course; using **scs** with an argument that doesn't select an available character set should cause a null result from *tparm*().

If a character set has to be defined before it can be used, the **scsd** control sequence is to be used before defining the character set, and the **rcsd** is to be used after. They should also cause a null result from *tparm ()* when used with an argument *N* that doesn't apply. If a character set still has to be selected after being defined, the **scs** control sequence should follow the **rcsd** control sequence. By examining the results of using each of the **scs**, **scsd**, and **rcsd** strings with a character set number in a call to *tparm*(), a program can determine which of the three are needed.

Between use of the **scsd** and **rcsd** strings, the **defc** string should be used to define each character. To print any character on printers covered by *terminfo*, the ASCII code is sent to the printer. This is true for characters in an alternate set as well as "normal" characters. Thus, the definition of a character includes the ASCII code that represents it. In addition, the width of the character in dots is given, with an indication of whether the character should descend below the print line (e.g., the lowercase letter "g" in most character sets). The width of the character in dots also indicates the number of image-data bytes that will follow the **defc** string. These image-data bytes indicate where in a dot-matrix pattern ink should be applied to "draw" the character; the number of these bytes and their form are defined below under *Dot-Mapped Graphics.*

It's easiest for the creator of *terminfo* entries to refer to each character set by number; however, these numbers will be meaningless to the application developer. The **csnm** string alleviates this problem by providing names for each number.

When used with a character set number in a call to *tparm*( ), the **csnm** string produces the equivalent name. These names should be used as a reference only. No naming convention is implied, although anyone who creates a *terminfo* entry for a printer should use names consistent with the names found in user documents for the printer. Application developers should allow a user to specify a character set by number (leaving it up to the user to examine the **csnm** string to determine the correct number), or by name, where the application examines the **csnm** string to determine the corresponding character set number.

These capabilities are usually used only with dot-matrix printers. If they are not available, the strings should not be defined. For printers that have manually changed print-wheels or font cartridges, the boolean **daisy** is set.

Section 2-6: Dot-Matrix Graphics

Dot-matrix printers typically have the capability of reproducing "raster-graphics" images. Three new numeric capabilities and three new string capabilities can help a program draw raster-graphics images independent of the type of dot-matrix printer or the number of pins or dots the printer can handle at one time.

Dot-Matrix Graphics	
npins	Number of pins, *N*, in print-head
spinv	Spacing of pins vertically in pins per inch
spinh	Spacing of dots horizontally in dots per inch
porder	Matches software bits to print-head pins
sbim	Start printing bit image graphics, *B* bits wide
rbim	End printing bit image graphics

The **sbim** sring is used with a single argument, *B*, the width of the image in dots.

The model of dot-matrix or raster-graphics that *terminfo* presents is similar to the technique used for most dot-matrix printers: each pass of the printer's print-head is assumed to produce a dot-matrix that is *N* dots high and *B* dots wide. This is typically a wide, squat, rectangle of dots. The height of this rectangle in dots varies from one printer to the next; this is given in the **npins** numeric capability. The size of the rectangle in fractions of an inch also varies; it can be deduced from the **spinv** and **spinh** numeric capabilities. With these three values, an application can divide a complete raster-graphics image into several horizontal strips, perhaps interpolating to account for different dot spacing vertically and horizontally.

The **sbim** and **rbim** strings are used to start and end a dot-matrix image, respectively. The **sbim** string is used with a single argument that gives the width of the dot-matrix in dots. A sequence of "image-data bytes" are sent to the printer after the **sbim** string and before the **rbim** string. The number of bytes is a integral multiple of the width of the dot-matrix; the multiple and the form of each byte is determined by the **porder** string.

The **porder** string is a comma separated list of pin numbers optionally followed by an numerical offset. The offset, if given, is separated from the list with a semicolon. The position of each pin number in the list corresponds to a bit in an 8-bit data byte. The pins are numbered consecutively from 1 to **npins**, with 1 being the top pin. Note that the term "pin" is used loosely here; "ink-jet" dot-matrix printers don't have pins, but can be considered to have an equivalent method of applying a single dot of ink to paper. The bit positions in **porder** are in groups of 8, with the first position in each group the most significant bit and the last position the least significant bit. An application produces 8-bit bytes in the order of the groups in **porder** .

An application computes the "image-data bytes" from the internal image, mapping vertical dot positions in each print-head pass into 8-bit bytes, using a 1 bit where ink should be applied and 0 where no ink should be applied. This can be reversed (0 bit for ink, 1 bit for no ink) by giving a negative pin number. If a position is skipped in **porder,** a 0 bit is used. If a position has a lowercase 'x' instead of a pin number, a 1 bit is used in the skipped position. For consistency, a lowercase 'o' can be used to represent a 0 filled, skipped bit. There must be a multiple of 8 bit positions used or skipped in **porder ;** if not, 0 bits are used to fill the last byte in the least significant bits. The offset, if given, is added to each data byte; the offset can be negative.

Some examples may help clarify the use of the **porder** string. The AT&T 470, AT&T 475 and C.Itoh 8510 printers provide 8 pins for graphics. The pins are identified top to bottom by the 8 bits in a byte, from least significant to most. The **porder** strings for these printers would be 8,7,6,5,4,3,2,1. The AT&T 478 and AT&T 479 printers also provide 8 pins for graphics. However, the pins are identified in the reverse order. The **porder** strings for these printers would be 1,2,3,4,5,6,7,8.

The AT&T 5310, AT&T 5320, DEC LA100, and DEC LN03 printers provide 6 pins for graphics. The pins are identified top to bottom by the decimal values 1,2,4,8,16,32. These correspond to the low 6 bits in an 8-bit byte, although the decimal values are further offset by the value 63. The **porder** string for these printers would be 6,5,4,3,2,1;63, or alternately o,o,6,5,4,3,2,1;63.

Section 2-7: Effect of Changing Printing Resolution

If the control sequences to change the character pitch or the line pitch are used, the pin or dot spacing may change:

Dot-Matrix Graphics
Changing the Character/Line Pitches

cpi	Change character pitch
cpix	If set, **cpi** changes **spinh**
lpi	Change line pitch
lpix	If set, **lpi** changes **spinv**

Programs that use **cpi** or **lpi** should recalculate the dot spacing:

Dot-Matrix Graphics
Effects of Changing the Character/Line Pitches

Before	*After*
Using **cpi** *with* **cpix** *clear:*	
spinh $'$	**spinh**
Using **cpi** *with* **cpix** *set:*	
spinh $'$	$\mathbf{spinh}=\mathbf{spinh}'\cdot\frac{\mathbf{orhi}}{\mathbf{orhi}'}$
Using **lpi** *with* **lpix** *clear:*	
spinv $'$	**spinv**
Using **lpi** *with* **lpix** *set:*	
spinv $'$	$\mathbf{spinv}=\mathbf{spinv}'\cdot\frac{\mathbf{orhi}}{\mathbf{orhi}'}$
Using **chr:**	
spinh $'$	**spinh**
Using **cvr:**	
spinv $'$	**spinv**

orhi′ and **orhi** are the values of the horizontal resolution in steps per inch, before using **cpi** and after using **cpi**, respectively. Likewise, **orvi′** and **orvi** are the values of the vertical resolution in steps per inch, before using **lpi** and after using **lpi**, respectively. Thus, the changes in the dots per inch for dot-matrix graphics follow the changes in steps per inch for printer resolution.

Section 2-8: Print Quality

Many dot-matrix printers can alter the dot spacing of printed text to produce near "letter quality" printing or "draft quality" printing. Usually, it is important to be able to choose one or the other because the rate of printing generally falls off as the quality improves. There are three new strings used to describe these capabilities:

Print Quality

snlq	Set near-letter quality print
snrmq	Set normal quality print
sdrfq	Set draft quality print

The capabilities are listed in decreasing levels of quality. If a printer doesn't have all three levels, one or two of the strings should be left blank as appropriate.

Section 2-9: Printing Rate and Buffer Size

Because there is no standard protocol that can be used to keep a program synchronized with a printer, and because modern printers can buffer data before printing it, a program generally cannot determine at any time what has been printed. Two new numeric capabilities can help a program estimate what has been printed.

Print Rate/Buffer Size

cps	Nominal print rate in characters per second
bufsz	Buffer capacity in characters

cps is the nominal or average rate at which the printer prints characters; if this value is not given, the rate should be estimated at one-tenth the prevailing baud rate. **bufsz** is the maximum number of subsequent characters buffered before the guaranteed printing of an earlier character, assuming proper flow control has been used. If this value is not given it is assumed that the printer does not buffer characters, but prints them as they are received.

As an example, if a printer has a 1000-character buffer, sending the letter "a" followed by 1000 additional characters is guaranteed to cause the letter "a" to print. If the same printer prints at the rate of 100 characters per second, it should take 10 seconds to print all the characters in the buffer, less if the buffer is not full. By keeping track of the characters sent to a printer, and knowing the print rate and buffer size, a program can synchronize itself with the printer.

Note that most printer manufacturers advertise the maximum print rate, not the nominal print rate. A good way to get a value to put in for **cps** is to generate a few pages of text, count the number of printable characters, then see how long it takes to print the text.

Applications that use these values should recognize the variability in the print rate. Straight text, in short lines, with no embedded control sequences may print at close to the advertised print rate and usually faster than the rate in **cps**.

Graphics data with many control sequences, or very long lines of text, print at well below the advertised rate and below the rate in **cps**. If the application is using **cps** to decide how long it should take a printer to print a block of text, the application should pad the estimate. If the application is using **cps** to decide how much text has already been printed, it should shrink the estimate. The application will err in favor of the user, who wants, above all, to see all the output in its correct place.

FILES

/usr/lib/terminfo/?/*	compiled terminal description database
/usr/lib/.COREterm/?/*	subset of compiled terminal description database
/usr/lib/tabset/*	tab settings for some terminals, in a format appropriate to be output to the terminal (escape sequences that set margins and tabs)

SEE ALSO

curses(3X), printf(3S) in the *Programmer's Reference Manual.*
captoinfo(1M), infocmp(1M), tic(1M), term(5), tty(7) in the *System Administrator's Reference Manual* .
tput(1) in the *User's Reference Manual.*
Chapter 10 of the *Programmer's Guide.*

WARNING

As described in the *Tabs and Initialization* section, a terminal's initialization strings, **is1, is2,** and **is3**, if defined, must be output before a *curses*(3X) program is run. An available mechanism for outputting such strings is **tput init** (see *tput*(1) and *profile*(4)).

If a null character (\0) is encountered in a string, the null and all characters after it are lost. Therefore, it is not possible to code a null character (\0) and send it to a device (either terminal or printer). The suggestion of sending a \0200, where a \0 (null) is needed can succeed only if the device (terminal or printer) ignores the eighth bit. For example, because all eight bits are used in the standard international ASCII character set, devices that adhere to this standard will treat \0200 differently from \0.

Tampering with entries in */usr/lib/.COREterm/?/** or */usr/lib/terminfo/?/** (e.g., changing or removing an entry) can affect programs like *vi*(1) that expect the entry to be present and correct. In particular, removing the description for the "dumb" terminal will cause unexpected problems.

NOTE

The *termcap* database (from earlier releases of SYSTEM V/88 Release 3.2) may not be supplied in future releases.

NAME

timezone – set default system time zone

SYNOPSIS

/etc/TIMEZONE

DESCRIPTION

This file sets and exports the time zone environmental variable **TZ**.

This file is "dotted" into other files that must know the time zone.

The syntax of **TZ** can be described as follows:

TZ	→	*zone*
		| *zone signed_time*
		| *zone signed_time zone*
		| *zone signed_time zone dst*
zone	→	*letter letter letter*
signed_time	→	*sign time*
		| *time*
time	→	*hour*
		| *hour : minute*
		| *hour : minute : second*
dst	→	*signed_time*
		| *signed_time ; dst_date , dst_date*
		| *; dst_date , dst_date*
dst_date	→	*julian*
		| *julian / time*
letter	→	*a* | *A* | *b* | *B* | ... | *z* | *Z*
hour	→	*00* | *01* | ... | *23*
minute	→	*00* | *01* | ... | *59*
second	→	*00* | *01* | ... | *59*
julian	→	*001* | *002* | ...| *366*
sign	→	− | +

EXAMPLES

The contents of **/etc/TIMEZONE** corresponding to the simple example below could be:

```
#       Time Zone
TZ=EST5EDT
export TZ
```

A simple setting for New Jersey could be:

TZ=EST5EDT

where **EST** is the abbreviation for the main time zone, **5** is the difference, in hours, between GMT (Greenwich Mean Time) and the main time zone, and **EDT** is the abbreviation for the alternate time zone.

The most complex representation of the same setting, for the year 1986, is

TZ="EST5:00:00EDT4:00:00;117/2:00:00,299/2:00:00"

where **EST** is the abbreviation for the main time zone, **5:00:00** is the difference, in hours, minutes, and seconds between GMT and the main time zone, **EDT** is the abbreviation for the alternate time zone, **4:00:00** is the difference, in hours, minutes, and seconds between GMT and the alternate time zone, **117** is the number of the day of the year (Julian day) when the alternate time zone will take effect, **2:00:00** is the number of hours, minutes, and seconds past midnight when the alternate time zone will take effect, **299** is the number of the day of the year when the alternate time zone will end, and **2:00:00** is the number of hours, minutes, and seconds past midnight when the alternate time zone will end.

A southern hemisphere setting such as the Cook Islands could be

TZ="KDT9:30KST10:00;64/5:00,303/20:00"

This setting means that **KDT** is the abbreviation for the main time zone, **KST** is the abbreviation for the alternate time zone, KST is **9** hours and **30** minutes later than GMT, KDT is **10** hours later than GMT, the starting date of KDT is the **64**th day at **5** AM, and the ending date of KDT is the **303**rd day at **8** PM.

Starting and ending times are relative to the alternate time zone. If the alternate time zone start and end dates and the time are not provided, the days for the United States that year will be used and the time will be 2 AM. If the start and end dates are provided but the time is not provided, the time will be midnight.

Note that in most installations, **TZ** is set to the correct value by default when the user logs on, via the local */etc/profile* file (see *profile*(4)).

NOTES

When the longer format is used, the **TZ** variable must be surrounded by double quotes as shown.

The system administrator must change the Julian start and end days annually if the longer form of the **TZ** variable is used.

Setting the time during the interval of change from the main time zone to the alternate time zone or vice versa can produce unpredictable results.

FILES

/etc/timezone

SEE ALSO

rc2(1M), profile(4), environ(5).
ctime(3C) in the *Programmer's Reference Manual.*

NAME

unistd – file header for symbolic constants

SYNOPSIS

#include <unistd.h>

DESCRIPTION

The header file <*unistd.h*> lists the symbolic constants and structures not already defined or declared in some other header file.

```
/*
 * POSIX defined symbols
 */

/* ANSI symbol mentioned in POSIX */
#define         NULL    0

/* Symbolic constants for the "access" function */
#define         R_OK    4       /* Test for Read permission */
#define         W_OK    2       /* Test for Write permission */
#define         X_OK    1       /* Test for execute permission */
#define         F_OK    0       /* Test for existence of File */

/* Symbolic constants for the "filno" function */
#define STDIN_FILENO 0
#define STDOUT_FILENO       1
#define STDERR_FILENO       2

/* Symbolic constants for the "lseek" function */
#define SEEK_SET 0 /* Set file pointer to "offset" */
#define SEEK_CUR 1 /* Set file pointer to current plus "offset" */
#define SEEK_END 2 /* Set file pointer to EOF plus "offset" */

/*
 * POSIX Compile-Time Symbolic Constants
 */
#define    _POSIX_SAVED_IDS      1
#define    _POSIX_VERSION        198808L

/*
 * POSIX Execution-Time Symbolic Constants
 */
#undef _POSIX_CHOWN_RESTRICTED
#undef _POSIX_NO_TRUNC
#undef _POSIX_VDISABLE
```

```
/*
 * POSIX Configurable System Variables
 */
#define  _SC_ARG_MAX      1 /* Bytes allowed for exec arguments */
#define  _SC_CHILD_MAX    2 /* Max child processes */
#define  _SC_CLK_TCK      3 /* Clock tick rate (HZ) */
#define  _SC_NGROUPS_MAX  4 /* Max multiple groups */
#define  _SC_OPEN_MAX     5 /* Max open files */
#define  _SC_JOB_CONTROL  6 /* Job control support */
#define  _SC_SAVED_IDS    7 /* saved-set-uid/gid support */
#define  _SC_VERSION      8 /* Posix version stamp */

/*
 * POSIX Configurable Pathname Variables
 */
#define    _PC_LINK_MAX           1
#define    _PC_MAX_CANON          2
#define    _PC_MAX_INPUT          3
#define    _PC_NAME_MAX           4
#define    _PC_PATH_MAX           5
#define    _PC_PIPE_BUF           6
#define    _PC_CHOWN_RESTRICTED   7
#define    _PC_NO_TRUNC           8
#define    _PC_VDISABLE           9

#define    STDIN_FILENO    0
#define    STDOUT_FILENO   1
#define    STDERR_FILENO   2

#ifndef _POSIX_SOURCE
/*
 * Non-POSIX symbols must be hidden by _POSIX_SOURCE
 */
#define F_ULOCK 0  /* Unlock a previously locked region */
#define F_LOCK  1  /* Lock a region for exclusive use */
#define F_TLOCK 2  /* Test and lock a region for exclusive use */
#define F_TEST  3  /* Test a region for other processes locks */

/* Path names */
#define GF_PATH "/etc/group"  /* Path name of the "group" file */
#define PF_PATH "/etc/passwd" /* Path name of the "passwd" file */
```

```
/* The following defines are specified in POSIX draft 12.0 and
are therefore  * necessary to compile the early NBS-PCTS
*/
#define _POSIX_GROUP_PARENT   0
#define _POSIX_CHOWN_SUP_GRP  0
#define _POSIX_DIR_DOTS       0
#define _POSIX_UTIME_OWNER    0

/*
 * BCS Configurable System Variables
 */
#define _SC_BCS_VERSION       9  /* BCS version stamp */
#define _SC_BCS_VENDOR_STAMP 10  /* Vendor stamp of system */
#define _SC_BCS_SYS_ID       11  /* unique machine id */
#define _SC_MAXUMEMV         12  /* Max user process */
                                 /* size 1-KB pages */
#define _SC_MAXUPROC         13  /* Max number of processes/user */
#define _SC_MAXMSGSZ         14  /* Max size of a message */
#define _SC_NMSGHDRS         15  /* Total number of msg /*
                                 /* headers/system */
#define _SC_SHMMAXSZ         16  /* Maximum size of shared segment
#define _SC_SHMMINSZ         17  /* Minimum size of shared segment
#define _SC_SHMSEGS          18  /* Max attached segs/process */
#define _SC_NMSYSSEM         19  /* Total number semaphores/system
#define _SC_MAXSEMVL         20  /* Max semaphore value */
#define _SC_NSEMMAP          21  /* Number of semaphore sets */
#define _SC_NSEMMSL          22  /* Number of semaphores/set */
#define _SC_NSHMMNI          23  /* Number of shared segments/syste
#define _SC_ITIMER_VIRT      24  /* System supports virtual timer *
#define _SC_ITIMER_PROF      25  /* System supports profiling timer
#define _SC_TIMER_GRAN       26  /* Granularity of timers in usec *
#define _SC_PHYSMEM          27  /* Total physical memory/system (k
#define _SC_AVAILMEM         28  /* Total physmem avail to user (kb
#define _SC_NICE             29  /* nice prioritization is supporte
#define _SC_MEMCTL_UNIT      30  /* bytes in a memory unit */
                                 /* in memctl system call */
#define _SC_SHMLBA           31  /* Memory address rounding used by
                                 /* shmsys in bytes */
#define _SC_SVSTREAMS        32  /* System V streams are supported
#define _SC_CPUID            33  /* return Processor Identification
                                 /* Register */


/*
 * BCS Configurable Pathname Variables
 */
#define _PC_BLKSIZE         10

/* Symbolic support for BCS requirements */

#define _BCS_VERSION      198902L    /* _SC_BCS_VERSION number */
```

```
#define _BCS_ITIMER_VIRT  1          /* Virtual timer support */
#define _BCS_ITIMER_PROF  1          /* Profiling timer support */
#define _BCS_NICE         1          /* Nice priorization support */
#define _BCS_SVSTREAMS    1          /* System V streams support */
#define _BCS_PTRACE_MAGIC 0x00088000 /* Ptrace_user magic number */
#define _BCS_PTRACE_REV   0x00000001 /* Ptrace_user version number */

/* ulimit symbolic constants (BCS) */

#define GET_ULIMIT      1
#define SET_ULIMIT      2
#define GET_BREAK       3
#define GET_MAX_OPEN    4

/* POSIX 12.0 symbols */
#define _PC_CHOWN_SUP_GRP    11
#define _PC_DIR_DOTS         12
#define _PC_GROUP_PARENT     13
#define _PC_UTIME_OWNER      14

#endif /* _POSIX_SOURCE */
```

NAME

utmp, wtmp – utmp and wtmp entry formats

SYNOPSIS

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

DESCRIPTION

These files, which hold user and accounting information for such commands as *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 */
 char   ut_host[24];     /* host name, if remote */
};
```

```
/*  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 FTP             128
#define REMOTE_LOGIN    129
#define REMOTE_PROCESS130
#define UTMAXTYPE       REMOTE_PROCESS /* 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"
```

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

getut(3C)
login(1), who(1), write(1) in the *User's Reference Manual*.

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

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
```

NAME

environ – user environment

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:

CFTIME

The default format string to be used by the *date*(1) command and the **ascftime**( ) and **cftime**( ) routines (see *ctime*(3C)). If **CFTIME** is not set or is null, the default format string specified in the **/lib/cftime/***LANGUAGE* file (if it exists) is used in its place (see *cftime*(4)).

CHRCLASS

A value that corresponds to a file in **/lib/chrclass** containing character classification and conversion information. This information is used by commands (such as *cat*(1), *ed*(1), *sort*(1), etc.) to classify characters as alphabetic, printable, uppercase, and to convert characters to upper- or lowercase.

When a program or command begins execution, the tables containing this information are initialized based on the value of **CHRCLASS**. If **CHRCLASS** is non-existent, null, set to a value for which no file exists in **/lib/chrclass**, or errors occur while reading the file, the ASCII character set is used. During execution, a program or command can change the values in these tables by calling the **setchrclass**() routine. For more detail, see *ctype*(3C).

These tables are created using the *chrtbl*(1M) command.

HOME

The name of the user's login directory, set by *login*(1) from the password file (see *passwd*(4)).

LANGUAGE

A language for which a printable file by that name exists in **/lib/cftime.** This information is used by commands (such as *date*(1), *ls*(1), *sort*(1), etc.) to print date and time information in the language specified.

If **LANGUAGE** is non-existent, null, set to a value for which no file exists in **/lib/cftime**, or errors occur while reading the file, the last language requested will be used. (If no language has been requested, the language **usa_english** is assumed.) For a description of the content of files in **/lib/cftime**, see *cftime*(4).

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**. (For more detail, see the "Execution" section of the *sh*(1) manual page.)

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), which may exploit special capabilities of that terminal.

TZ

Time zone information. The simplest format is **xxx***n***zzz** where **xxx** is the standard local time zone abbreviation, *n* is the difference in hours from GMT (Greenwich Mean Time), and **zzz** is the abbreviation for an alternate time zone (usually the daylight-saving local time zone), if any; for example,

TZ="EST5EDT"

The most complex format allows you to specify the difference in hours of the alternate time zone from GMT and the starting day and time and ending day and time for using this alternate time zone. For example, in 1985 the complex format corresponding to the above simple example is:

TZ="EST5:00:00EDT4:00:00;118/2:00:00,300/2:00:00"

When the above complex format is used, it must be surrounded by double quotes. For more details, see *ctime*(3C) and *timezone*(4).

Further names may be placed in the environment by the *export* command and "name=value" arguments in *sh*(1), 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 *profile*(4)).

NOTES

References to the *cftime*(4), *ctime*(3C), and *ctype(3C)* manual pages refer to programming capabilities available beginning with Issue 4.1 of the C Programming Language Utilities.

Administrators should note the following: if you attempt to set the current date to one of the dates that the standard and alternate time zones change (for example, the date that daylight time is starting or ending), and you attempt to set the time to a time in the interval between the end of standard time and the beginning of the alternate time (or the end of the alternate time and the beginning of standard time), the results are unpredictable.

SEE ALSO

chrtbl(1M), cftime(4), passwd(4), profile(4), timezone(4), in the *System Administrator's Reference Manual*.
exec(2), ctime(3C), ctype(3C) in the *Programmer's Reference Manual*.
cat(1), date(1), ed(1), env(1), ls(1), login(1), nice(1), nohup(1), sh(1) sort(1), time(1), vi(1) in the *User's Reference Manual*.

NAME

fcntl – file control options

SYNOPSIS

#include <fcntl.h>

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_ACCMODE 3      /* Get file stat flags
                            (POSIX 12.2 6.5.1.2.7) */
#define O_APPEND 010     /* append (writes guaranteed
                            at the end) */
#define O_NONBLOCK 0100 /* Non-blocking I/O
                            (POSIX 12.2 5.3.1.2) */

/* 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 */
#define O_NOCTTY          04000 /* Not a control tty
                                   (POSIX 12.2 5.3.1.2) */

/* 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 */
#define F_GETLK           5 /* Get file lock */
#define F_SETLK           6 /* Set file lock */
#define F_SETLKW          7 /* Set file lock and wait */

/* file segment locking control structure */
struct flock {
     short l_type;
     short l_whence;
     off_t l_start;
     off_t l_len;      /* if 0 then until EOF */
```

```
        short l_sysid;    /* returned with F_GETLK*/
}

/* file segment locking types */
#define F_RDLCK 01   /* Read lock */
#define F_WRLCK 02   /* Write lock */
#define F_UNLCK 03   /* Remove locks */

/* File descriptor flags (POSIX 12.2 6.5.1.2.2) */
#define  FD_CLOEXEC 1  /* Close the file descriptor upon
                          exec call */


#ifndef _POSIX_SOURCE
/*
 * Non-POSIX symbols must be hidden by _POSIX_SOURCE
 */

/* open(2) and fcntl(2) flags */
#define  O_NDELAY 04  /* Non-blocking I/O */
#define  O_SYNC   020 /* synchronous write option */
#ifndef  FASYNC
#define  FASYNC 040   /* signal pgroup when ready to read */
#endif


/* fcntl(2) requests */
#define  F_CHKFL  8   /* Check legality of file flag
                         changes */
#define  F_ALLOCSP 10 /* reserved */
#define  F_FREESP 11  /* reserved */

#endif  /* _POSIX_SOURCE */
```

SEE ALSO

fcntl(2), open(2) in the *Programmer's Reference Manual.*

NAME

math – math functions and constants

SYNOPSIS

#include <math.h>

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.

The following mathematical constants are defined for user convenience:

M_E	The base of natural logarithms (e).
M_LOG2E	The base-2 logarithm of e.
M_LOG10E	The base-10 logarithm of e.
M_LN2	The natural logarithm of 2.
M_LN10	The natural logarithm of 10.
M_PI	π, the ratio of the circumference of a circle to its diameter.
M_PI_2	$\pi/2$.
M_PI_4	$\pi/4$.
M_1_PI	$1/\pi$.
M_2_PI	$2/\pi$.
M_2_SQRTPI	$2/\sqrt{\pi}$.
M_SQRT2	The positive square root of 2.
M_SQRT1_2	The positive square root of 1/2.

For the definitions of various machine-dependent "constants," see the description of the **<values.h>** header file.

SEE ALSO

intro(3), matherr(3M), values(5)

NAME

prof – profile within a function

SYNOPSIS

#define MARK
#include <prof.h>

void MARK (*name*)

DESCRIPTION

MARK will introduce a mark called *name* that will be treated the same as a function entry point. Execution of the mark will add to a counter for that mark, and program-counter time spent will be accounted to the immediately preceding mark or to the function if there are no preceding marks within the active function.

name may be any combination of numbers or underscores. Each *name* in a single compilation must be unique, but may be the same as any ordinary program symbol.

For marks to be effective, the symbol MARK must be defined before the header file **<prof.h>** is included. This may be defined by a preprocessor directive as in the synopsis, or by a command line argument, for example:

cc –p –DMARK foo.c

If MARK is not defined, the *MARK*(name) statements may be left in the source files containing them and will be ignored.

EXAMPLE

In this example, marks can be used to determine how much time is spent in each loop. Unless this example is compiled with *MARK* defined on the command line, the marks are ignored.

```
#include <prof.h>
foo( )
{
        int i, j;
        .
        .
        .
        MARK(loop1);
        for (i = 0; i < 2000; i++) {
                . . .
        }
```

```
        MARK(loop2);
        for (j = 0; j < 2000; j++) {
                . . .
        }
}
```

SEE ALSO

prof(1), profil(2), monitor(3C)

NAME

regexp – 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;*

extern char *loc1, *loc2, *locs;

extern int circf, sed, nbra;

DESCRIPTION

This page describes general-purpose regular expression matching routines in the form of *ed*(1), defined in **<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 unpleasantly 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 that 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 where the compiled regular expression may be placed. 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 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 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 *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();
```

SEE ALSO

ed(1), expr(1), grep(1), sed(1) in the *User's Reference Manual.*

NAME

stat – data returned by stat system call

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>

DESCRIPTION

The system calls *stat* and *fstat* 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;         /* New type (POSIX 12.2 5.6.1.2) */
mode_t   st_mode;        /* New type (POSIX 12.2 5.6.1.2) */
nlink_t  st_nlink;       /* New type (POSIX 12.2 5.6.1.2) */
uid_t    st_uid;         /* New type (POSIX 12.2 5.6.1.2) */
gid_t    st_gid;         /* New type (POSIX 12.2 5.6.1.2) */
dev_t    st_rdev;
off_t    st_size;
time_t   st_atime;
time_t   st_ausec;       /* atime extra usecs (BCS 9.13.3) */
time_t   st_mtime;
time_t   st_musec;       /* mtime extra usecs (BCS 9.13.3) */
time_t   st_ctime;
time_t   st_cusec;       /* ctime extra usecs (BCS 9.13.3) */
charst_pad[456];         /* BCS 9.13.3 */
};

/*
 * permission bits from st_mode
 */
#define S_IRWXU 00700  /* read, write, execute: owner */
#define S_IRUSR 00400  /* read permission: owner */
#define S_IWUSR 00200  /* write permission: owner */
#define S_IXUSR 00100  /* execute permission: owner */
#define S_IRWXG 00070  /* read, write, execute: group */
#define S_IRGRP 00040  /* read permission: group */
#define S_IWGRP 00020  /* write permission: group */
#define S_IXGRP 00010  /* execute permission: group */
#define S_IRWXO 00007  /* read, write, execute: other */
#define S_IROTH 00004  /* read permission: other */
#define S_IWOTH 00002  /* write permission: other */
```

```
#define S_IXOTH 00001  /* execute permission: other */
#define S_ISUID 04000  /* set user id on execution */
#define S_ISGID 02000  /* set group id on execution */
#define S_ISDIR(m)  (m & S_IFDIR) /* True if directory */
#define S_ISCHR(m)  (m & S_IFCHR) /* True if character special
                                     file */
#define S_ISBLK(m)  (m & S_IFBLK) /* True if block special
                                      file */
#define S_ISREG(m)  (m & S_IFREG) /* True if regular file */
#define S_ISFIFO(m)  (m & S_IFIFO) /* True if pipe or FIFO
                                    special file */
#ifndef _POSIX_SOURCE
/*
 * Non-POSIX symbols must be hidden by _POSIX_SOURCE
 */
#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_IFLNK 0120000 /* symbolic link */
   /* 0120000 = 0xA000 in inode.h */
#define  S_IFIFO 0010000 /* fifo */
#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 */
#define S_ENFMT S_ISGID  /* record locking enforcement
                                  flag */
#endif /* _POSIX_SOURCE */
```

SEE ALSO

stat(2), types(5)

NAME

term – conventional names for terminals

DESCRIPTION

These names are used by certain commands (e.g., *man*(1), *tabs*(1), *tput*(1), *vi*(1) and *curses*(3X)) and are maintained as part of the shell environment in the environment variable **TERM** (see *sh*(1), *profile*(4), and *environ*(5)).

Entries in *terminfo*(4) source files consist of a number of comma-separated fields. (To obtain the source description for a terminal, use the **–I** option of *infocmp*(1M).) White space after each comma is ignored. The first line of each terminal description in the *terminfo*(4) database gives the names by which *terminfo*(4) knows the terminal, separated by bar (|) characters. The first name given is the most common abbreviation for the terminal (this is the one to use to set the environment variable **TERMINFO** in **$HOME/.profile**; see *profile*(4)), 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 contain no blanks and must be unique in the first 14 characters; the last name may contain 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, for example, for the AT&T 4425 terminal, **att4425**. This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Up to 8 characters, chosen from [a–z0–9], make up a basic terminal name. 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. Terminal sub-models, operational modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. Thus, an AT&T 4425 terminal in 132 column mode would be **att4425–w**. The following suffixes should be used where possible:

Suffix	Meaning	Example
–w	Wide mode (more than 80 columns)	att4425–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
–np	Number of pages of memory	c100–4p
–rv	Reverse video	att4415–rv

To avoid conflicts with the naming conventions used in describing the different modes of a terminal (e.g., **–w**), it is recommended that a terminal's root name not contain hyphens. Further, it is good practice to make all terminal names used in the *terminfo*(4) database unique. Terminal entries that are present only for inclusion in other entries via the **use=** facilities should have a '+' in their name, as in **4415+nl**.

Some of the known terminal names may include the following (for a complete list, type: **ls -C /usr/lib/terminfo/?**):

155	Motorola EXORterm 155
2621,hp2621	Hewlett-Packard 2621 series
2631	Hewlett-Packard 2631 line printer
2631–c	Hewlett-Packard 2631 line printer - compressed mode
2631–e	Hewlett-Packard 2631 line printer - expanded mode
2640,hp2640	Hewlett-Packard 2640 series
2645,hp2645	Hewlett-Packard 2645 series
3270	IBM Model 3270
33,tty33	AT&T Teletype Model 33 KSR
35,tty35	AT&T Teletype Model 35 KSR
37,tty37	AT&T Teletype Model 37 KSR
4000a	Trendata 4000a
4014,tek4014	TEKTRONIX 4014
40,tty40	AT&T Teletype Dataspeed 40/2
43,tty43	AT&T Teletype Model 43 KSR
4410,5410	AT&T 4410/5410 terminal in 80-column mode - version 2
4410–nfk,5410–nfk	AT&T 4410/5410 without function keys - version 1
4410–nsl,5410–nsl	AT&T 4410/5410 without pln defined
4410–w,5410–w	AT&T 4410/5410 in 132-column mode
4410v1,5410v1	AT&T 4410/5410 terminal in 80-column mode - version 1
4410v1–w,5410v1–w	AT&T 4410/5410 terminal in 132-column mode - version 1
4415,5420	AT&T 4415/5420 in 80-column mode
4415–nl,5420–nl	AT&T 4415/5420 without changing labels
4415–rv,5420–rv	AT&T 4415/5420 80 columns in reverse video
4415–rv–nl,5420–rv–nl	AT&T 4415/5420 reverse video without changing labels
4415–w,5420–w	AT&T 4415/5420 in 132-column mode
4415–w–nl,5420–w–nl	AT&T 4415/5420 in 132-column mode without changing labels
4415–w–rv,5420–w–rv	AT&T 4415/5420 132 columns in reverse video
4415–w–rv–nl,5420–w–rv–nl	AT&T 4415/5420 132 columns reverse video without changing labels
4418,5418	AT&T 5418 in 80-column mode

4418–w,5418–w	AT&T 5418 in 132-column mode
4420	AT&T Teletype Model 4420
4424	AT&T Teletype Model 4424
4424-2	AT&T Teletype Model 4424 in display function group ii
4425,5425	AT&T 4425/5425
4425–fk,5425–fk	AT&T 4425/5425 without function keys
4425–nl,5425–nl	AT&T 4425/5425 without changing labels in 80-column mode
4425–w,5425–w	AT&T 4425/5425 in 132-column mode
4425–w–fk,5425–w–fk	AT&T 4425/5425 without function keys in 132-column mode
4425–nl–w,5425–nl–w	AT&T 4425/5425 without changing labels in 132-column mode
4426	AT&T Teletype Model 4426S
450	DASI 450 (same as Diablo 1620)
450–12	DASI 450 in 12-pitch mode
500,att500	AT&T-IS 500 terminal
510,510a	AT&T 510/510a in 80-column mode
513bct,att513	AT&T 513 bct terminal
5320	AT&T 5320 hardcopy terminal
5420_2	AT&T 5420 model 2 in 80-column mode
5420_2–w	AT&T 5420 model 2 in 132-column mode
5620,dmd	AT&T 5620 terminal 88 columns
5620–24,dmd–24	AT&T Teletype Model DMD 5620 in a 24x80 layer
5620–34,dmd–34	AT&T Teletype Model DMD 5620 in a 34x80 layer
610,610bct	AT&T 610 bct terminal in 80-column mode
610–w,610bct–w	AT&T 610 bct terminal in 132-column mode
7300,pc7300,unix_pc	AT&T UNIX PC Model 7300
735,ti	Texas Instruments TI735 and TI725
745	Texas Instruments TI745
dumb	generic name for terminals that lack reverse line-feed and other special escape sequences
hp	Hewlett-Packard (same as 2645)
lp	generic name for a line printer
pt505	AT&T Personal Terminal 505 (22 lines)
pt505–24	AT&T Personal Terminal 505 (24-line mode)
sync	generic name for synchronous Teletype Model 4540-compatible terminals
vt100	DEC VT100

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*.

FILES

/usr/lib/terminfo/?/* compiled terminal description database

SEE ALSO

curses(3X), profile(4), terminfo(4), environ(5)
man(1), sh(1), stty(1), tabs(1), tput(1), tplot(1G), vi(1) in the *User's Reference Manual.*
infocmp(1M) in the *System Administrator's Reference Manual*.
Chapter 10 of the *Programmer's Guide.*

NOTES

Not all programs follow the above naming conventions.

NAME

types – primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

The data types defined in the include file are used in the system code; some data of these types are accessible to user code:

```
/*
 * POSIX defined symbols
 */
typedef  unsigned long   dev_t;      /* device number */
typedef  unsigned long   gid_t;      /* group ID's */
typedef  unsigned long   ino_t;      /* file serial number */
typedef  unsigned long   mode_t;     /* file attributes */
typedef unsigned long    nlink_t;    /* link counts */
typedef  long            off_t;      /* file sizes */
typedef  long            pid_t;      /* process IDs and process
                                                group IDs */
typedef  unsigned long   uid_t;      /* user IDs */
typedef  unsigned long   clock_t;    /* intervals per second */
typedef  long            time_t;     /* time */

#ifndef _POSIX_SOURCE
/*
 * Non-POSIX symbols must be hidden if _POSIX_SOURCE is defined
 */
typedef  struct { int r[1]; } *physadr;
typedef  long            daddr_t;    /* <disk address> type */
typedef  char            *caddr_t;   /* ?<core address> type */
typedef  unsigned char   unchar;
typedef  unsigned short  ushort;
typedef  unsigned int    uint;
typedef  unsigned long   ulong;
typedef  short           cnt_t;      /* ?<count> type */
#define  LABELSIZE       24
typedef  int             label_t[LABELSIZE];
typedef  long            paddr_t;    /* <physical address> type */
typedef  long            key_t;      /* IPC key type */
typedef  unsigned short  use_t;      /* use count for swap.  */
typedef  short           sysid_t;
typedef  short           index_t;
typedef  short           lock_t;     /* lock work for busy wait */
```

```
typedef  unsigned int    size_t;     /* len param for
                                           string funcs */

/*
 * Distributed UNIX hook
 */
typedef struct cookie {
         long            c_sysid;
         long            c_rcvd;
} *cookie_t;

/*
 * Select defines per Berk 4.3 MAN
 */
#define  NBBY            8    /* number of bits in a byte */
/*
 * Select uses bit masks of file descriptors in longs.
 * These macros manipulate such bit fields (the filesystem macros
   use chars).
 * FD_SETSIZE may be defined by the user, but the default here
 * should be >= NOFILE (param.h).
 */
#ifndef  FD_SETSIZE
#define  FD_SETSIZE      256
#endif

typedef long             fd_mask;
#define NFDBITS          (sizeof(fd_mask)*NBBY)/* bits per mask */
#ifndef howmany
#define  howmany(x, y)   (((x)+((y)-1))/(y))
#endif

typedef  struct fd_set {
         fd_mask         fds_bits[howmany(FD_SETSIZE, NFDBITS)];
} fd_set;

#define  FD_SET(n, p)    ((p)->fds_bits[(n)/NFDBITS]
                                   |= (1 << ((n) % NFDBITS)))
#define  FD_CLR(n, p)    ((p)->fds_bits[(n)/NFDBITS]
                                   &= ~(1 << ((n) % NFDBITS)))
#define  FD_ISSET(n, p)  ((p)->fds_bits[(n)/NFDBITS]
                                   & (1 << ((n) % NFDBITS)))
#ifdef INKERNEL
#define FD_ZERO(p)       bzero((caddr_t)p, sizeof(*(p)))
#else
```

```
#define FD_ZERO(p)       memset((caddr_t)p, 0, sizeof(*(p)))
#endif

#endif   /* _POSIX_SOURCE */
#ifndef __ULNG__
#endif
/* these macros are used for device drivers to keep aligned */

typedef struct ulng
{
         unsigned short  hi;          /* High word */
         unsigned short  lo;          /* Low word */
} ULNG;

         /*
         **   These macro defines will set and get long values.
         */

#define  SETULNG(p, v) {  \
     p.hi = ((unsigned long)v >> 16); \
     p.lo = (unsigned short)v;   \
   }
#define GETULNG(p) ((p.hi << 16) | p.lo)
#endif
#define __ULNG__
#endif
```

The form *daddr_t* is used for disk addresses except in an i-node on disk, see *fs*(4). 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.

SEE ALSO

fs(4)

NAME

values – machine-dependent values

SYNOPSIS

#include <values.h>

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.

SEE ALSO

intro(3), math(5)

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), vprintf(3S)

NOTES

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.

MOTOROLA INC.

Microcomputer Division
2900 South Diablo Way
Tempe, Arizona 85282
P.O. Box 2953
Phoenix, Arizona 85062

11046 PRINTED IN USA (3/90) WPC 2,500