UNISYS

U 6000 Series
System V
Programmer's Reference Manual
Volume 2

March 1988

Printed in U S America
UP–13712.3

Priced Item

This document is intended for software releases based on AT&T Release 3 of UNIX System V or a subsequent release of the System unless otherwise indicated.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

6000/50
Customization Package

This customization package contains changes to your *Programmer's Reference Manual, Volume 2,* which reflect the *AT&T System V Release 3.1* and the value-added features of the *Unisys System V Operating System*. Please add these pages to your base manual to produce a fully customized *Programmer's Reference Manual*.

To fully customize your Programmer's Reference Manual, Volume 1, replace the existing generic manual cover with the new customized manual cover.

Customization Directions

The table below indicates the name of the document and directions for making your manual current. The document found in your customization package may be used to replace an already existing document of the same name, or it may be added as a new document to the manual. You may also be directed to remove an existing document from the current manual. For the location of specific documents, please refer to the Table of Contents.

New Customization Document	*Customization Directions*
Table of Contents	replace old with new
intro(3)	replace old with new
dbm(3B)	add new
ndbm(3B)	add new
ctime(3C)	replace old with new
ctype(3C)	replace old with new
dial(3C)	replace old with new
fpgetround(3C)	replace old with new
crypt(3X)	replace old with new
curses(3X)	replace old with new
libdev(3X)	add new
ocurse(3X)	add new
otermcap(3X)	add new
sputl(3X)	add new

Table of Contents

(The following are contained in three volumes.)

4. File Formats

NAME

intro - introduction to functions and libraries

DESCRIPTION

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

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

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

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

(3N) This contains sets of functions constituting the Network Services library. These sets provide protocol independent interfaces to networking services based on the service definitions of the OSI (Open Systems Interconnection) reference model. Application developers access the function sets that provide services at a particular level.

This library contains the functions of the TRANSPORT INTERFACE (TI) - provide the services of the OSI Transport Layer. These services provide reliable end-to-end data transmission using the services of an underlying

network. Applications written using the TI functions are independent of the underlying protocols. Declarations for these functions may be obtained from the **#include** file **<tiuser.h>**. The link editor *ld*(1) searches this library under the **-lnsl_s** option.

(3B) These functions are part of the System V BSD Berkeley networking package. To use these functions you must have the network protocols on your system.

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

DEFINITIONS

character
Any bit pattern able to fit into a byte on the machine.

null character
A character with value 0, represented in the C language as '\0'.

character array
A sequence of characters.

null-terminated character array
A sequence of characters, the last of which is the *null character*.

string
A designation for a *null-terminated character array*.

null string
A character array containing only the null character.

NULL pointer
The value that is obtained by casting **0** into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error.

NULL
Defined as **0** in **<stdio.h>**; the user can include an appropriate definition if not using **<stdio.h>**.

Netbuf
In the Network Services library, *netbuf* is a structure used in various Transport Interface (TI) functions to send and receive data and information. It contains the following members:

```
unsigned int maxlen;
unsigned int len;
char *buf;
```

Buf points to a user input and/or output buffer. *Len* generally specifies the number of bytes contained in the buffer. If the structure is used for both input and output, the function will replace the user value of *len* on return.

Maxlen generally has significance only when *buf* is used to receive output from the TI function. In this case, it specifies the physical size of the buffer, the maximum value of *len* that can be set by the function. If *maxlen* is not large enough to hold the returned information, an TBUFOVFLW error will generally result. However, certain functions may return part of the data and not generate an error.

FILES

/lib
/lib/libc.a
/lib/libc_s.a
/lib/libm.a
/shlib/libc_s
/shlib/libnsl_s (3N)
/usr/lib/libnsl_s.a (3N)

SEE ALSO

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

DIAGNOSTICS

Functions in the C and Math Libraries (3C and 3M) may return the conventional values **0** or ± **HUGE** (the largest-magnitude single-precision floating-point numbers; **HUGE** is defined in the **<math.h>** header file) when the function is undefined for

the given arguments or when the value is not representable. In these cases, the external variable *errno* [see *intro*(2)] is set to the value EDOM or ERANGE.

WARNING

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

NAME

dbminit, fetch, store, delete, firstkey, nextkey - data base subroutines

SYNOPSIS

```
#include <dbm.h>

typedef struct {
    char *dptr;
    int dsize;
} datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions handle very large (a billion blocks) databases and access a keyed item in one or two file system accesses. The functions are obtained with the loader option **-ldbm**.

Keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

Before a database can be accessed, it must be opened by *dbminit*. At the time of this call, the files *file*.**dir** and *file*.**pag** must exist. An empty database is created by creating zero-length **.dir** and **.pag** files.

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. A key (and its

associated contents) is deleted by *delete*. A linear pass through all keys in a database may be made, in (apparently) random order, by use of *firstkey* and *nextkey*. *Firstkey* returns the first key in the database. With any key *nextkey* returns the next key in the database. This code traverses the data base:

```
for (key = firstkey( ); key.dptr != NULL;
key = nextkey(key))
```

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates OK. Routines that return a *datum* indicate errors with a null (0) *dptr*.

BUGS

The **.pag** file contains holes so that its apparent size is about four times its actual content. Other UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (**cp**, **cat**, **tp**, **tar**, **ar**) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. *Store* returns an error in the event that a disk block fills with inseparable data.

Delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *firstkey* and *nextkey* depends on a hashing function, not on anything interesting.

NAME

dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr - data base subroutines

SYNOPSIS

```
#include <ndbm.h>

typedef struct {
    char *dptr;
    int dsize;
} datum;

DBM *dbm_open(file, flags, mode)
char *file;
int flags, mode;

void dbm_close(db)
DBM *db;

datum dbm_fetch(db, key)
DBM *db;
datum key;

int dbm_store(db, key, content, flags)
DBM *db;
datum key, content;
int flags;

int dbm_delete(db, key)
DBM *db;
datum key;

datum dbm_firstkey(db)
DBM *db;

datum dbm_nextkey(db)
DBM *db;

int dbm_error(db)
DBM *db;

int dbm_clearerr(db)
DBM *db;
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions handle very large databases (a billion blocks) and access a keyed item in one or two file system accesses.

This package replaces, and is incompatible with, the earlier *dbm*(3B) library, which managed only a single database.

Keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

Before a database can be accessed, it must be opened by *dbm_open*. This opens and/or creates the files *file*.**dir** and *file*.**pag** depending on the flags parameter [see *open*(2)].

Once open, the data stored under a key is accessed by *dbm_fetch* and data is placed under a key by *dbm_store*. The *flags* field can be either DBM_INSERT or DBM_REPLACE. DBM_INSERT only inserts new entries into the database and does not change an existing entry with the same key. DBM_REPLACE replaces an existing entry if it has the same key. A key (and its associated contents) is deleted by *dbm_delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *dbm_firstkey* and *dbm_nextkey*. *Dbm_firstkey* returns the first key in the database. *Dbm_nextkey* returns the next key in the database. This code traverses the database:

```
for (key = dbm_firstkey(db); key.dptr != NULL;
key = dbm_nextkey(db))
```

Dbm_error returns non-zero when an error has occurred reading or writing the database. *Dbm_clearerr* resets the error condition on the named database.

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates no error condition. Routines that return a *datum* indicate errors with a null (0) *dptr*. If *dbm_store* is called with a *flags* value of DBM_INSERT, and finds an existing entry with the same key, it returns 1.

WARNINGS

The **.pag** file contains holes so that its apparent size is about four times its actual content. These files cannot be copied by normal means (*cp*, *cat*, *tp*, *tar*, *ar*) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 4096 bytes). Moreover all key/content pairs that hash together must fit in a single block. *Dbm_store* returns an error in the event that a disk block fills with inseparable data.

Dbm_delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *dbm_firstkey* and *dbm_nextkey* depends on a hashing function, not on anything interesting.

SEE ALSO

dbm(3B).

NOTE

This function is for use with a version of the System V kernel that supports networking protocols.

[This page left blank.]

NAME

ctime, localtime, gmtime, asctime, tzset - convert date and time to string

SYNOPSIS

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

char *ctime (clock)
time_t *clock;

struct tm *localtime (clock)
time_t *clock;

struct tm *gmtime (clock)
time_t *clock;

char *asctime (tm)
struct tm *tm;

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset ()
```

DESCRIPTION

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

```
Sun Sep 16 01:03:52 1985\n\0
```

Localtime and *gmtime* return pointers to **tm** structures, described below. *Localtime* corrects for the time zone and possible Daylight Savings Time; *gmtime* converts directly to Greenwich Mean Time (GMT), which is the time the System V system uses.

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

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

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

Tm_isdst is non-zero if Daylight Savings Time is in effect.

The external **long** variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, *timezone* is 5*60*60); the external variable *daylight* is non-zero if and only if the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

If an environment variable named **TZ** is present, *asctime* uses the contents of the variable to override the default time zone. The value of **TZ** must be a three-letter time zone name, followed by a number representing the difference between local time and Greenwich Mean Time in hours, followed by an optional three-letter name for a daylight time zone. For example, the setting for New Jersey would be **EST5EDT**. The effects of setting **TZ** are thus to change the values of the external variables *timezone* and *daylight*; in addition, the time zone names contained in the external variable

```
char *tzname[2] = { "EST", "EDT" };
```

are set from the environmentvariable TZ. The function *tzset* sets these external variables from **TZ**; *tzset* is called by *asctime* and may also be called explicitly by the user.

Note that in most installations, **TZ** is set by default when the user logs on, to a value in the local **/etc/profile** file [see *profile*(4)].

SEE ALSO

time(2), getenv(3C), profile(4), environ(5).

CAVEAT

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

[This page left blank.]

NAME

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

SYNOPSIS

#include <ctype.h>

int isalpha (c)
int c;

...

DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *Isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value **EOF** [-1; see *stdio*(3S)].

isalpha — *c* is a letter.

isupper — *c* is an upper-case letter.

islower — *c* is a lower-case letter.

isdigit — *c* is a digit [0-9].

isxdigit — *c* is a hexadecimal digit [0-9], [A-F] or [a-f].

isalnum — *c* is an alphanumeric (letter or digit).

isspace — *c* is a space, tab, carriage return, newline, vertical tab, or form-feed.

ispunct — *c* is a punctuation character (neither control nor alphanumeric).

isprint — *c* is a printing character, code 040 (space) through 0176 (tilde).

isgraph — *c* is a printing character, like *isprint* except false for space.

iscntrl — *c* is a delete character (0177) or an ordinary control character (less than 040).

isascii — *c* is an ASCII character, code less than 0200.

SEE ALSO

stdio(3S), ascii(5).

DIAGNOSTICS

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

NAME

dial - establish an out-going terminal line connection

SYNOPSIS

#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
int fd;

DESCRIPTION

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

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

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

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

The CALL element *speed* is intended only for use with an out-going dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high- or low-speed

setting on the 212A modem. Note that the 113A modem or the low-speed setting of the 212A modem will transmit at any rate between 0 and 300 bits per second. However, the high-speed setting of the 212A modem transmits and receives at 1200 bits per secound only. The CALL element *baud* is for the desired transmission baud rate. For example, one might set *baud* to 110 and *speed* to 300 (or 1200). However, if **speed** set to 1200 **baud** must be set to high (1200).

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

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. The termination symbol will be supplied by the *dial* function, and should not be included in the *telno* string passed to *dial* in the CALL structure.

The CALL element *modem* is used to specify modem control for direct lines. This element should be non-zero if modem control is required. The CALL element *attr* is a pointer to a *termio* structure, as defined in the **<termio.h>** header file. A NULL value for this pointer element may be passed to the *dial* function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud-rate.

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

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

FILES

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

SEE ALSO

alarm(2), read(2), write(2).
termio(7) in the *Administrator's Reference Manual*.
uucp(1C) in the *User's Reference Manual*.

DIAGNOSTICS

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

```
INTRPT   -1    /* interrupt occurred */
D_HUNG   -2    /* dialer hung (no return from write) */
NO_ANS   -3    /* no answer within 10 seconds */
ILL_BD   -4    /* illegal baud-rate */
A_PROB   -5    /* acu problem (open() failure) */
L_PROB   -6    /* line problem (open() failure) */
NO_Ldv   -7    /* can't open LDEVS file */
DV_NT_A  -8    /* requested device not available */
DV_NT_K  -9    /* requested device not known */
NO_BD_A  -10   /* no device available at requested baud */
NO_BD_K  -11   /* no device known at requested baud */
DV_NT_E  -12   /* requested speed does not match */
```

WARNINGS

The *dial*(3C) library function is not compatible with Basic Networking Utilities on UNIX System V Release 2.0.

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

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

BUGS

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

[This page left blank.]

NAME

fpgetround, fpsetround, fpgetmask, fpsetmask, fpgetsticky, fpsetsticky - IEEE floating point environment control

SYNOPSIS

```
#include <ieeefp.h>

typedef enum {
    FP_RN=0,    /* round to nearest */
    FP_RZ=0x10, /* round to zero (truncate) */
    FP_RM=0x20,          /* round to minus */
    FP_RP=0x30, /* round to plus */
    } fp_rnd;

fp_rnd fpgetround();

fp_rnd fpsetround(rnd_dir)
fp_rnd rnd_dir;

#define   fp_except     int
#define   FP_X_INV      0x80  /* invalid operation */
                              /* exception */
#define   FP_X_OFL      0x40  /* overflow */
                              /* exception */
#define   FP_X_UFL      0x20  /* underflow */
                              /* exception */
#define   FP_X_DZ       0x10  /* divide-by-zero */
                              /* exception */
#define   FP_X_IMP      0x08  /* imprecise (loss */
                              /* of precision) */

fp_except fpgetmask();

fp_except fpsetmask(mask);
fp_except mask;

fp_except fpgetsticky();

fp_except fpsetsticky(sticky);
fp_except sticky;
```

DESCRIPTION

These routines let the user change the behavior on occurrence of any of five floating point exceptions: divide-by-zero, overflow, underflow, imprecise (inexact) result, and invalid operation. The routines also change the rounding mode for floating point operations. When a floating point exception occurs, the

corresponding sticky bit is set (1), and if the mask bit is enabled (1), the trap takes place. The routines are valid only on systems that are equipped with floating point accelerator hardware; otherwise, floating point operations are compiled differently and handled in software.

fpgetround() returns the current rounding mode.

fpsetround() sets the rounding mode and returns the previous rounding mode.

fpgetmask() returns the current exception masks.

fpsetmask() sets the exception masks and returns the previous setting.

fpgetsticky() returns the current exception sticky flags.

fpsetsticky() sets (clears) the exception sticky flags and returns the previous setting.

The environment for Convergent computers that combine the MC68020 CPU with the MC68881 or MC68882 floating point processor is:

- Rounding mode set to nearest(FP_RN),
- Divide-by-zero,
- Floating point overflow, and
- Invalid operation traps enabled.

SEE ALSO

isnan(3C).

WARNINGS

fpsetsticky() modifies all sticky flags. *fpsetmask*() changes all mask bits.

C requires truncation (round to zero) for floating point to integral conversions. The current rounding mode has no effect on these conversions.

CAVEATS

The utilities described in this manual page are applicable only for computers that are equipped with both the MC68020 microprocessor for the CPU and the MC68881 or MC68882 microprocessor for a hardware floating point accelerator. Programs that invoke these utilities that are run on computers without the floating point hardware result in no operation and no returned error message for the particular function.

One must clear the sticky bit to recover from the trap and to proceed. If the sticky bit is not cleared before the next trap occurs, a wrong exception type may be signaled.

For the same reason, when calling *fpsetmask*() the user should make sure that the sticky bit corresponding to the exception being enabled is cleared.

[This page left blank.]

NAME

crypt - password and file encryption functions

SYNOPSIS

cc [flag ...] file ... -lcrypt [library ...]

```
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 the only version supplied with System V, as part of the *C Programming 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.

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).
login(1), passwd(1) in the *User's Reference Manual*.

CAVEATS

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

Only the international version of *crypt*(3X) is supplied with System V. The domestic version is not available.

NAME

curses - terminal screen handling and optimization package

OVERVIEW

The *curses* manual page is organized as follows:

In *SYNOPSIS*:

- Compiling information.
- Summary of parameters used by *curses* routines.
- Alphabetical list of *curses* routines, showing their parameters.

In *DESCRIPTION*:

- An overview of how *curses* routines should be used.

In *ROUTINES*, descriptions of each *curses* routine are grouped under the appropriate topics:

- Overall Screen Manipulation.
- Window and Pad Manipulation.
- Output Routines.
- Input Routines.
- Output Options Setting.
- Input Options Setting.
- Environment Queries.
- Soft Label Routines.
- Low-level Curses Access.
- Terminfo-Level Manipulations.
- Termcap Emulation.
- Miscellaneous.
- Use of **curscr.**
- Obsolete calls.

Then come sections on:

- *ATTRIBUTES.*

- *FUNCTION KEYS*.
- *LINE GRAPHICS*.

SYNOPSIS

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

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

The parameters in the following list are not global variables, but rather this is a summary of the parameters used by the *curses* library routines. All routines return the **int** values **ERR** or **OK** unless otherwise noted. Routines that return pointers always return **NULL** on error. (**ERR**, **OK**, and **NULL** are all defined in **<curses.h>**.) Routines that return integers are not listed in the parameter list below.

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

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

addch(ch)
addstr(str)
attroff(attrs)
attron(attrs)
attrset(attrs)
baudrate()
beep()
box(win, vertch, horch)
cbreak()
clear()
clearok(win, bf)
clrtobot()
clrtoeol()
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)
getstr(str)
getsyx(y, x)
getyx(win, y, x)
halfdelay(tenths)
has_ic()
has_il()
idlok(win, bf)
inch()
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)
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)
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)
set_curterm(nterm)
set_term(new)
setscrreg(top, bot)
setsyx(y, x)
setupterm(term, fildes, errret)
slk_clear()
slk_init(fmt)
slk_label(labnum)
slk_noutrefresh()
slk_refresh()
slk_restore()
slk_set(labnum, label, fmt)
slk_touch()
standend()
standout()
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)
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.

In order to initialize the routines, 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. This can be done by executing the *tput* **init** command after the shell environment variable **TERM** has been 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 C 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**, 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 show up in modes such as underlined or 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**.

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 in 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 filled in by **initscr**() with the size of the screen. (For more information, see the subsection

"Terminfo-Level Manipulations.") 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.

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 **refresh()** 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.

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 is 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 the window to be 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 both windows. When using this routine, often it will be necessary to call **touchwin**() or **touchline**() on *orig* before calling **wrefresh**().

delwin(win)
Delete the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

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 (for example, 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 both windows. When using this routine, often it will be necessary to call **touchwin**() or **touchline**() on *orig* before calling **prefresh**().

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.

Output Routines

These routines are used to "draw" text on 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 **clrtoeol**() 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**() after adding a control character will not return the control character, but instead will return 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 **inch**() and **addch**().) See **standout**(), 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 **A_STANDOUT**, **A_REVERSE**, **A_BOLD**, **A_DIM**, **A_BLINK**, **A_UNDERLINE**, and **A_ALTCHARSET**. These constants 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. **Attrset**(attrs) sets the current attributes of the given window to *attrs*. **Attroff**(attrs) turns off the named attributes without turning on or off any other attributes. **attron**(attrs) turns on the named attributes without affecting any others. **Standout**() is the same as **attron**(A_STANDOUT). **Standend**() is the same as **attrset(0)**, that is, it turns off all attributes.

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 terminal 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*. This is implemented as a macro, so no **"&"** is necessary before the variables.

getbegyx(win, y, x)
getmaxyx(win, y, x)
Like **getyx**(), these routines store the current beginning coordinates and size of the specified window.

Note that **getbegyx**() and **getmaxyx**() are macros.

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, possibly 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 **refresh**() 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 *srcwin* on top of *dstwin*; that is, all text in *srcwin* is copied into *dstwin*. *scrwin* and *dstwin* need not be the same size; only text where the two windows overlap is copied. The difference is that **overlay**() is non-destructive (blanks are not copied), while **overw**() is destructive.

copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)
This routine provides a finer grain of control over the **overlay**() and **overwrite**() routines. Like 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**(3S). The string which would be output by **printf**(3S) is instead output using **waddstr**() on the given window.

vwprintw(win, fmt, varglist)
This routine corresponds to *vfprintf* [see *vprintf*(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. As an optimization, if the window is **stdscr** and the scrolling region is the entire window, the physical screen will be scrolled at the same time.

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* .

Input Routines

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. No **refresh**() will occur between the **move**() and the **getch**() done within the routines **mvgetch**() and **mvwgetch**().

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 **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 **getch**() is made, until a newline, carriage return, or enter key is received. The resulting value is placed in the area pointed at by the character pointer *str*. The user's erase and kill characters are interpreted. As in **mvgetch**(), no **refresh**() is done between the **move**() and **getstr**() within the routines **mvgetstr**() and **mvwgetstr**().

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

flushinp()

Throws away any typeahead that has been typed by the user and has not yet been read by the program.

ungetch(c)

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

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.

vwscanw(win, fmt, ap)

This routine is similar to **vwprintw**() above in that 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.

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**().

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.

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**() and **wsetscrreg**() are macros.

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**().)

nl()
nonl()
These routines control whether newline is translated into

carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations using **nonl**(), *curses* is able to make better use of the linefeed capability, resulting in faster cursor motion.

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**().

For more information on these options, see "curses/terminfo" in the *System V Programmer's Guide*.

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**() overrides **raw**(). See **getch**() under "Input Routines" 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 **getch**() as they are typed. Echoing by the tty driver is always disabled, but initially **getch**() 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 **getch**() under "Input Routines" for a discussion of how these routines interact with **cbreak**() and **nocbreak**().

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 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) and off (made to work locally), turning on this option will cause the terminal keypad to be turned on when **wgetch**() is called.

meta(win, bf)
If enabled, characters returned by **wgetch**() are transmitted with all 8 bits, instead of with the highest bit stripped. In order for **meta**() to work correctly, the **km** (has_meta_key) capability has to be specified in the terminal's **terminfo**(4) entry.

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. The differences are that in RAW mode, the interrupt, quit, suspend, and flow control characters are passed through uninterpreted, instead of generating a signal. RAW mode also causes 8-bit input and output.

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 **refresh**() 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.

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**().

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.

Soft Label Routines

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, if you want to simulate them, *curses* will take over the bottom line of **stdscr**, reducing the size of **stdscr** and the variable **LINES**. *Curses* standardizes on 8 labels of 8 characters each.

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 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*, with leading and trailing blanks stripped, is returned.

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.

Low-Level *curses* Access

The following routines give low-level access to various *curses* functionality. These routines typically would be used inside of 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*. Like **getyx**(), the variables *y* and *x* do not take an "&" before them. If **leaveok**() is currently **TRUE**, then **-1,-1** will be returned. If lines may have been removed from the top of the screen using **ripoffline**() and the values are to be used beyond just passing them on to **setsyx**(), the value *y* + **stdscr->_yoffset** should be used for those other uses.

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

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 mess up 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**(). 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 time-stamp of the tty is old or the *terminfo*(4) capability **nrrmc** is true.

curs_set(visibility)
The cursor is set to invisible, normal, or very visible for *visibility* equal to **0**, **1** or **2**.

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.

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. 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)
Like **setupterm**() after a memory restore.

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*-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*-like routine **putc**().

vidattr(attrs)
Like **vidputs**(), except that it outputs through *putchar*.

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

The following routines return the value of the capability corresponding to the *terminfo*(4) *capname* passed to them, such as **xenl**.

tigetflag(capname)
The value **-1** is returned if *capname* is not a boolean capability.

tigetnum(capname)
The value **-2** is returned if *capname* is not a numeric capability.

tigetstr(capname)
The value (char *) **-1** is returned if *capname* is not a string capability.

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.

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(codes)
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."

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.

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.

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 **attron**(), **attroff**(), and **attrset**(), or OR'ed with the characters passed to **addch**().

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_CHARTEXT	Bit-mask to extract character [described under **winch**()]
A_ATTRIBUTES	Bit-mask to extract attributes [described under **winch**()]
A_NORMAL	Bit mask to reset all attributes off (for example: **attrset (A_NORMAL)**

FUNCTION-KEYS

The following function keys, defined in **<curses.h>**, might be returned by **getch**() 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 reserved.
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 charcter 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

RETURN VALUES

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 **setscrreg**(), **wsetscrreg**(), **getsyx**(), **getyx**(), **getbegy**(), **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 will be changed 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*. All programs that ran with UNIX System V Release 2 *curses* will run with System V Release 3.0. You may link applications with object files based on the Release 2 *curses/terminfo* with the Release 3.0 *libcurses.a* library. You may link applications with object files based on the Release 3.0 *curses/terminfo* with the Release 2 *libcurses.a* library, so long as the application does not use the new features in the Release 3.0 *curses/terminfo*.

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

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.

SEE ALSO

cc(1), ld(1), ioctl(2), plot(3X), putc(3S), scanf(3S), stdio(3S),

system(3S), vprintf(3S), profile(4), term(4), terminfo(4), varargs(5).
termio(7), tty(7) in the *Administrator's Reference Manual*.
"curses/terminfo" in the *Programmer's Guide*.

NAME

libdev - manipulate Volume Home Blocks (VHB)

SYNOPSIS

```
#include <sys/gdisk.h>

struct vhbd *vhbd;
short sl, *slp;
char *s, *device;
int fd;

int gdnsec(vhbd, sl)
int gdstrk(vhbd, sl)
int gdftrk(vhbd, sl)
int gdnszc(vhbd)
int isdisk(fd)
struct vhbd *readvhb(s, slp)
struct vhbd *sreadvhb(device)
struct vhbd *freadvhb(fd, slp)
char *adevname(fd)
char *bdevname(s)
int dismnt(fd)
char *gdname(s, slp)
char *fgdname(fd, slp)
int gdnlblk(fd)
int writvhb(s, sl, vhbd)
int swritevhb(device, vhbd)
int fwritevhb(fd, sl, vhbd)
```

DESCRIPTION

In each of the above subroutines the arguments denote:

vhbd A pointer to a disk volume home block, as returned by *readvhb*, *sreadvhb*, or *freadvhb*.

sl Slice number on the drive.

slp Pointer to a slice number. This argument is actually used by the subroutine to return a slice number.

s The name of a special file in **/dev/rdsk**. This filename is used to obtain a file descriptor to access a VHB. The name need not be for slice zero of the disk.

device The name of a special file in **/dev/rdsk**. This filename is used to obtain a file descriptor to

access a VHB. The name must be for slice zero of a disk.

fd Open file descriptor for slice zero of a disk.

The subroutines in **/usr/lib/libdev.a** form a device and machine independent interface to the VHB of 6000/50 disks. The function of each subroutine is described below.

Gdnsec Returns the number of sectors in slice *sl* of the VHB indicated by *vhbd*.

Gdstrk Returns the starting track of slice *sl* of the VHB pointed to by *vhbd*.

Gdftrk returns 1 if slice *sl* of the VHB pointed to by *vhbd* extends to the end of the disk.

Gdnszc Returns the number of sectors per cylinder.

Isdisk Returns 1 if the file descriptor *fd* is opened to a *special* disk device.

Readvhb, *Sreadvhb*, and *Freadvhb*
Return a pointer to a VHB for the device described by their arguments.

Adevname
Returns the character device name for the disk drive that the file descriptor *fd* is opened to.

Bdevname
Returns the block device name for the disk drive that the string *s* names. The filename *S* may be either for any slice on either a raw or a block device.

Dismnt Exercises the GDDISMNT ioctl call for the disk drive that the file descriptor *fd* is opened to.

Gdname Returns the file name for the character special slice zero of a disk that the filename *s* name a slice of. The value pointed to by *slp* is set to the slice number of the filename *s*. *Fgdname* performs as does *gdname*, but uses the file descriptor *fd* instead of the filename *s*.

Gdnlblk Returns the number of logical blocks in the slice that the file descriptor *fd* is opened to.

writevhb, *swritevhb*, and *fwritevhb*
Write the volume home block onto the *device*.

FILES

/dev/rdsk/c?d?s?
/dev/dsk/c?d?s?
/usr/lib/libdev.a

SEE ALSO

iv(1) in the *User's Reference Manual*.
disk(7) in the *Administrator's Reference Manual*.

[This page left blank.]

NAME

ocurse - optimized screen functions

SYNOPSIS

#include <ocurse.h>

DESCRIPTION

Ocurse is the old Berkeley curses library that uses *otermcap*(4).

These functions optimally update the screen.

Each *curses* program begins by calling *initscr* and ends by calling *endwin*.

Before a program can change a screen, it must specify the changes. It stores changes in a variable of type **WINDOW** by calling *curses* functions with the variable as argument. Once the variable contains all the changes desired, the program calls *wrefresh* to write the changes to the screen.

Most programs only need a single **WINDOW** variable. *Curses* provides a standard **WINDOW** variable for this case and a group of functions that operate on it. The variable is called *stdscr*; its special functions have the same name as the general functions minus the initial **w**.

FILES

/usr/include/ocurse.h	header file
/usr/lib/libocurse.a	curses library
/usr/lib/libtermcap.a	termcap library, used by curses

SEE ALSO

otermcap(4).

FUNCTIONS

addch(ch)	Add a character to *stdscr*.
addstr(str)	Add a string to *stdscr*.
box(win,vert,hor)	Draw a box around a window.
crmode()	Set cbreak mode.
clear()	Clear *stdscr*.
clearok(scr,boolf)	Set clear flag for *scr*.
clrtobot()	Clear to bottom on *stdscr*.
clrtoeol()	Clear to end of line on *stdscr*.
delch()	Delete a character.

deleteln()	Delete a line.
delwin(win)	Delete *win*.
echo()	Set echo mode.
endwin()	End window modes.
erase()	Erase *stdscr*.
getch()	Get a character through *stdscr*.
getcap(name)	Get terminal capability *name*.
getstr(str)	Get a string through *stdscr*.
gettmode()	Get tty modes.
getyx(win,y,x)	Get (y,x) co-ordinates.
inch()	Get character at current (y,x) co-ordinates.
initscr()	Initialize screens.
insch(c)	Insert a char.
insertln()	Insert a line.
leaveok(win,boolf)	Set leave flag for *win*.
longname(termbuf,name)	Get long name from *termbuf*.
move(y,x)	Move to (y,x) on *stdscr*.
mvcur(lasty,lastx,newy,newx)	Actually move cursor.
newwin(lines,cols,begin_y,begin_x)	Create a new window.
nl()	Set newline mapping.
nocrmode()	Unset cbreak mode.
noecho()	Unset echo mode.
nonl()	Unset newline mapping.
noraw()	Unset raw mode.
overlay(win1,win2)	Overlay win1 on win2.
overwrite(win1,win2)	Overwrite win1 on top of win2.
printw(fmt,arg1,arg2,...)	Printf on *stdscr*.
raw()	Set raw mode.
refresh()	Make current screen look like *stdscr*.
resetty()	Reset tty flags to stored value.
savetty()	Stored current tty flags.
scanw(fmt,arg1,arg2,...)	Scanf through *stdscr*.
scroll(win)	Scroll *win* one line.
scrollok(win,boolf)	Set scroll flag.
setterm(name)	Set term variables for name.
standend()	End standout mode.

standout()	Start standout mode.
subwin(win,lines,cols,begin_y,begin_x)	Create a subwindow.
touchwin(win)	"change" all of *win*.
unctrl(ch)	Printable version of *ch*.
waddch(win,ch)	Add char to *win*.
waddstr(win,str)	Add string to *win*.
wclear(win)	Clear *win*.
wclrtobot(win)	Clear to bottom of *win*.
wclrtoeol(win)	Clear to end of line on *win*.
wdelch(win,c)	Delete character from *win*.
wdeleteln(win)	Delete line from *win*.
werase(win)	Erase *win*.
wgetch(win)	Get a character through *win*.
wgetstr(win,str)	Get a string through *win*.
winch(win)	Get character at current (y,x) in *win*.
winsch(win,c)	Insert character into *win*.
winsertln(win)	Insert line into *win*.
wmove(win,y,x)	Set current (y,x) co-ordinates on *win*.
wprintw(win,fmt,arg1,arg2,...)	Printf on *win*.
wrefresh(win)	Make screen look like *win*.
wscanw(win,fmt,arg1,arg2,...)	Scanf through *win*.
wstandend(win)	End standout mode on *win*.
wstandout(win)	Start standout mode on *win*.

[This page left blank.]

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs - terminal independent operations

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cmstr, destcol, destline)
char *cmstr;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

DESCRIPTION

These functions are now emulated in *curses*(3X) using the *terminfo*(4) databases. Refer to the "Termcap Emulation" section in *curses*(3X).

This manual page is included for historical reason only. New and existing applications should use *curses*(3X) and *terminfo*(4) for terminal independent operations. However, as a conversion aid, these functions extract and use information from terminal descriptions that follow the conventions in *otermcap*(4). The functions only do basic screen manipulation: they find and output specified terminal function strings and interpret the **cm** string.

Tgetent finds and copies a terminal description. *Name* is the name of the description; *bp* points to a buffer to hold the description. *Tgetent* passes *bp* to the other *termcap* functions; the buffer must remain allocated until the program is done with the *termcap* functions.

Tgetent uses the **TERM** and **TERMCAP** environment variables to locate the terminal description.

- If **TERMCAP** isn't set or is empty, *tgetent* searches for *name* in */etc/termcap*.
- If **TERMCAP** contains the full pathname of a file (any string that begins with /), *tgetent* searches for *name* in that file.
- If **TERMCAP** contains any string that does not begin with / and **TERM** is not set or matches *name*, *tgetent* copies the **TERMCAP** string.
- If **TERMCAP** contains any string that does not begin with / and **TERM** does not match *name*, *tgetent* searches for *name* in */etc/termcap.*

Tgetent returns -1 if it couldn't open the terminal capability file, 0 if it couldn't find an entry for *name,* and 1 upon success.

Tgetnum returns the value of the numeric capability whose name is *id*. It returns -1 if the terminal lacks the specified capability or it is not a numeric capability.

Tgetflag returns 1 if the terminal has boolean capability whose name is *id*, 0 if it does not or it is not a boolean capability.

Tgetstr copies and interprets the value of the string capability named by *id*. *Tgetstr* expands instances in the string of \ and ^. It leaves the expanded string in the buffer *indirectly* pointed to by *area* and leaves the buffer's direct pointer pointing to the end of the expanded string; for example,

```
tgetstr("cl", &ptr);
```

where *ptr* is a character pointer, not an array name. *Tgetstr* returns a (direct) pointer to the beginning of the string.

Tgoto interprets the % escapes in a **cm** string. It returns *cmstr* with the % sequences changed to the position indicated by *destcol* and *destline*. This function must have the external variables *BC* and *UP* set to the values of the **bc** and **up**

capabilities; if the terminal lacks the capability, set the external variable to null. If *tgoto* can't interpret all the % sequences in **cm**, it returns "OOPS"

Tgoto avoids producing characters that might be misinterpreted by the terminal interface. If expanding a % sequence would produce a control-d or null, the function will, if possible, send the cursor to the next line or column and use *BC* or *UP* to move to the correct location. Note that *tgoto* does not avoid producing tabs; a program must turn off the **TAB3** feature of the terminal interface [*termio*(7)]. This is a good idea anyway: some terminals use the tab character as a non-destructive space.

Tputs directs the output of a string returned by *tgetstr* or *tgoto*. This function must have the external variable *PC* set to the value of the **pc** capability; if the terminal lacks the capability, set the external variable to null. *Tputs* interprets any delay at the beginning of the string. *Cp* is the string to be output; *affcnt* is the number of lines affected by the action (1 if "number of lines affected" doesn't mean anything); and *outc* points to a function that takes a single **char** argument and outputs it, such as *putchar*.

FILES

/usr/lib/libtermcap.a	library
/etc/termcap	data base

SEE ALSO

curses(3X), terminfo(4).

[This page left blank.]

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

[This page left blank.]