Character User Interface Reference Manual

FMLI Commands(1F)

coproc(1F)
Form and Menu Language Interpreter

NAME

`coproc:`
`cocreate, cosend, cocheck, coreceive, codestroy -` communicate with a process

SYNOPSIS

`cocreate` [`−r` *rpath*] [`−w` *wpath*] [`−i` *id*] [`−R` *refname*] [`−s` *send_string*] [`−e` *expect_string*] *command*

`cosend` [`−n`] *proc_id string*

`cocheck` *proc_id*

`coreceive` *proc_id*

`codestroy` [`−R` *refname*] *proc_id* [*string*]

DESCRIPTION

These co-processing functions provide a flexible means of interaction between FMLI and an independent process. They enable FMLI to be responsive to asynchronous activity.

The `cocreate` function starts *command* as a co-process and initializes communications by setting up pipes between FMLI and the standard input and standard output of *command*. The argument *command* must be an executable and its arguments (if any). This means that *command* expects strings on its input (supplied by `cosend`) and sends information on its output that can be handled in various ways by FMLI. The following options can be used with `cocreate`.

`−r` *rpath*
: If `−r` is specified, *rpath* is the pathname from which FMLI reads information. This option is usually used to set up communication with processes that naturally write to a certain path. If `−r` is not specified, `cocreate` chooses a unique path in `/var/tmp`.

`−w` *wpath*
: If `−w` is specified, *wpath* is the pathname to which `cosend` writes information. This option is usually used so that one process can talk to many different FMLI processes through the same pipe. If `−w` is not specified, `cocreate` chooses a unique path in `/var/tmp`.

`−i` *id*
: If `−i` is specified, *id* is an alternative name for the co-process initialized by this `cocreate`. If `−i` is not specified, *id* defaults to *command*. The argument *id* can later be used with the other co-processing functions rather than *command*. This option facilitates the creation of two or more co-processes generated from the same *command*. (For example, `cocreate -i ID1 program args` and `cocreate -i ID2 program different_args`.)

`−R` *refname*
: If `−R` is specified, *refname* is a local name for the co-process. The `cocreate` function can be issued more than once; therefore, a *refname* is useful when the same co-process is referenced a second or subsequent time. If a co-process already exists when the `−R` option is specified, a new one will not be created. The same pipes are shared. *refname* can be used as an argument to the `−R` option to `codestroy` when you want to end a particular connection to a co-process and leave other connections undisturbed. (The co-process is only killed after `codestroy -R` has been called as many times as `cocreate -R` was called.)

−s *send_string*

The −s option specifies *send_string* as a string that will be appended to all output sent to the co-process using cosend. This option allows a co-process to know when input from FMLI has completed. The default *send_string* is a newline if −s is not specified.

−e *expect_string*

The −e option specifies *expect_string* as a string that identifies the end of all output returned by the co-process. *expect_string* need only be the initial part of a line. There must be a newline at the end of the co-process output. This option allows FMLI to know when output from the co-process has completed. The default *expect_string* is a newline if −e is not specified.

The cosend function sends *string* to the co-process identified by *proc_id* through the pipe set up by cocreate (optionally *wpath*), where *proc_id* can be either the *command* or *id* specified in cocreate. By default, cosend blocks, waiting for a response from the co-process. Also by default, FMLI does not send a *send_string* and does not expect an *expect_string* (except a newline). That is, it reads only one line of output from the co-process. If −e *expect_string* was not defined when the pipe was created, then the output of the co-process is any single string followed by a newline: any other lines of output remain on the pipe. If the −e option was specified when the pipe was created, cosend reads lines from the pipe until it reads a line starting with *expect_string*. All lines except the line starting with *expect_string* become the output of cosend. The following option can be used with cosend:

−n If the −n option is specified, cosend will not wait for a response from the co-process. It simply returns, providing no output. If the −n option is not used, a co-process that does not answer will cause FMLI to permanently hang, waiting for input from the co-process.

The cocheck function determines if input is available from the process identified by *proc_id*, where *proc_id* can be either the *command* or *id* specified in cocreate. It returns a Boolean value, which makes cocheck useful in if statements and in other backquoted expressions in Boolean descriptors. cocheck receives no input from the co-process; it simply indicates if input is available from the co-process. You must use coreceive to actually accept the input. The cocheck function can be called from a reread descriptor to force a frame to update when new data is available. This is useful when the default value of a field in a form includes coreceive.

The coreceive function is used to read input from the co-process identified by *proc_id*, where *proc_id* can be either the *command* or *id* specified in cocreate. It should only be used when it has been determined, using cocheck, that input is actually available. If the −e option was used when the co-process was created, coreceive continues to return lines of input until *expect_string* is read. At this point, coreceive terminates. The output of coreceive is all the lines that were read excluding the line starting with *expect_string*. If the −e option was not used in the cocreate, each invocation of coreceive returns exactly one line from the co-process. If no input is available when coreceive is invoked, it terminates without producing output.

The codestroy function terminates the read/write pipes to *proc-id*, where *proc_id* can be either the *command* or *id* specified in cocreate. It generates a SIGPIPE signal to the (child) co-process. This kills the co-process, unless the co-process ignores the SIGPIPE signal. If the co-process ignores the SIGPIPE, it will not die, even after the FMLI process terminates (the parent process id of the co-process is 1).

The optional argument *string* is sent to the co-process before the co-process dies. If *string* is not supplied, a NULL string is passed, followed by the normal *send_string* (newline by default). That is, codestroy will call cosend *proc_id string*. This implies that codestroy writes any output generated by the co-process to stdout. For example, if an interactive co-process is written to expect a quit string when the communication is over, the close descriptor could be defined like this:

```
close=`codestroy ID 'quit' | message`
```

and any output generated by the co-process when the string quit is sent to it through codestroy (using cosend) would be redirected to the message line.

The codestroy function should usually be given the −R option because you may have more than one process with the same name. You do not want to kill the wrong process. codestroy keeps track of the number of *refnames* you have assigned to a process with cocreate, and when the last instance is killed, it kills the process (*id*) for you. codestroy is typically called as part of a close descriptor because close is

evaluated when a frame is closed. This is important because the co-process will continue to run if `codestroy` is not issued.

When writing programs to use as co-processes, the following tips may be useful. If the co-process program is written in C language, be sure to flush output after writing to the pipe. (Currently, `awk(1)` and `sed(1)` cannot be used in a co-process program because they do not flush after lines of output.) Shell scripts are well-mannered, but slow. C language is recommended. If possible, use the default *send_string*, *rpath* and *wpath*. In most cases, *expect_string* will have to be specified. This, of course, depends on the co-process.

In the case where asynchronous communication from a co-process is desired, a co-process program should use `vsig` to force strings into the pipe and then signal FMLI that output from the co-process is available. This causes the `reread` descriptor of all frames to be evaluated immediately.

EXAMPLES

```
    .
    .
    .
init=`cocreate -i BIGPROCESS initialize`
close=`codestroy BIGPROCESS`
    .
    .
    .
reread=`cocheck BIGPROCESS`
name=`cosend -n BIGPROCESS field1`
    .
    .
    .
name="Receive field"
inactive=TRUE
value=`coreceive BIGPROCESS`
```

NOTES

If `cosend` is used without the `-n` option, a co-process that does not answer causes FMLI to permanently hang.

Avoid using non-alphabetic characters in input and output strings to a co-process because they may not get transferred correctly.

SEE ALSO

`awk(1)`, `cat(1)`, `sed(1)`, `vsig(1F)`.

Character User Interface Reference Manual

echo(1F)
Form and Menu Language Interpreter

NAME

`echo -` put string on virtual output

SYNOPSIS

`echo` [*string ...*]

DESCRIPTION

The `echo` function directs each string it is passed to `stdout`. If no argument is given, `echo` looks to `stdin` for input. It is often used in conditional execution or for passing a string to another command.

EXAMPLES

Set the `done` descriptor to `help` if a test fails like this:

```
done=`if [ -s $F1 ];
 then echo close;
 else echo help;
 fi`
```

SEE ALSO

`echo`(1) .

Character User Interface Reference Manual

fmlcut(1F)
Form and Menu Language Interpreter

NAME

`fmlcut -` cut out selected fields of each line of a file

SYNOPSIS

`fmlcut −c`*list* [*file* ...]

`fmlcut −f`*list* [`−d`*char*] [`−s`] [*file* ...]

DESCRIPTION

The `fmlcut` function cuts out columns from a table or fields from each line in *file*. In database parlance, `fmlcut` implements the projection of a relation. `fmlcut` can be used as a filter. If *file* is not specified or is `−` then the standard input is read. *list* specifies the fields to be selected. Fields can be fixed length (character positions) or variable length (separated by a field delimiter character), depending on whether `−c` or `−f` is specified.

Either the `−c` or the `−f` option must be specified.

These options are available with `fmlcut`:

list A comma-separated list of integer field numbers in increasing order. Use of the `−` to indicate ranges is optional. For example: `1,4,7` or `1−3,8` or `−5,10` (short for `1−5,10`) or `3−` (short for third through last field).

`−c`*list*

If `−c` is specified, *list* specifies character positions (for example, `−c1−72` passes the first 72 characters of each line). No space intervenes between `−c` and *list*.

`−f`*list*

If `−f` is specified, *list* is a list of fields assumed to be separated in the file by the default delimiter character, TAB, or by *char* if the `−d` option is specified. For example, `−f1,7` copies the first and seventh field only. Lines with no delimiter characters are passed through intact (useful for table subheadings), unless `−s` is specified. No space intervenes between `−f` and *list*. The following options can be used if you have specified `−f`.

`−d`*char*

If `−d` is specified, *char* is the field delimiter. Space or other characters with special meaning to FMLI must be quoted. No space intervenes between `−d` and *char*. The default field delimiter is TAB.

`−s` Suppresses lines with no delimiter characters. If `−s` is not specified, lines with no delimiters are passed through untouched.

EXAMPLES

`fmlcut −d: −f1,5 /etc/passwd`
Gets login IDs and names

`` `who am i | fmlcut −f1 −d" "` ``
Gets the current login name

DIAGNOSTICS

`fmlcut` returns the following exit values:

`0` The selected field is successfully cut out

`2` Syntax errors

The following error messages may be displayed on the FMLI message line:

`ERROR: line too long`
A line has more than 1023 characters or fields or there is no new-line character.

`ERROR: bad list for c/f option`
Missing `-c` or `-f` option or incorrectly specified *list*. No error occurs if a line has fewer fields than the *list* calls for.

`ERROR: no fields`
The *list* is empty.

`ERROR: no delimiter`
Missing *char* on `-d` option.

NOTES

`fmlcut` cannot correctly process lines longer than 1023 characters or lines with no newline character.

SEE ALSO

`fmlgrep`(1F) .

Character User Interface Reference Manual

fmlexpr(1F)
Form and Menu Language Interpreter

NAME

`fmlexpr -` evaluate arguments as an expression

SYNOPSIS

`fmlexpr` *arguments*

DESCRIPTION

The `fmlexpr` function evaluates its arguments as an expression. After evaluation, the result is written on the standard output. Terms of the expression must be separated by blanks. Characters special to FMLI must be escaped. A `0` is returned to indicate a zero value rather than the null string. Strings containing blanks or other special characters should be quoted. Integer-valued arguments may be preceded by a unary minus sign. Internally, integers are treated as 32-bit, 2s complement numbers.

The operators and keywords are listed below. Characters that need to be escaped are preceded by `\`. The list is in order of increasing precedence with equal precedence operators grouped within { } symbols.

expr `\|` *expr*
: Returns the first *expr* if it is neither null nor `0`; otherwise returns the second *expr*.

expr `\&` *expr*
: Returns the first *expr* if neither *expr* is null or `0`; otherwise returns `0`.

expr { `=`, `\>`, `\>=`, `\<`, `\<=`, `!=` } *expr*
: Returns the result of an integer comparison if both arguments are integers; otherwise returns the result of a lexical comparison.

expr { `+`, `-` } *expr*
: Addition or subtraction of integer-valued arguments.

expr { `*`, `/`, `%` } *expr*
: Multiplication, division, or remainder of the integer-valued arguments.

expr `:` *expr*
: The matching operator `:` compares the first argument with the second argument which must be a regular expression. Regular expression syntax is the same as that of `ed(1)`, except that all patterns are "anchored" (for example, begin with `^`) therefore, `^` is not a special character in this context. Normally the matching operator returns the number of bytes matched (`0` on failure). The `\(...\)` pattern symbols can be used to return a portion of the first argument.

EXAMPLES

1. Add 1 to the variable `a`:

   ```
   `fmlexpr $a + 1 | set -l a`
   ```

2. For `$a` equal to either `/usr/abc/file` or just `file`:

   ```
   fmlexpr  $a  :  '.*/\(.*\)'  \|  $a
   ```

   The example returns the last segment of a path name (for example, *file*). Watch out for `/` alone as an argument: `fmlexpr` will take it as the division operator.

3. A better representation of example 2.

   ```
   fmlexpr  //$a  :  '.*/\(.*\)'
   ```

   The addition of the `//` characters eliminates any ambiguity about the division operator because it makes it impossible for the left-hand expression to be interpreted as the division operator. It simplifies the whole expression.

4. Return the number of characters in `$VAR`.

```
fmlexpr $VAR : .*
```

DIAGNOSTICS

As a side effect of expression evaluation, `fmlexpr` returns the following exit values:

`0` If the expression is neither null nor `0` (for example, TRUE)

`1` If the expression is null or `0` (for example, FALSE)

`2` For invalid expressions (for example, FALSE).

`syntax error`
For operator/operand errors.

`non-numeric argument`
If arithmetic is attempted on such a string.

An error message is printed at the current cursor position for syntax errors and non-numeric arguments. Use `refresh` to redraw the screen.

NOTES

After argument processing by FMLI, `fmlexpr` cannot tell the difference between an operator and an operand except by the value. If `$a` is an `=`, the command:

```
fmlexpr  $a  =  '='
```

looks like this:

```
fmlexpr  =  =  =
```

The arguments are passed to `fmlexpr` and they will all be taken as the `=` operator. The following works, and returns TRUE:

```
fmlexpr  X$a  =  X=
```

SEE ALSO

`ed(1)`, `expr(1)`, `sh(1)`, `set(1F)`.

Character User Interface Reference Manual

fmlgrep(1F)
Form and Menu Language Interpreter

NAME

`fmlgrep -` search a file for a pattern

SYNOPSIS

`fmlgrep` [*options*] *limited_regular_expression* [*file ...*]

DESCRIPTION

`fmlgrep` searches *file* for a pattern and prints all lines that contain that pattern. The `fmlgrep` function uses limited regular expressions (expressions that have string values that use a subset of the possible alphanumeric and special characters) like those used with `ed(1)` to match the patterns. It uses a compact non-deterministic algorithm.

Be careful when using FMLI special characters (for example, `$`, `` ` ``, `´`, `"`) in *limited_regular_expression*. It is safest to enclose the entire *limited_regular_expression* in single quotes `'...'`.

If *file* is not specified `fmlgrep` assumes standard input. Normally each line matched is copied to standard output. The file name is printed before each line matched if there is more than one input file.

OPTIONS

`-b` Precede each line by the block number on which it was found. This can be useful in locating block numbers by context (first block is 0).

`-c` Print only a count of the lines that contain the pattern.

`-i` Ignore upper/lower case distinction during comparisons.

`-l` Print only the names of files with matching lines, separated by new-lines. Does not repeat the names of files when the pattern is found more than once.

`-n` Precede each line by its line number in the file (first line is 1).

`-s` Suppress error messages about nonexistent or unreadable files.

`-v` Print all lines except those that contain the pattern.

DIAGNOSTICS

`fmlgrep` returns the following exit values:

`0` Pattern is found (for example, TRUE)

`1` Pattern is not found (for example, FALSE)

`2` An invalid expression was used or *file* is inaccessible

NOTES

Lines are limited to `BUFSIZ` characters. Longer lines are truncated. `BUFSIZ` is defined in `/usr/include/stdio.h`.

If there is a line with embedded nulls, `fmlgrep` will only match up to the first null. If it matches, it will print the entire line.

SEE ALSO

`ed(1)`, `egrep(1)`, `fgrep(1)`, `fmlcut(1F)`.

Character User Interface Reference Manual

getfrm(1F)
Form and Menu Language Interpreter

NAME

`getfrm -` returns the current frameID number

SYNOPSIS

```
getfrm
```

DESCRIPTION

`getfrm` returns the current frameID number. The frameID number is a number assigned to the frame by FMLI and displayed flush left in the frame's title bar. If a frame is closed its frameID number may be reused when a new frame is opened. `getfrm` takes no arguments.

EXAMPLES

If a menu whose frameID is `3` defines an item to have this `action` descriptor:

```
action=open text stdtext `getfrm`
```

then the text frame defined in the definition file `stdtext` is passed the argument `3` when it is opened.

NOTES

It is not a good idea to use `getfrm` in a backquoted expression coded on a line by itself. Stand-alone backquoted expressions are evaluated before any descriptors are parsed, thus the frame is not yet fully current, and may not have been assigned a frameID number.

getitems(1F)
Form and Menu Language Interpreter

NAME

`getitems -` returns a list of currently marked menu items

SYNOPSIS

`getitems` [*delimiter_string*]

DESCRIPTION

The `getitems` function returns the value of `lininfo` if defined, else it returns the value of the `name` descriptor for all currently marked menu items. Each value in the list is delimited by *delimiter_string*. The default value of *delimiter_string* is newline.

EXAMPLES

The `done` descriptor in the following menu definition file executes `getitems` when the user presses ENTER (the menu is `multiselect`):

```
Menu="Example"
multiselect=TRUE
done=`getitems ":" | message`
name="Item 1"
action=`message "You selected item 1"`
name="Item 2"
lininfo="This is item 2"
action=`message "You selected item 2"`
name="Item 3"
action=`message "You selected item 3"`
```

If a user marked all three items in this menu, pressing ENTER would cause the following string to be displayed on the message line:

```
Item 1:This is item 2:Item 3
```

`lininfo` is defined for the second menu item therefore, its value is displayed instead of the value of the `name` descriptor.

Character User Interface Reference Manual

indicator(1F)
Form and Menu Language Interpreter

NAME

`indicator -` display application specific alarms and/or the "working"indicator

SYNOPSIS

`indicator` [–b [*n*]] [–c *column*] [–l *length*] [–o] [–w] [*string ...*]

DESCRIPTION

The `indicator` function displays application specific alarms or the "working" indicator, or both on the FMLI banner line. By default, `indicator` is set to `Working`. The argument *string* is a string to be displayed on the banner line and should always be the last argument given. *string* is not automatically cleared from the banner line.

OPTIONS

–b *n* The –b option rings the terminal bell *n* times, where *n* is an integer from 1 to 10. The default value is 1. If the terminal has no bell, the screen is flashed, if possible.

–c *column*
The –c option defines the column of the banner line at which to start the indicator string. The argument *column* must be an integer from `0` to `DISPLAYW-1`. If the –c option is not used, *column* defaults to `0`.

–l *length*
The –l option defines the maximum length of the string displayed. If *string* is longer than *length* characters, it is truncated. The argument *length* must be an integer from `1` to `DISPLAYW`. If the –l option is not used, *length* defaults to `DISPLAYW`. If *string* doesn't fit, it is truncated.

–o The –o option causes `indicator` to duplicate its output to `stdout`.

–w The –w option turns on the "working" indicator.

EXAMPLES

When the value entered in a form field is invalid, the following use of `indicator` rings the bell three times and displays the word `WRONG` starting at column 1 of the banner line.

```
invalidmsg=`indicator –b 3 –c 1 "WRONG"`
```

To clear the indicator after telling the user the entry is wrong:

```
invalidmsg=`indicator –b 9 –c 1 "WRONG"; sleep(3);
     indicator –c 1 "     "`
```

In this example the value of `invalidmsg` (in this case the default value `Input is not valid`), still appears on the FMLI message line.

Character User Interface Reference Manual

message(1F)
Form and Menu Language Interpreter

NAME

`message -` puts its arguments on FMLI message line

SYNOPSIS

`message` [`–t`] [`–b` [*num*]] [`–o`] [`–w`] [*string*]

`message` [`–f`] [`–b` [*num*]] [`–o`] [`–w`] [*string*]

`message` [`–p`] [`–b` [*num*]] [`–o`] [`–w`] [*string*]

DESCRIPTION

The `message` command puts *string* out on the FMLI message line. If there is no string, the `stdin` input to `message` will be used. The output of `message` has a duration (length of time it remains on the message line). The default duration is transient. It or one of two other durations can be requested with the following mutually-exclusive options:

`–t` Explicitly defines a message to have transient duration. Transient messages remain on the message line only until the user presses another key or a `CHECKWORLD` occurs. The descriptors `itemmsg`, `fieldmsg`, `invalidmsg`, `choicemsg`, the default-if-not-defined value of `oninterrupt`, and FMLI generated error messages (for example, from syntax errors) also output transient duration messages. Transient messages take precedence over both frame messages and permanent messages.

`–f` Defines a message to have frame duration. Frame messages remain on the message line as long as the frame in which they are defined is current. The descriptor `framemsg` also outputs a frame duration message. Frame messages take precedence over permanent messages.

`–p` Defines a message to have permanent duration. Permanent messages remain on the message line for the length of the FMLI session unless explicitly replaced by another permanent message or temporarily superseded by a transient message or frame message. A permanent message is not affected by navigating away from, or by closing, the frame which generated the permanent message. The descriptor `permanentmsg` also outputs a permanent duration message.

Messages displayed with `message –p` replace (change the value of) any message currently displayed or stored through use of the `permanentmsg` descriptor. Likewise, `message –f` replaces any message currently displayed or stored through use of the `framemsg` descriptor. If more than one message in a frame definition file is specified with the `–p` option, the last one specified is the permanent duration message.

The *string* argument should always be the last argument. The following options are available with `message`:

`–b` [*num*]
Rings the terminal bell *num* times, where *num* is an integer from 1 to 10. The default value is 1. If the terminal has no bell, the screen flashes *num* times, if possible.

`–o` Forces `message` to duplicate its message to `stdout`.

`–w` Turns on the working indicator.

EXAMPLES

When a value entered in a field is invalid, ring the bell 3 times and then display `Invalid Entry: Try again!` on the message line like this:

```
invalidmsg=`message –b 3 "Invalid Entry: Try again!"`
```

Display a message that tells the user what is being done:

```
done=`message EDITOR has been set in your environment` close
```

Display a message on the message line and `stdout` for each field in a form (a pseudo-field duration message).

```
fieldmsg="`message -o -f "Enter a filename."`"
```

Display a blank transient message (effect is to remove a permanent or frame duration message).

```
done=`message  ""` nop
```

NOTES

If `message` is coded more than once on a single line, it may appear that only the right-most instance is interpreted and displayed. Use `sleep(1)` between uses of `message` to display multiple messages.

`message -f` should not be used in a stand-alone backquoted expression or with the `init` descriptor because the frame is not yet current when these are evaluated.

In cases where `` `message -f "*string*"` `` is part of a stand-alone backquoted expression, the context for evaluation of the expression is the previously current frame. The previously current frame can be the frame that issued the `open` command for the frame containing the backquoted expression, or it can be a frame given as an argument when `fmli` was invoked. That is, the previously current frame is the one whose frame message will be modified.

Permanent duration messages are displayed when the user navigates to the command line.

SEE ALSO

`sleep(1)` .

Character User Interface Reference Manual

pathconv(1F)
Form and Menu Language Interpreter

NAME

`pathconv` - search FMLI criteria for filename

SYNOPSIS

`pathconv` [`–f`] [`–v` *alias*]

`pathconv` [`–t`] [`–l`] [`–n`*num*] [`–v` *string*]

DESCRIPTION

The `pathconv` function converts an alias to its pathname. By default, it takes the alias as a string from `stdin`.

OPTIONS

`–f` If `–f` is specified, the full path is returned. This is the default.

`–t` If `–t` is specified, `pathconv` truncates a pathname specified in *string* in a format suitable for display as a frame title. This format is a shortened version of the full pathname. It is created by deleting components of the path from the middle of the string until it is under `DISPLAYW - 6` characters in length and then inserting ellipses (`. . .`) between the remaining pieces. Ellipses are also used to show truncation at the ends of the strings, if necessary, unless the `–l` option is given.

`–l` If `–l` is specified < and > is used instead of ellipses (`. . .`) to indicate truncation at the ends of the string generated by the `–t` option. Using `–l` allows display of the longest possible string while still notifying users it has been truncated.

`–n`*num*
If `–n` is specified, *num* is the maximum length of the string (in characters) generated by the `–t` option. The argument *num* can be any integer from 1 to 255.

`–v` *arg*
If the `–v` option is used, then *alias* or *string* can be specified when `pathconv` is called. The argument *alias* must be an alias defined in the *alias_file* named when `fmli` was invoked. The argument *string* can only be used with the `–t` option and must be a pathname.

EXAMPLES

Here is a menu descriptor that uses `pathconv` to construct the menu title. It searches for `MYPATH` in the *alias_file* named when `fmli` is invoked:

```
menu=`pathconv –v MYPATH/ls`
    .
    .
    .
```

There is a line in *alias_file* that defines `MYPATH`. For example, `MYPATH=$HOME/bin:/usr/bin`.

Here is a menu descriptor that takes *alias* from `stdin`.

```
menu=`echo MYPATH/ls | pathconv`
    .
    .
    .
```

SEE ALSO

`fmli`(1).

Character User Interface Reference Manual

readfile(1F)
Form and Menu Language Interpreter

NAME

`readfile, longline -` reads file, gets longest line

SYNOPSIS

`readfile` *file*

`longline` [*file*]

DESCRIPTION

The `readfile` function reads *file* and copies it to `stdout`. No translation of NEWLINE is done. It keeps track of the longest line it reads and if there is a subsequent call to `longline`, the length of that line (including the NEWLINE character) is returned.

The `longline` function returns the length (including the NEWLINE character) of the longest line in *file*. If *file* is not specified, it uses the file named in the last call to `readfile`.

EXAMPLES

Here is a typical use of `readfile` and `longline` in a text frame definition file:

```
  .
  .
  .
text="`readfile myfile`"
columns=`longline`
  .
  .
  .
```

DIAGNOSTICS

If *file* does not exist, `readfile` returns FALSE (for example, the expression has an error return).

`longline` returns 0 if a `readfile` has not previously been issued.

NOTES

More than one descriptor can call `readfile` in the same frame definition file. In text frames, if one of those calls is made from the `text` descriptor, then a subsequent use of `longline` will always get the longest line of the file read by the `readfile` associated with the `text` descriptor, even if it was not the most recent use of `readfile`.

SEE ALSO

`cat(1)`.

Character User Interface Reference Manual

regex(1F)
Form and Menu Language Interpreter

NAME

`regex -` match patterns against a string

SYNOPSIS

`regex` [`–e`] [`–v "`*string*`"`] [*pattern template*] ... *pattern* [*template*]

DESCRIPTION

The `regex` command takes a string from `stdin` and a list of *pattern/template* pairs and runs `regex(3G)` to compare the string against each *pattern* until there is a match. When a match occurs, `regex` writes the corresponding *template* to `stdout` and returns TRUE. The last (or only) *pattern* does not need a template. If that is the pattern that matches the string, the function simply returns TRUE. If no match is found, `regex` returns FALSE.

OPTIONS

`–e` `regex` evaluates the corresponding template and writes the result to `stdout`.

`–v "`*string*`"`
If `–v` is specified, *string* is used instead of `stdin` to match against patterns.

The argument *pattern* is a regular expression of the form described in `regex(3G)` . In most cases *pattern* should be enclosed in single quotes to turn off special meanings of characters. Only the final *pattern* in the list may lack a *template*.

The argument *template* may contain the strings `$m0` through `$m9`, which is expanded to the part of *pattern* enclosed in `(...)$0` through `(...)$9` constructs. If you use this feature, you must be sure to enclose *template* in single quotes so that FMLI doesn't expand `$m0` through `$m9` at parse time. This feature gives `regex` much of the power of `cut(1)` , `paste(1)` , and `grep(1)` , and some of the capabilities of `sed(1)` . If there is no *template*, the default is "`$m0$m1$m2$m3$m4$m5$m6$m7$m8$m9`".

EXAMPLES

The following example shows you how to cut the 4th through 8th letters out of a string. This example will output `strin` and return TRUE:

```
`regex -v "my string is nice" '^.{3}(.{5})$0' '$m0'`
```

This example shows you how to to validate input to field 5 as an integer in a form.

```
valid=`regex -v "$F5" '^[0-9]+$'`
```

This example shows you how to to translate an environment variable which contains one of the numbers `1`, `2`, `3`, `4`, `5` to the letters `a`, `b`, `c`, `d`, `e` in a form.

```
value=`regex -v "$VAR1" 1 a 2 b 3 c 4 d 5 e '.*' 'Error'`
```

Use the pattern `'.*'` to mean anything else.

In the following example, all three lines constitute a single backquoted expression. This expression, by itself, could be put in a menu definition file. Backquoted expressions are expanded as they are parsed. Output from a backquoted expression becomes part of the definition file being parsed. The expression in this example would read `/etc/passwd` and make a dynamic menu of all the login IDs on the system.

```
`cat /etc/passwd | regex '^([^:]*)$0.*$' '
name=$m0
action=`message "$m0 is a user"`'`
```

DIAGNOSTICS

If none of the patterns match, `regex` returns FALSE, otherwise TRUE.

NOTES

Patterns and templates must often be enclosed in single quotes to turn off the special meanings of characters. Especially if you use the `$m0` through `$m9` variables in the template, since FMLI expands the variables (usually to "") before `regex` even sees them.

Single characters in character classes (inside `[]`) must be listed before character ranges, otherwise they will not be recognized. For example, `[a-zA-Z_/]` will not find underscores (_) or slashes (/), but `[_/a-zA-Z]` will.

The regular expressions accepted by `regcmp` differ slightly from other utilities (for example, `sed`, `grep`, `awk`, or `ed`).

`regex` with the `–e` option forces subsequent commands to be ignored. In other words if a backquoted statement appears like this:

```
`regex -e ...; command1; command2`
```

command1 and *command2* would never be executed. However, dividing the expression into two would yield the desired result.

```
`regex -e ...``command1; command2`
```

SEE ALSO

`awk`(1), `cut`(1), `grep`(1), `paste`(1), `sed`(1), `regcmp`(3G), `regex`(3G).

Character User Interface Reference Manual

reinit(1F)
Form and Menu Language Interpreter

NAME

`reinit -` runs an initialization file

SYNOPSIS

`reinit` *file*

DESCRIPTION

The `reinit` command is used to change the values of descriptors defined in the initialization file that was named when `fmli` was invoked and/or define additional descriptors. FMLI will parse and evaluate the descriptors in *file*, and then continue running the current application. The argument *file* must be the name of a valid FMLI initialization file.

The `reinit` command does not re-display the introductory frame or change the layout of screen labels for function keys.

Character User Interface Reference Manual

reset(1F)
Form and Menu Language Interpreter

NAME

`reset -` reset the current form field to its default values

SYNOPSIS

```
reset
```

DESCRIPTION

The `reset` function changes the entry in a field of a form to its default value; that is, the value displayed when the form was opened.

Character User Interface Reference Manual

run(1F)
Form and Menu Language Interpreter

NAME

`run -` run an executable

SYNOPSIS

`run [ –s ] [ –e ] [ –n ] [ –t` *string* `]` *program*

DESCRIPTION

The `run` function runs *program* using the PATH variable to find it. By default, the user is prompted when *program* has completed before being returned to FMLI. The prompt looks like this: `Press ENTER to continue:`. The argument *program* is a UNIX system executable followed by its options (if any).

OPTIONS

`–e` If `–e` is specified the user will be prompted before returning to FMLI only if there is an error condition

`–n` If `–n` is specified the user will never be prompted before returning to FMLI (useful for programs like `vi`, in which the user must perform a specific action to exit).

`–s` The `–s` option means silent, implying that the screen will not have to be repainted when *program* has completed. The `–s` option should only be used when *program* does not write to the terminal. In addition, when `–s` is used, *program* cannot be interrupted even if it recognizes interrupts.

`–t` *string*
If `–t` is specified, *string* is the name this process will have in the pop-up menu generated by the `frm-list` command. This feature requires the executable `facesuspend` to suspend the UNIX system process and return to the FMLI application.

EXAMPLES

Here is a menu that uses `run`:

```
menu="Edit special System files"
name="Password file"
action=`run –e vi /etc/passwd`
name="Group file"
action=`run –e vi /etc/group`
name="My .profile"
action=`run –n vi $HOME/.profile`
```

Character User Interface Reference Manual

set(1F)
Form and Menu Language Interpreter

NAME

`set, unset -` set and unset local or global environment variables

SYNOPSIS

`set` [`-l` *variable*[=*value*]] ...

`set` [`-e` *variable*[=*value*]] ...

`set` [`-f`*file variable*[=*value*]] ...

`unset -l` *variable* ...

`unset -f`*file variable* ...

DESCRIPTION

The `set` command sets *variable* in the environment or adds *variable=value* to *file*. If *variable* is not equated it to a value, `set` expects the value to be on `stdin`. The `unset` command removes *variable*. The FMLI predefined read-only variables (such as `ARG1`), may not be set or unset.

FMLI inherits the Reliant UNIX environment when invoked:

`-l` Sets or unsets the specified variable in the local environment. Variables set with `-l` will not be inherited by processes invoked from FMLI.

`-e` Sets the specified variable in the Reliant UNIX environment. Variables set with `-e` will be inherited by any processes started from FMLI. These variables cannot be `unset`.

`-f`*file*
Sets or unsets the specified variable in the global environment. The argument *file* is the name or pathname of a file containing lines of the form *variable=value*. *file* will be created if it does not already exist. No space intervenes between `-f` and *file*.

At least one of the above options must be used for each variable being set or unset. If you set a variable with the `-f`*filename* option, you must thereafter include *filename* in references to that variable. For example, `${`*(file)VARIABLE*`}`.

EXAMPLES

Storing a selection made in a menu is done like this:

```
name=Selection 2
action=`set -l SELECTION=2`close
```

NOTES

Variables set to be available to the Reliant UNIX environment (those set using the `-e` option) can only be set for the current FMLI process and the processes it calls.

When using the `-f` option (unless *file* is unique to the process) other users of FMLI on the same machine are able to expand these variables, depending on the read/write permissions on *file*.

A variable set in one frame may be referenced or unset in any other frame. This includes local variables.

SEE ALSO

`env`(1), `sh`(1).

Character User Interface Reference Manual

setcolor(1F)
Form and Menu Language Interpreter

NAME

`setcolor -` redefine or create a color

SYNOPSIS

`setcolor` *color red_level green_level blue_level*

DESCRIPTION

The `setcolor` command takes four arguments: *color* which must be a string naming the color; and the arguments *red_level*, *green_level*, and *blue_level* which must be integer values defining, respectively, the intensity of the red, green, and blue components of *color*. Intensities must be in the range of 0 to 1000. If you are redefining an existing color, you must use its current name (default color names are: `black`, `blue`, `green`, `cyan`, `red`, `magenta`, `yellow`, and `white`). `setcolor` returns the name string of the color.

EXAMPLES

```
`setcolor blue 100 24 300`
```

Character User Interface Reference Manual

shell(1F)
Form and Menu Language Interpreter

NAME

`shell -` run a command using shell

SYNOPSIS

`shell` *command* [*command*] ...

DESCRIPTION

The `shell` function concatenates its arguments, separating each by a space, and passes this string to the Reliant UNIX system shell (`$SHELL` if set, otherwise `/usr/bin/sh`).

EXAMPLES

Since the Form and Menu Language does not directly support background processing, the `shell` function can be used instead.

```
`shell "build prog > /dev/null &"`
```

If you want the user to continue to be able to interact with the application while the background job is running, the output of an executable run by `shell` in the background must be redirected: to a file if you want to save the output or to `/dev/null` if you don't want to save it (or if there is no output); otherwise your application may appear to be hung until the background job finishes processing.

`shell` can also be used to execute a command that has the same name as an FMLI built-in function.

NOTES

The arguments to `shell` will be concatenated using spaces, which may or may not do what is expected. The variables set in local environments will not be expanded by the shell because local means local to the current process.

SEE ALSO

sh(1) .

vsig(1F)
Form and Menu Language Interpreter

NAME

`vsig -` synchronize a co-process with the controlling FMLI application

SYNOPSIS

`vsig`

DESCRIPTION

The `vsig` executable sends a `SIGUSR2` signal to the controlling FMLI process. This signal/alarm causes FMLI to execute the FMLI built-in command `checkworld` which causes all posted objects with a `reread` descriptor evaluating to TRUE to be reread. `vsig` takes no arguments.

EXAMPLES

The following is a segment of a shell program:

```
echo "Sending this string to an FMLI process"
vsig
```

The `vsig` executable flushes the output buffer *before* it sends the `SIGUSR2` signal to make sure the string is actually in the pipe created by the `cocreate` function.

NOTES

`vsig` synchronizes with FMLI. It should be used rather than `kill` to send a `SIGUSR2` signal to FMLI.

SEE ALSO

`kill(1)`, `coproc(1F)`, `kill(2)`, `signal(2)`.

Special C Functions(3X)

add_wch(3X) - cur_term(3X)

addch(3X)

NAME

`addch, mvaddch, mvwaddch, waddch -` add asingle-byte character and rendition to a window and advance the cursor

SYNOPSIS

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

```
#include <curses.h>
int addch(const chtype ch);
int mvaddch(int y, int x, const chtype ch);
int mvwaddch(WINDOW *win, int y, int x, const chtype ch);
int waddch(WINDOW *win, const chtype ch);
```

DESCRIPTION

The `addch()`, `mvaddch()`, `mvwaddch()` and `waddch()` functions place *ch* into the current or specified window at the current or specified position, and then advance the window's cursor position. These functions perform wrapping. These functions perform special-character processing.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise they return `ERR`.

ERRORS

No errors are defined.

NOTES

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the `A_` prefix.

SEE ALSO

`add_wch`(3X), `attroff`(3X), `curses`(3X), `doupdate`(3X), `curses`(5).

Character User Interface Reference Manual

addchstr(3X)

NAME

`addchstr, addchnstr, mvaddchstr, mvaddchnstr, mvwaddchstr, mvwaddchnstr, waddchstr, waddchnstr -` add string of single-byte characters and renditions to awindow

SYNOPSIS

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

```
#include <curses.h>
int addchstr(const chtype *chstr);
int addchnstr(const chtype *chstr, int n);
int mvaddchstr(int y, int x, const chtype *chstr);
int mvaddchnstr(int y, int x, const chtype *chstr, int n);
int mvwaddchstr(WINDOW *win, int y, int x, const chtype *chstr);
int mvwaddchnstr(WINDOW *win, int y, int x, const chtype *chstr,
int n);
int waddchstr(WINDOW *win, const chtype *chstr);
int waddchnstr(WINDOW *win, const chtype *chstr, int n);
```

DESCRIPTION

These functions overlay the contents of the current or specified window, starting at the current or specified position, with the contents of the array pointed to by *chstr* until a null `chtype` is encountered in the array pointed to by *chstr*.

These functions do not change the cursor position. These functions do not perform special-character processing. These functions do not perform wrapping.

The `addchnstr()`, `mvaddchnstr()`, `mvwaddchnstr()` and `waddchnstr()` functions copy at most *n* items, but no more than will fit on the current or specified line. If *n* is –1 then the whole string is copied, to the maximum number that fit on the current or specified line.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the `A_` prefix.

SEE ALSO

`addch`(3X), `add_wch`(3X), `add_wchstr`(3X), `curses`(3X), `curses`(5).

Character User Interface Reference Manual

addnstr(3X)

NAME

addnstr, addstr, mvaddnstr, mvaddstr, mvwaddnstr, mvwaddstrwaddnstr, waddstr - add astring of multi-byte characters without rendition to a window andadvance cursor

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
int addnstr(const char *str, int n);
int addstr(const char *str);
int mvaddnstr(int y, int x, const char *str, int n);
int mvaddstr(int y, int x, const char *str);
int mvwaddnstr(WINDOW *win, int y, int x, const char *str, int n);
int mvwaddstr(WINDOW *win, int y, int x, const char *str);
int waddnstr(WINDOW *win, const char *str, int n);
int waddstr(WINDOW *win, const char *str);
```

DESCRIPTION

These functions write the characters of the string *str* on the current or specified window starting at the current or specified position using the background rendition.

These functions advance the cursor position. These functions perform special character processing. These functions perform wrapping.

The `addstr()`, `mvaddstr()`, `mvwaddstr()` and `waddstr()` functions are similar to calling `mbstowcs()` on *str*, and then calling `addwstr()`, `mvaddwstr()`, `mvwaddwstr()` and `waddwstr()`, respectively.

The `addnstr()`, `mvaddnstr()`, `mvwaddnstr()` and `waddnstr()` functions use at most *n* bytes from *str*. These functions add the entire string when *n* is –. These functions are similar to calling `mbstowcs()` on the first *n* bytes of *str*, and then calling `addwstr()`, `mvaddwstr()`, `mvwaddwstr()` and `waddwstr()`, respectively.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`mbstowcs`(3C), `addnwstr`(3X), `curses`(3X), `curses`(5).

Character User Interface Reference Manual

addnwstr(3X)

NAME

addnwstr, addwstr, mvaddnwstr, mvaddwstr, mvwaddnwstr, mvwaddwstr, waddnwstr, waddwstr - adda wide-character string to a window and advance the cursor

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
int addnwstr(const wchar_t *wstr, int n);
int addwstr(const wchar_t *wstr);
int mvaddnwstr(int y, int x, const wchar_t *wstr, int n);
int mvaddwstr(int y, int x, const wchar_t *wstr);
int mvwaddnwstr(WINDOW *win, int y, int x, const wchar_t *wstr, int n);
int mvwaddwstr(WINDOW *win, int y, int x, const wchar_t *wstr);
int waddnwstr(WINDOW *win, const wchar_t *wstr, int n);
int waddwstr(WINDOW *win, const wchar_t *wstr);
```

DESCRIPTION

These functions write the characters of the wide character string *wstr* on the current or specified window at that window's current or specified cursor position.

These functions advance the cursor position. These functions perform special character processing. These functions perform wrapping.

The effect is similar to building a `cchar_t` from the `wchar_t` and the background rendition and calling `wadd_wch()`, once for each `wchar_t` character in the string. The cursor movement specified by the `mv()` functions occurs only once at the start of the operation.

The `addnwstr()`, `mvaddnwstr()`, `mvwaddnwstr()` and `waddnwstr()` functions write at most *n* wide characters. If *n* is –, then the entire string will be added.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`add_wch`(3X), `curses`(3X), `curses`(5).

Character User Interface Reference Manual

add_wch(3X)

NAME

`add_wch, mvadd_wch, mvwadd_wch, wadd_wch -` add acomplex character and rendition to a window

SYNOPSIS

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

```
#include <curses.h>
int add_wch(const cchar_t * wch);
int wadd_wch(WINDOW * win, const cchar_t *wch);
int mvadd_wch(int  y, int x, const cchar_t *wch);
int mvwadd_wch(WINDOW * win, int y, int x, const cchar_t *wch);
```

DESCRIPTION

These functions add information to the current or specified window at the current or specified position, and then advance the cursor. These functions perform wrapping. These functions perform special-character processing.

- If *wch* refers to a spacing character, then any previous character at that location is removed, a new character specified by *wch* is placed at that location with rendition specified by *wch*; then the cursor advances to the next spacing character on the screen.
- If *wch* refers to a non-spacing character, all previous characters at that location are preserved, the non-spacing characters of *wch* are added to the spacing complex character, and the rendition specified by *wch* is ignored.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`addch`(3X) , `curses`(3X) , `curses`(5) .

Character User Interface Reference Manual

add_wchnstr(3X)

NAME

`add_wchnstr, add_wchstr, mvadd_wchnstr, mvadd_wchstr, mvwadd_wchnstr, mvwadd_wchstr, wadd_wchnstr, wadd_wchstr -` add an array of complex charactersand renditions to a window

SYNOPSIS

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

```
#include <curses.h>
int add_wchnstr(const cchar_t * wchstr, int n);
int add_wchstr(const cchar_t * wchstr);
int wadd_wchnstr(WINDOW * win, const cchar_t *wchstr, int n);
int wadd_wchstr(WINDOW * win, const cchar_t *wchstr);
int mvadd_wchnstr(int  y, int x, const cchar_t *wchstr, int n);
int mvadd_wchstr(int  y, int x, const cchar_t *wchstr);
int mvwadd_wchnstr(WINDOW * win, int y, int x, const cchar_t *wchstr,
□□□□□□□□□□ int n);□
□
int mvwadd_wchstr(WINDOW *win, int y, int x, const cchar_t *wchstr);□
```

DESCRIPTION

These functions write the array of `cchar_t` specified by *wchstr* into the current or specified window starting at the current or specified cursor position.

These functions do not advance the cursor. The results are unspecified if *wchstr* contains any special characters.

The functions end successfully on encountering a null `cchar_t`. The functions also end successfully when they fill the current line. If a character cannot completely fit at the end of the current line, those columns are filled with the background character and rendition.

The `add_wchnstr()`, `mvadd_wchnstr()`, `mvwadd_wchnstr()` and `wadd_wchnstr()` functions end successfully after writing *n* `cchar_t`s (or the entire array of `cchar_t`s, if *n* is –).

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)` , `curses(5)` .

Character User Interface Reference Manual

attroff(3X)

NAME

attroff, attron, attrset, wattroff, wattron, wattrset - restricted window attribute control functions

SYNOPSIS

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

#include <curses.h>

int attroff(int attrs);
int attron(int attrs);
int attrset(int attrs);
int wattroff(WINDOW *win, int attrs);
int wattron(WINDOW *win, int attrs);
int wattrset(WINDOW *win, int attrs);
```

DESCRIPTION

These functions manipulate the window attributes of the current or specified window.

The `attroff()` and `wattroff()` functions turn off *attrs* in the current or specified window without affecting any others.

The `attron()` and `wattron()` functions turn on *attrs* in the current or specified window without affecting any others.

The `attrset()` and `wattrset()` functions set the background attributes of the current or specified window to *attrs*.

It is unspecified whether these functions can be used to manipulate attributes other than `A_BLINK`, `A_BOLD`, `A_DIM`, `A_REVERSE`, `A_STANDOUT` and `A_UNDERLINE`.

RETURN VALUE

These functions always return either `OK` or 1.

ERRORS

No errors are defined.

SEE ALSO

attr_get(3X), curses(3X), standend(3X), curses(5).

Character User Interface Reference Manual

attr_get(3X)

NAME

attr_get, attr_off, attr_on, attr_set, color_set, wattr_get, wattr_off, wattr_on, wattr_set, wcolor_set - window attribute control functions

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
int attr_get(attr_t *attrs, short *color_pair_number, void *opts);
int attr_off(attr_t attrs, void *opts);
int attr_on(attr_t attrs, void *opts);
int attr_set(attr_t attrs, short color_pair_number, void *opts);
int color_set(short color_pair_number, void *opts);
int wattr_get(WINDOW *win, attr_t *attrs, short *color_pair_number,
□
□□□□□□ void *opts);
int wattr_off(WINDOW *win, attr_t attrs, void *opts);
int wattr_on(WINDOW *win, attr_t attrs, void *opts);
int wattr_set(WINDOW *win, attr_t attrs, short color_pair_number,
□
□□□□□□ void *opts);
int wcolor_set(WINDOW *win, short color_pair_number, void *opts);□
```

DESCRIPTION

These functions manipulate the attributes and colour of the window rendition of the current or specified window.

The `attr_get()` and `wattr_get()` functions obtain the current rendition of a window. If *attrs* or *color_pair_number* is a null pointer, no information will be obtained on the corresponding rendition information and this is not an error.

The `attr_off()` and `wattr_off()` functions turn off *attrs* in the current or specified window without affecting any others.

The `attr_on()` and `wattr_on()` functions turn on *attrs* in the current or specified window without affecting any others.

The `attr_set()` and `wattr_set()` functions set the window rendition of the current or specified window to *attrs* and *color_pair_number*.

The `color_set()` and `wcolor_set()` functions set the window colour of the current or specified window to *color_pair_number*.

RETURN VALUE

These functions always return `OK`.

ERRORS

No errors are defined.

SEE ALSO

`attroff(3X)`, `curses(3X)`, `curses(5)`.

Character User Interface Reference Manual

baudrate(3X)

NAME

`baudrate -` get terminal baud rate

SYNOPSIS

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

```
#include <curses.h>
int baudrate(void);
```

DESCRIPTION

The `baudrate()` function extracts the output speed of the terminal in bits per second.

RETURN VALUE

The `baudrate()` function returns the output speed of the terminal.

ERRORS

No errors are defined.

SEE ALSO

`tcgetattr`(2), `curses`(3X), `curses`(5).

Character User Interface Reference Manual

beep(3X)

NAME

`beep` - audible signal

SYNOPSIS

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

```
#include <curses.h>
int beep(void);
```

DESCRIPTION

The `beep()` function alerts the user. It sounds the audible alarm on the terminal, or if that is not possible, it flashes the screen (visible bell). If neither signal is possible, nothing happens.

RETURN VALUE

The `beep()` function always returns `OK`.

ERRORS

No errors are defined.

NOTES

Nearly all terminals have an audible alarm, but only some can flash the screen.

SEE ALSO

`curses`(3X), `flash`(3X), `curses`(5).

Character User Interface Reference Manual

bkgd(3X)

NAME

`bkgd, bkgdset, getbkgd, wbkgd, wbkgdset -` turnoff the previous background attributes

SYNOPSIS

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

```
#include <curses.h>
int bkgd(chtype ch);
void bkgdset(chtype ch);
chtype getbkgd(WINDOW *win);
int wbkgd(WINDOW *win, chtype ch);
void wbkgdset(WINDOW *win, chtype ch);
```

DESCRIPTION

The `bkgdset()` and `wbkgdset()` functions turn off the previous background attributes, logical OR the requested attributes into the window rendition, and set the background property of the current or specified window based on the information in *ch*. If *ch* refers to a multi-column character, the results are undefined.

The `bkgd()` and `wbkgd()` functions turn off the previous background attributes, logical OR the requested attributes into the window rendition, and set the background property of the current or specified window and then apply this setting to every character position in that window:

- The rendition of every character on the screen is changed to the new window rendition.
- Wherever the former background character appears, it is changed to the new background character.

The `getbkgd()` function extracts the specified window's background character and rendition.

RETURN VALUE

Upon successful completion, `bkgd()` and `wbkgd()` return `OK`. Otherwise, they return `ERR`.

The `bkgdset()` and `wbkgdset()` functions do not return a value.

Upon successful completion, `getbkgd()` returns the specified window's background character and rendition. Otherwise, it returns `(chtype)ERR`.

ERRORS

No errors are defined.

NOTES

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the `A_` prefix.

SEE ALSO

`curses(3X)` , `curses(5)` .

Character User Interface Reference Manual

bkgrnd(3X)

NAME

`bkgrnd, bkgrndset, getbkgrnd, wbkgrnd, wbkgrndset, wgetbkgrnd -` turn off the previous backgroundattributes

SYNOPSIS

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

```
#include <curses.h>
int bkgrnd(const cchar_t * wch);
void bkgrndset(const cchar_t * wch);
int getbkgrnd(cchar_t * wch);
int wbkgrnd(WINDOW * win, const cchar_t *wch);
void wbkgrndset(WINDOW * win, const cchar_t *wch);
int wgetbkgrnd(WINDOW * win, cchar_t *wch);
```

DESCRIPTION

The `bkgrndset()` and `wbkgrndset()` functions turn off the previous background attributes, OR the requested attributes into the window rendition, and set the background property of the current or specified window based on the information in *wch*.

The `bkgrnd()` and `wbkgrnd()` functions turn off the previous background attributes, OR the requested attributes into the window rendition, and set the background property of the current or specified window and then apply this setting to every character position in that window:

- The rendition of every character on the screen is changed to the new window rendition.
- Wherever the former background character appears, it is changed to the new background character.

If *wch* refers to a non-spacing complex character for `bkgrnd()`, `bkgrndset()`, `wbkgrnd()` and `wbkgrndset()`, then *wch* is added to the existing spacing complex character that is the background character. If *wch* refers to a multi-column character, the results are unspecified.

The `getbkgrnd()` and `wgetbkgrnd()` functions store, into the area pointed to by *wch*, the value of the window's background character and rendition.

RETURN VALUE

The `bkgrndset()` and `wbkgrndset()` functions do not return a value.

Upon successful completion, the other functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)` , `curses(5)` .

Character User Interface Reference Manual

border(3X)

NAME

`border, wborder -` draw borders from single-byte characters andrenditions

SYNOPSIS

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

```
#include <curses.h>
```

`int border(chtype` *ls*, chtype *rs*, chtype *ts*, chtype *bs*, chtype *tl*,□
□□□□□ chtype *tr*, chtype *bl*, chtype *br*);

`int wborder(WINDOW *`*win*, chtype *ls*, chtype *rs*, chtype *ts*, chtype *bs*,□
□□□□□ chtype *tl*, chtype *tr*, chtype *bl*, chtype *br*);□

DESCRIPTION

The `border()` and `wborder()` functions draw a border around the edges of the current or specified window. These functions do not advance the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

The arguments in the left-hand column of the following table contain single-byte characters with renditions, which have the following uses in drawing the border:

Argument Name	Usage	Default Value
ls	Starting-column side	`ACS_VLINE`
rs	Ending-column side	`ACS_VLINE`
ts	First-line side	`ACS_HLINE`
bs	Last-line side	`ACS_HLINE`
tl	Corner of the first line and the starting column	`ACS_ULCORNER`
tr	Corner of the first line and the ending column	`ACS_URCORNER`
bl	Corner of the last line and the starting column	`ACS_BLCORNER`
br	Corner of the last line and the ending column	`ACS_BRCORNER`

If the value of any argument in the left-hand column is 0, then the default value in the right-hand column is used. If the value of any argument in the left-hand column is a multi-column character, the results are undefined.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the `A_` prefix.

SEE ALSO

`border_set`(3X), `box`(3X), `curses`(3X), `hline`(3X), `curses`(5).

Character User Interface Reference Manual

border_set(3X)

NAME

`border_set, wborder_set -` draw borders from complex charactersand renditions

SYNOPSIS

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

```
#include <curses.h>
int border_set(const cchar_t * ls, const cchar_t *rs, const cchar_t *ts,
const cchar_t * bs, const cchar_t *tl, const cchar_t *tr,
const cchar_t * bl, const cchar_t *br);
int wborder_set(WINDOW * win, const cchar_t *ls, const cchar_t *rs,
const cchar_t * ts, const cchar_t *bs,
const cchar_t * tl, const cchar_t *tr,
const cchar_t * bl, const cchar_t *br);
```

DESCRIPTION

The `border_set()` and `wborder_set()` functions draw a border around the edges of the current or specified window. These functions do not advance the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

The arguments in the left-hand column of the following table contain spacing complex characters with renditions, which have the following uses in drawing the border:

Argument Name	Usage	Default Value
ls	Starting-column side	`WACS_VLINE`
rs	Ending-column side	`WACS_VLINE`
ts	First-line side	`WACS_HLINE`
bs	Last-line side	`WACS_HLINE`
tl	Corner of the first line and the starting column	`WACS_ULCORNER`
tr	Corner of the first line and the ending column	`WACS_URCORNER`
bl	Corner of the last line and the starting column	`WACS_BLCORNER`
br	Corner of the last line and the ending column	`WACS_BRCORNER`

If the value of any argument in the left-hand column is a null pointer, then the default value in the right-hand column is used. If the value of any argument in the left-hand column is a multi-column character, the results are undefined.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`box_set`(3X), `curses`(3X), `hline_set`(3X), `curses`(5).

Character User Interface Reference Manual

box(3X)

NAME

box - draw borders from single-byte characters and renditions

SYNOPSIS

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

```
#include <curses.h>
```

int box(WINDOW *win, chtype verch, chtype horch);

DESCRIPTION

The box() function draws a border around the edges of the specified window. This function does not advance the cursor position. This function does not perform special character processing. This function does not perform wrapping.

The function box(*win*, *verch*, *horch*) has an effect equivalent to:

wborder(*win*, *verch*, *verch*, *horch*, *horch*, 0, 0, 0, 0);

RETURN VALUE

Upon successful completion, box() returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

NOTES

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO

border(3X), box_set(3X), curses(3X), hline(3X), curses(5).

Character User Interface Reference Manual

box_set(3X)

NAME

box_set - draw borders from complex characters and renditions

SYNOPSIS

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

#include <curses.h>

int box_set(WINDOW * *win*, const cchar_t **verch*, const cchar_t **horch*);

DESCRIPTION

The box_set() function draws a border around the edges of the specified window. This function does not advance the cursor position. This function does not perform special character processing. This function does not perform wrapping.

The function box_set(*win*, *verch*, *horch*) has an effect equivalent to:

wborder_set(*win*, *verch*, *verch*, *horch*, *horch*, NULL, NULL, NULL, NULL);

RETURN VALUE

Upon successful completion, this function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

SEE ALSO

border_set(3X), curses(3X), hline_set(3X), curses(5).

Character User Interface Reference Manual

can_change_color(3X)

NAME

can_change_color, color_content, COLOR_PAIR, has_colors, init_color, init_pair, start_color, pair_content, PAIR_NUMBER, COLOR_PAIRS, COLORS - colour manipulation functions

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
bool can_change_color(void);
int color_content(short color, short *red, short *green, short *blue);
int COLOR_PAIR(int n);
bool has_colors(void);
int init_color(short color, short red, short green, short blue);
int init_pair(short pair, short f, short b);
int start_color(void);
int pair_content(short pair, short *f, short *b);
int PAIR_NUMBER(int value);
extern int COLOR_PAIRS;
extern int COLORS;
```

DESCRIPTION

These functions manipulate colour on terminals that support colour.

Querying Capabilities

The has_colors() function indicates whether the terminal is a colour terminal. The can_change_color() function indicates whether the terminal is a colour terminal on which colours can be redefined.

Initialisation

The start_color() function must be called in order to enable use of colours and before any colour manipulation function is called. The function initialises eight basic colours (black, blue, green, cyan, red, magenta, yellow, and white) that can be specified by the colour macros (such as COLOR_BLACK) defined in <curses.h>. The initial appearance of these eight colours is not specified.

The function also initialises two global external variables:

COLORS
: defines the number of colours that the terminal supports. (See "Colour Identification" below.) If COLORS is 0, the terminal does not support redefinition of colours (and can_change_color() will return FALSE).

COLOR_PAIRS
: defines the maximum number of colour-pairs that the terminal supports. (See "User-defined Colour Pairs" below.)

The start_color() function also restores the colours on the terminal to terminal-specific initial values. The initial background colour is assumed to be black for all terminals.

Colour Identification

The init_color() function redefines colour number *color*, on terminals that support the redefinition of colours, to have the red, green, and blue intensity components specified by *red*, *green*, and *blue*, respectively. Calling init_color() also changes all occurrences of the specified colour on the screen to the new

definition.

The `color_content()` function identifies the intensity components of colour number *color*. It stores the red, green, and blue intensity components of this colour in the addresses pointed to by *red*, *green*, and *blue*, respectively.

For both functions, the `color()` argument must be in the range from 0 to and including `COLORS`–1. Valid intensity values range from 0 (no intensity component) up to and including 1000 (maximum intensity in that component).

User-Defined Colour Pairs

Calling `init_pair()` defines or redefines colour-pair number *pair* to have foreground colour *f* and background colour *b*. Calling `init_pair()` changes any characters that were displayed in the colour pair's old definition to the new definition and refreshes the screen.

After defining the colour pair, the macro `COLOR_PAIR(`*n*`)` returns the value of colour pair *n*. This value is the colour attribute as it would be extracted from a `chtype`. Conversely, the macro `PAIR_NUMBER(`*value*`)` returns the colour pair number associated with the colour attribute *value*.

The `pair_content()` function retrieves the component colours of a colour-pair number *pair*. It stores the foreground and background colour numbers in the variables pointed to by *f* and *b*, respectively.

With `init_pair()` and `pair_content()`, the value of *pair* must be in a range from 0 to and including `COLOR_PAIRS`–1. (There may be an implementation-specific upper limit on the valid value of *pair*, but any such limit is at least 63.) Valid values for *f* and *b* are the range from 0 to and including `COLORS`–1.

RETURN VALUE

The `has_colors()` function returns `TRUE` if the terminal can manipulate colors; otherwise, it returns `FALSE`.

The `can_change_color()` function returns `TRUE` if the terminal supports colors and can change their definitions; otherwise, it returns `FALSE`.

Upon successful completion, the other functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

To use these functions, `start_color()` must be called, usually right after `initscr()`.

The `can_change_color()` and `has_colors()` functions facilitate writing terminal-independent programs. For example, a programmer can use them to decide whether to use colour or some other video attribute.

On colour terminals, a typical value of `COLORS` is 8 and the macros such as `COLOR_BLACK` return a value within the range from 0 to and including 7. However, applications cannot rely on this to be true.

SEE ALSO

`attroff`(3X), `curses`(3X), `delscreen`(3X), `curses`(5).

Character User Interface Reference Manual

cbreak(3X)

NAME

`cbreak, nocbreak, noraw, raw -` input mode controlfunctions

SYNOPSIS

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

```
#include <curses.h>
int cbreak(void);
int nocbreak(void);
int noraw(void);
int raw(void);
```

DESCRIPTION

The `cbreak()` function sets the input mode for the current terminal to `cbreak` mode and overrides a call to `raw()`.

The `nocbreak()` function sets the input mode for the current terminal to Cooked Mode without changing the state of `ISIG` and `IXON`.

The `noraw()` function sets the input mode for the current terminal to Cooked Mode and sets the `ISIG` and `IXON` flags.

The `raw()` function sets the input mode for the current terminal to Raw Mode.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

If the application is not certain what the input mode of the process was at the time it called `initscr()`, it should use these functions to specify the desired input mode.

SEE ALSO

`curses(3X)` , `curses(5)` .

Character User Interface Reference Manual

chgat(3X)

NAME

`chgat, mvchgat, mvwchgat, wchgat -` changerenditions of characters in a window

SYNOPSIS

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

```
#include <curses.h>
```

`int chgat(int` *n*, attr_t *attr*, short *color*, const void **opts*);

`int mvchgat(int` *y*, int *x*, int *n*, attr_t *attr*, short *color*,
□□□□□ const void **opts*);

`int mvwchgat(WINDOW` **win*, int *y*, int *x*, int *n*, attr_t *attr*,
□□□□□□ short *color*, const void **opts*);

`int wchgat(WINDOW` **win*, int *n*, attr_t *attr*, short *color*,
□□□□□ const void **opts*);□

DESCRIPTION

These functions change the renditions of the next *n* characters in the current or specified window (or of the remaining characters on the current or specified line, if *n* is –), starting at the current or specified cursor position. The attributes and colors are specified by *attr* and *color* as for `setcchar()`.

These functions do not update the cursor. These functions do not perform wrapping.

A value of *n* that is greater than the remaining characters on a line is not an error.

The *opts* argument is reserved for definition in a future edition of this document. Currently, the application must provide a null pointer as *opts*.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X) , `setcchar`(3X) , `curses`(5) .

Character User Interface Reference Manual

clear(3X)

NAME

`clear, erase, wclear, werase -` clear a window

SYNOPSIS

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

```
#include <curses.h>
int clear(void);
int erase(void);
int wclear(WINDOW *win);
int werase(WINDOW *win);
```

DESCRIPTION

The `clear()`, `erase()`, `wclear()` and `werase()` functions clear every position in the current or specified window.

The `clear()` and `wclear()` functions also achieve the same effect as calling `clearok()`, so that the window is cleared completely on the next call to `wrefresh()` for the window and is redrawn in its entirety.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`clearok`(3X), `curses`(3X), `doupdate`(3X), `curses`(5).

Character User Interface Reference Manual

clearok(3X)

NAME

`clearok, idlok, leaveok, scrollok, setscrreg, wsetscrreg -` terminal output control functions

SYNOPSIS

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

```
#include <curses.h>
int clearok(WINDOW *win, bool bf);
int idlok(WINDOW *win, bool bf);
int leaveok(WINDOW *win, bool bf);
int scrollok(WINDOW *win, bool bf);
int setscrreg(int top, int bot);
int wsetscrreg(WINDOW *win, int top, int bot);
```

DESCRIPTION

These functions set options that deal with output within Curses.

The `clearok()` function assigns the value of *bf* to an internal flag in the specified window that governs clearing of the screen during a refresh. If, during a refresh operation on the specified window, the flag in `curscr` is `TRUE` or the flag in the specified window is `TRUE`, then the implementation clears the screen, redraws it in its entirety, and sets the flag to `FALSE` in `curscr` and in the specified window. The initial state is unspecified.

The `idlok()` function specifies whether the implementation may use the hardware insert-line, delete-line, and scroll features of terminals so equipped. If *bf* is `TRUE`, use of these features is enabled. If *bf* is `FALSE`, use of these features is disabled and lines are instead redrawn as required. The initial state is `FALSE`.

The `leaveok()` function controls the cursor position after a refresh operation. If *bf* is `TRUE`, refresh operations on the specified window may leave the terminal's cursor at an arbitrary position. If *bf* is `FALSE`, then at the end of any refresh operation, the terminal's cursor is positioned at the cursor position contained in the specified window. The initial state is `FALSE`.

The `scrollok()` function controls the use of scrolling. If *bf* is `TRUE`, then scrolling is enabled for the specified window, with the consequences discussed in in "X/Open Curses, Issue 4, Version 2, Section 3.4.2". If *bf* is `FALSE`, scrolling is disabled for the specified window. The initial state is `FALSE`.

The `setscrreg()` and `wsetscrreg()` functions define a software scrolling region in the current or specified window. The *top* and *bot* arguments are the line numbers of the first and last line defining 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 last line of the margin causes all lines in the scrolling region to scroll one line in the direction of the first line. Only characters in the window are scrolled. If a software scrolling region is set and `scrollok()` is not enabled, an attempt to move off the last line of the margin does not reposition any lines in the scrolling region.

RETURN VALUE

Upon successful completion, `setscrreg()` and `wsetscrreg()` return `OK`. Otherwise, they return `ERR`.

The other functions always return `OK`.

ERRORS

No errors are defined.

NOTES

The only reason to enable the `idlok()` feature is to use scrolling to achieve the visual effect of motion of a partial window, such as for a screen editor. In other cases, the feature can be visually annoying.

The `leaveok()` option provides greater efficiency for applications that do not use the cursor.

SEE ALSO

`clear`(3X), `curscr`(3X), `curses`(3X), `delscreen`(3X), `doupdate`(3X), `scrl`(3X), `curses`(5).

Character User Interface Reference Manual

clrtobot(3X)

NAME

`clrtobot, wclrtobot -` clear from cursor to end of window

SYNOPSIS

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

```
#include <curses.h>
int clrtobot(void);
int wclrtobot(WINDOW *win);
```

DESCRIPTION

The `clrtobot()` and `wclrtobot()` functions erase all lines following the cursor in the current or specified window, and erase the current line from the cursor to the end of the line, inclusive. These functions do not update the cursor.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)` , `doupdate(3X)` , `curses(5)` .

Character User Interface Reference Manual

clrtoeol(3X)

NAME

clrtoeol, wclrtoeol - clear from cursor to end of line

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
int clrtoeol(void);
int wclrtoeol(WINDOW *win);
```

DESCRIPTION

The `clrtoeol()` and `wclrtoeol()` functions erase the current line from the cursor to the end of the line, inclusive, in the current or specified window. These functions do not update the cursor.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

curses(3X), doupdate(3X), curses(5).

Character User Interface Reference Manual

copywin(3X)

NAME

`copywin` - copy a region of a window

SYNOPSIS

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

```
#include <curses.h>
int copywin(const WINDOW *srcwin, WINDOW *dstwin, int sminrow,
      int smincol, int dminrow, int dmincol, int dmaxrow,
      int dmaxcol, int overlay);
```

DESCRIPTION

The `copywin()` function provides a finer granularity of control over the `overlay()` and `overwrite()` functions. As in the `prefresh()` function, 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 `overlay()` is `TRUE`, then copying is non-destructive, as in `overlay()`. If `overlay()` is `FALSE`, then copying is destructive, as in `overwrite()`.

RETURN VALUE

Upon successful completion, `copywin()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)`, `newpad(3X)`, `overlay(3X)`, `curses(5)`.

Character User Interface Reference Manual

curscr(3X)

NAME

`curscr` - current window

SYNOPSIS

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

```
#include <curses.h>
extern WINDOW *curscr;
```

DESCRIPTION

The external variable `curscr` points to an internal data structure. It can be specified as an argument to certain functions, such as `clearok()`, where permitted in this specification.

SEE ALSO

`clearok`(3X), `curses`(3X), `curses`(5).

Character User Interface Reference Manual

curses(3X)
Enhanced Curses

NAME

curses - character-oriented screen handling and optimization package

SYNOPSIS

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

#include <curses.h>

DESCRIPTION

The curses library routines give the user a terminal-independent method of updating character screens with reasonable optimization. A program using these routines must be compiled with the −lcurses option of cc.

The curses package allows: overall screen, window and pad manipulation; output to windows and pads; reading terminal input; control over terminal and curses input and output options; environment query routines; color manipulation; use of soft label keys; terminfo access; and access to low-level curses routines.

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. The routine endwin() must be called before exiting. To get character-at-a-time input without echoing (most interactive, screen oriented programs want this), the following sequence should be used:

```
initscr,cbreak,noecho;
```

Most programs would additionally use the sequence:

```
nonl,intrflush(stdscr,FALSE),keypad(stdscr,TRUE);
```

Before a curses program is run, the tab stops of the terminal 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. See terminfo(4) for further details.

Screens, Windows and Terminals

Screen

A screen is the physical output device of the terminal. In curses, a SCREEN data type is an opaque data type associated with a terminal. Each window (see below) is associated with a SCREEN.

Windows

The curses functions permit manipulation of window objects, which can be thought of as two-dimensional arrays of characters and their renditions. A default window called stdscr, which is the size of the terminal screen, is supplied. Others may be created with newwin().

Variables declared as WINDOW* refer to windows (and to subwindows, derived windows, and pads, as described below). These data structures are manipulated with functions described on the 3X reference manual pages. Among the most basic functions are move() and addch(). More general versions of these functions are included that allow a process to specify a window.

After using functions to manipulate a window, refresh() is called, telling curses to make the character-oriented screen look like stdscr.

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 line drawing characters and input values use names defined in <curses.h> [see curses(5)].

Each window has a flag that indicates that the information in the window could differ from the information displayed on the terminal device. Making any change to the contents of the window, moving or modifying the window, or setting the window's cursor position, sets this flag (*touches* the window). Refreshing the window clears this flag. For further information, see is_linetouched(3X) .

Subwindows

A subwindow is a window, created within another window (called the *parent window*), and positioned relative to the parent window. A subwindow can be created by calling `derwin()`, `newpad()` or `subwin()`. Changes made to a subwindow do not affect its parent window.

Subwindows can be created from a parent window by calling `subwin()`. The position and size of subwindows on the screen must be identical to or totally within the parent window. Changes to a parent window will affect both subwindows and derived windows. Window clipping is not a property of subwindows.

Ancestors

The term ancestor refers to a window's parent, or its parent, and so on.

Derived Windows

Derived windows are subwindows whose position is defined by reference to the parent window rather than in absolute screen coordinates. Derived windows are otherwise no different from subwindows.

Pads

A pad is a specialised case of subwindow that is not necessarily associated with a viewable part of a screen. Functions that deal with pads are all discussed in `newpad(3X)` .

Terminal

A terminal is the logical input and output device through which character-based applications interact with the user. `TERMINAL` is an opaque data type associated with a terminal. A `TERMINAL` data structure primarily contains information about the capabilities of the terminal, as defined by `terminfo`. A `TERMINAL` also contains information about the terminal modes and current state for input and output operations. Each screen (see above) is associated with a `TERMINAL`.

Input Processing

The `curses` input model provides a variety of ways to obtain input from the keyboard.

Keypad Processing

The application can enable or disable "keypad translation" by calling `keypad()`. When translation is enabled, `curses` attempts to translate a sequence of terminal input that represents the pressing of a function key into a single key code. When translation is disabled, `curses` passes terminal input to the application without such translation, and any interpretation of the input as representing the pressing of a keypad key must be done by the application.

The complete set of key codes for keypad keys that `curses` can process is specified by the constants defined in `<curses.h>`. whose names begin with `KEY_`. Each terminal type described in the `terminfo` database may support some or all of these key codes. The `terminfo` database specifies the sequence of input characters from the terminal type that correspond to each key code [see `keypad(3X)`].

The `curses` implementation cannot translate keypad keys on terminals where pressing the keys does not transmit a unique sequence.

When translation is enabled and a character that could be the beginning of a function key (such as escape) is received, `curses` notes the time and begins accumulating characters. If `curses` receives additional characters that represent the pressing of a keypad key, within an unspecified interval from the time the first character was received, then `curses` converts this input to a key code for presentation to the application. If such characters are not received during this interval, translation of this input does not occur and the individual characters are presented to the application separately. (Because `curses` waits for this interval to accumulate a key code, many terminals experience a delay between the time a user presses the escape key and the time the escape is returned to the application.)

In addition, No Timeout Mode provides that in any case where `curses` has received part of a function key sequence, it waits indefinitely for the complete key sequence. The "unspecified interval" in the previous paragraph becomes infinite in No Timeout Mode. No Timeout Mode allows the use of function keys over slow communication lines. No Timeout Mode lets the user type the individual characters of a function key sequence, but also delays application response when the user types a character (not a function key) that begins a function key sequence. For this reason, in No Timeout Mode many terminals will appear to hang

between the time a user presses the escape key and the time another key is pressed. No Timeout Mode is switchable by calling `notimeout()`.

If any special characters (<backspace>, <carriage return>, <newline>, <tab>) are defined or redefined to be characters that are members of a function key sequence, then `curses` will be unable to recognise and translate those function keys.

Several of the modes discussed below are described in terms of availability of input. If keypad translation is enabled, then input is not available once `curses` has begun receiving a keypad sequence until the sequence is completely received or the interval has elapsed.

Input Mode

The XBD ("Special Characters") defines flow-control characters, the interrupt character, the erase character, and the kill character. Four mutually-exclusive `curses` modes let the application control the effect of these input characters:

Input Mode	Effect
Cooked Mode	This achieves normal line-at-a-time processing with all special characters handled outside the application. This achieves the same effect as canonical-mode input processing as specified in the XBD. The state of the `ISIG` and `IXON` flags are not changed upon entering this mode by calling `nocbreak()`, and are set upon entering this mode by calling `noraw()`. The implementation supports erase and kill characters from any supported locale, no matter what the width of the character is.
cbreak Mode	Characters typed by the user are immediately available to the application and `curses` does not perform special processing on either the erase character or the kill character. An application can select *cbreak* mode to do its own line editing but to let the abort character be used to abort the task. This mode achieves the same effect as non-canonical-mode, Case B input processing (with `MIN` set to 1 and `ICRNL` cleared) as specified in the XBD. The state of the `ISIG` and `IXON` flags are not changed upon entering this mode.
Half-Delay Mode	The effect is the same as *cbreak*, except that input functions wait until a character is available or an interval defined by the application elapses, whichever comes first. This mode achieves the same effect as non-canonical-mode, Case C input processing (with `TIME` set to the value specified by the application) as specified in the XBD. The state of the `ISIG` and `IXON` flags are not changed upon entering this mode.
Raw Mode	Raw mode gives the application maximum control over terminal input. The application sees each character as it is typed. This achieves the same effect as non-canonical mode, Case D input processing as specified in the XBD. The `ISIG` and `IXON` flags are cleared upon entering this mode.

The terminal interface settings are recorded when the process calls `initscr()` or `newterm()` to initialise `curses` and restores these settings when `endwin()` is called. The initial input mode for `curses` operations is unspecified unless the implementation supports Enhanced Curses compliance, in which the initial input mode is *cbreak* mode.

The behaviour of the BREAK key depends on other bits in the display driver that are not set by `curses`.

Delay Mode

Two mutually-exclusive delay modes specify how quickly certain `curses` functions return to the application when there is no terminal input waiting when the function is called:

No Delay
: The function fails.

Delay
: The application waits until the implementation passes text through to the application. If *cbreak* or Raw

Mode is set, this is after one character. Otherwise, this is after the first <newline> character, end-of-line character, or end-of-file character.

The effect of No Delay Mode on function key processing is unspecified.

Echo Processing

Echo mode determines whether `curses` echoes typed characters to the screen. The effect of Echo mode is analogous to the effect of the `ECHO` flag in the local mode field of the `termios` structure associated with the terminal device connected to the window. However, `curses` always clears the `ECHO` flag when invoked, to inhibit the operating system from performing echoing. The method of echoing characters is not identical to the operating system's method of echoing characters, because `curses` performs additional processing of terminal input.

If in Echo mode, `curses` performs its own echoing: Any visible input character is stored in the current or specified window by the input function that the application called, at that window's cursor position, as though `addch()` were called, with all consequent effects such as cursor movement and wrapping.

If not in Echo mode, any echoing of input must be performed by the application. Applications often perform their own echoing in a controlled area of the screen, or do not echo at all, so they disable Echo mode.

It may not be possible to turn off echo processing for synchronous and networked asynchronous terminals because echo processing is done directly by the terminals. Applications running on such terminals should be aware that any characters typed will appear on the screen at wherever the cursor is positioned.

Environment Variables

If the environment variables `LINES` and `COLUMNS` are set, or if the program is executing in a window environment, line and column information in the environment will override information read by `terminfo`. This would effect a program running in an AT&T 630 layer, for example, where the size of a screen is changeable.

If the environment variable `TERMINFO` is defined, any program using `curses` checks for a local terminal definition before checking in the standard place. For example, if `TERM` is set to `att4424`, then the compiled terminal definition is found in

`/usr/share/lib/terminfo/a/att4424`.

(The `a` is copied from the first letter of `att4424` to avoid creation of huge directories.) However, if `TERMINFO` is set to `$HOME/myterms`, `curses` first checks

`$HOME/myterms/a/att4424`,

and if that fails, it then checks

`/usr/share/lib/terminfo/a/att4424`.

This is useful for developing experimental definitions or when write permission in `/usr/share/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. The constants `TRUE` and `FALSE` have the values `1` and `0`, respectively.

The `curses` routines also define the `WINDOW * ` variable `curscr` which is used for certain low-level operations like clearing and redrawing a screen containing garbage. The `curscr` can be used in only a few routines.

Function Name Conventions

The 3X reference manual pages present families of multiple `curses` functions. Most function families have different functions that give the programmer the following options:

- A function with the basic name operates on the window `stdscr`. A function with the same name plus the `w` prefix [see `w`(3X)] operates on a window specified by the *win* argument.

  When the reference manual page for a function family refers to the current or specified window, it means `stdscr` for the basic functions and the window specified by *win* for any `w` function.

  Functions whose names have the `p` prefix require an argument that is a pad instead of a window.
- A function with the basic name operates based on the current cursor position (of the current or specified

window, as described above). A function with the same name plus the mv [see mv(3X)] prefix moves the cursor to a position specified by the *y* and *x* arguments before performing the specified operation.

When the reference manual page for a function family refers to the current or specified position, it means the cursor position for the basic functions and the position (*y*, *x*) for any mv function.

The mvw prefix exists and combines the mv semantics discussed here with the w semantics discussed above. The window argument is always specified before the coordinates.

- A function with the basic name is often provided for historical compatibility and operates only on single-byte characters. A function with the same name plus the w infix operates on wide (multi-byte) characters. A function with the same name plus the _w infix operates on complex characters and their renditions.
- When a function with the basic name operates on a single character, there is sometimes a function with the same name plus the n infix that operates on multiple characters. An *n* argument specifies the number of characters to process. The respective manual page specifies the outcome if the value of *n* is inappropriate.

Function Families Provided

Function Names	Description	s	w	c	Refer to
	Add (Overwrite)				
[mv][w]addch	add a character	Y	Y	Y	addch(3X)
[mv][w]addch[n]str	add a character string	N	N	N	addchstr(3X)
[mv][w]add[n]str	add a string	Y	Y	Y	addnstr(3X)
[mv][w]add[n]wstr	add a wide character string	Y	Y	Y	addnwstr(3X)
[mv][w]add_wch	add a wide character and rendition	Y	Y	Y	add_wch(3X)
[mv][w]add_wch[n]str	add an array of wide characters and renditions	?	N	N	add_wchnstr(3X)
	Change Renditions				
[mv][w]chgat	change renditions of characters in a window	–	N	N	chgat(3X)
	Delete				
[mv][w]delch	delete a character	–	–	N	delch(3X)
	Get (Input from Keyboard to Window)				
[mv][w]getch	get a character	Y	Y	Y	getch(3X)
[mv][w]get[n]str	get a character string	Y	Y	Y	getnstr(3X)
[mv][w]get_wch	get a wide character	Y	Y	Y	get_wch(3X)
[mv][w]get[n]_wstr	get an array of wide characters and key codes	Y	Y	Y	get_wstr(3X)
	Explicit Cursor Movement				
[w]move	move the cursor	–	–	–	move(3X)
	Input (Read Back from Window)				
[mv][w]inch	input a character	–	–	–	inch(3X)
[mv][w]inch[n]str	input an array of characters and attributes	–	–	–	inchnstr(3X)

[mv][w]in[n]str	input a string	–	–	–	innstr(3X)
[mv][w]in[n]wstr	input a string of wide characters	–	–	–	innwstr(3X)
[mv][w]in_wch	input a wide character and rendition	–	–	–	in_wch(3X)
[mv][w]in_wch[n]str	input an array of wide characters and renditions	–	–	–	inwchnstr(3X)
	Insert				
[mv][w]insch	insert a character	Y	N	N	insch(3X)
[mv][w]ins[n]str	insert a character string	Y	N	N	insnstr(3X)
[mv][w]ins_[n]wstr	insert a wide character string	Y	N	N	ins_nwstr(3X)
[mv][w]ins_wch	insert a wide character	Y	N	N	ins_wch(3X)
	Print and Scan				
[mv][w]printw	print formatted output	–	–	–	mvprintw(3X)
[mv][w]scanw	convert formatted input	–	–	–	mvscanw(3X)

Legend:

The following notation indicates the effect when characters are moved to the screen. (For the Get functions, this applies only when echoing is enabled.)

s "Y" means these functions perform special-character processing. "N" means they do not. "?" means the results are unspecified when these functions are applied to special characters.

w "Y" means these functions perform wrapping. "N" means they do not.

c "Y" means these functions advance the cursor. "N" means they do not.

– The attribute specified by this column does not apply to these functions.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

SEE ALSO

mv(3X), no(3X), w(3X), terminfo(4), curses(5), term(5), unctrl(5).

Character User Interface Reference Manual

curs_set(3X)

NAME

`curs_set` - set the cursor mode

SYNOPSIS

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

```
#include <curses.h>
int curs_set(int visibility);
```

DESCRIPTION

The `curs_set()` function sets the appearance of the cursor based on the value of *visibility*:

Value of *visibility*	Appearance of Cursor
0	Invisible
1	Terminal-specific normal mode
2	Terminal-specific high visibility mode

The terminal does not necessarily support all the above values.

RETURN VALUE

If the terminal supports the cursor mode specified by *visibility*, then `curs_set()` returns the previous cursor state. Otherwise, the function returns `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X) , `curses`(5) .

Character User Interface Reference Manual

cur_term(3X)

NAME

`cur_term` - current terminal information

SYNOPSIS

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

```
#include <term.h>
extern TERMINAL *cur_term;
```

DESCRIPTION

The external variable `cur_term` identifies the record in the `terminfo` database associated with the terminal currently in use.

SEE ALSO

`curses(3X)`, `set_curterm(3X)`, `tigetflag(3X)`, `term(5)`.

def-prog-mode(3X) - hline-set(3X)

Character User Interface Reference Manual

def_prog_mode(3X)

NAME

`def_prog_mode, def_shell_mode, reset_prog_mode, reset_shell_mode` - save/restore program or shell terminal modes

SYNOPSIS

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

```
#include <curses.h>
int def_prog_mode(void);
int def_shell_mode(void);
int reset_prog_mode(void);
int reset_shell_mode(void);
```

DESCRIPTION

The `def_prog_mode()` function saves the current terminal modes as the "program" (in Curses) state for use by `reset_prog_mode()`.

The `def_shell_mode()` function saves the current terminal modes as the "shell" (not in Curses) state for use by `reset_shell_mode()`.

The `reset_prog_mode()` function restores the terminal to the "program" (in Curses) state.

The `reset_shell_mode()` function restores the terminal to the "shell" (not in Curses) state.

These functions affect the mode of the terminal associated with the current screen.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

The `initscr()` function achieves the effect of calling `def_shell_mode()` to save the prior terminal settings so they can be restored during the call to `endwin()`, and of calling `def_prog_mode()` to specify an initial definition of the program terminal mode.

Applications normally do not need to refer to the shell terminal mode. Applications may find it useful to save and restore the program terminal mode.

SEE ALSO

`curses(3X)`, `doupdate(3X)`, `endwin(3X)`, `initscr(3X)`, `curses(5)`.

Character User Interface Reference Manual

delay_output(3X)

NAME

`delay_output` - delay output

SYNOPSIS

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

```
#include <curses.h>
int delay_output(int ms);
```

DESCRIPTION

On terminals that support pad characters, `delay_output()` pauses the output for at least *ms* milliseconds. Otherwise, the length of the delay is unspecified.

RETURN VALUE

Upon successful completion, `delay_output()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

NOTES

Whether or not the terminal supports pad characters, the `delay_output()` function is not a precise method of timekeeping.

SEE ALSO

`curses(3X)`, `napms(3X)`, `curses(5)`.

Character User Interface Reference Manual

delch(3X)

NAME

`delch, mvdelch, mvwdelch, wdelch -` delete acharacter from a window.

SYNOPSIS

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

```
#include <curses.h>
int delch(void);
int mvdelch(int y, int x);
int mvwdelch(WINDOW *win, int y, int x);
int wdelch(WINDOW *win);
```

DESCRIPTION

These functions delete the character at the current or specified position in the current or specified window. This function does not change the cursor position.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)`, `curses(5)`.

Character User Interface Reference Manual

deleteln(3X)

NAME

`deleteln, wdeleteln -` delete lines in a window

SYNOPSIS

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

```
#include <curses.h>
int deleteln(void);
int wdeleteln(WINDOW *win);
```

DESCRIPTION

The `deleteln()` and `wdeleteln()` functions delete the line containing the cursor in the current or specified window and move all lines following the current line one line toward the cursor. The last line of the window is cleared. The cursor position does not change.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)`, `insdelln(3X)`, `curses(5)`.

Character User Interface Reference Manual

delscreen(3X)

NAME

`delscreen -` free storage associated with a screen

SYNOPSIS

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

```
#include <curses.h>
void delscreen(SCREEN *sp);
```

DESCRIPTION

The `delscreen()` function frees storage associated with the `SCREEN` pointed to by *sp*.

RETURN VALUE

The `delscreen()` function does not return a value.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X), `endwin`(3X), `initscr`(3X), `curses`(5).

Character User Interface Reference Manual

delwin(3X)

NAME

`delwin -` delete a window

SYNOPSIS

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

```
#include <curses.h>
int delwin(WINDOW *win);
```

DESCRIPTION

The `delwin()` function deletes *win*, freeing all memory associated with it. The application must delete subwindows before deleting the main window.

RETURN VALUE

Upon successful completion, `delwin()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X), `derwin`(3X), `dupwin`(3X), `curses`(5).

Character User Interface Reference Manual

del_curterm(3X)

NAME

del_curterm, restartterm, set_curterm, setupterm - interfaces to the terminfo database

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <term.h>
int del_curterm(TERMINAL *oterm);
int restartterm(char *term, int fildes, int *errret);
TERMINAL *set_curterm(TERMINAL *nterm);
int setupterm(char *term, int fildes, int *errret);
```

DESCRIPTION

These functions retrieve information from the terminfo(4) database.

To gain access to the terminfo database, setupterm() must be called first. It is automatically called by initscr() and newterm(). The setupterm() function initialises the other functions to use the terminfo record for a specified terminal (which depends on whether use_env() was called). It sets the cur_term external variable to a TERMINAL structure that contains the record from the terminfo database for the specified terminal.

The terminal type is the character string *term*; if *term* is a null pointer, the environment variable TERM is used. If TERM is not set or if its value is an empty string, then "unknown" is used as the terminal type. The application must set *fildes* to a file descriptor, open for output, to the terminal device, before calling setupterm(). If *errret* is not null, the integer it points to is set to one of the following values to report the function outcome:

-1 The terminfo database was not found (function fails).

0 The entry for the terminal was not found in terminfo (function fails).

1 Success.

If setupterm() detects an error and *errret* is a null pointer, setupterm() writes a diagnostic message and exits.

A simple call to setupterm() that uses all the defaults and sends the output to stdout is:

```
setupterm((char *)0, fileno(stdout), (int *)0);
```

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

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

The restartterm() function assumes a previous call to setupterm() (perhaps from initscr() or newterm()). It lets the application specify a different terminal type in *term* and updates the information returned by baudrate() based on *fildes*, but does not destroy other information created by initscr(), newterm() or setupterm().

RETURN VALUE

Upon successful completion, set_curterm() returns the previous value of cur_term. Otherwise, it returns a null pointer.

Upon successful completion, the other functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

NOTES

An application would call `setupterm()` if it required access to the `terminfo` database but did not otherwise need to use Curses.

SEE ALSO

`putc`(3S), `baudrate`(3X), `curses`(3X), `erasechar`(3X), `has_ic`(3X), `longname`(3X), `termattrs`(3X), `termname`(3X), `tgetent`(3X), `tigetflag`(3X), `use_env`(3X), `terminfo`(4), `term`(5).

Character User Interface Reference Manual

derwin(3X)

NAME

derwin, newwin, subwin - window creation functions

SYNOPSIS

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

#include <curses.h>

WINDOW *derwin(WINDOW **orig*, int *nlines*, int *ncols*, int *begin_y*, int *begin_x*);

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

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

DESCRIPTION

The derwin() function is the same as subwin(), except that *begin_y* and *begin_x* are relative to the origin of the window *orig* rather than absolute screen positions.

The newwin() function creates a new window with *nlines* lines and *ncols* columns, positioned so that the origin is (*begin_y*, *begin_x*). If *nlines* is zero, it defaults to LINES – *begin_y*; if *ncols* is zero, it defaults to COLS – *begin_x*.

The subwin() function creates a new window with *nlines* lines and *ncols* columns, positioned so that the origin is at (*begin_y*, *begin_x*). (This position is an absolute screen position, not a position relative to the window *orig*.) If any part of the new window is outside *orig*, the function fails and the window is not created.

RETURN VALUE

Upon successful completion, these functions return a pointer to the new window. Otherwise, they return a null pointer.

ERRORS

No errors are defined.

NOTES

Before performing the first refresh of a subwindow, portable applications should call touchwin() or touchline() on the parent window.

Each window maintains internal descriptions of the screen image and status. The screen image is shared among all windows in the window hierarchy. Refresh operations rely on information on what has changed within a window, which is private to each window. Refreshing a window, when updates were made to a different window, may fail to perform needed updates because the windows do not share this information.

A new full-screen window is created by calling:

```
newwin(0, 0, 0, 0);
```

SEE ALSO

curses(3X), delwin(3X), doupdate(3X), is_linetouched(3X), curses(5).

Character User Interface Reference Manual

doupdate(3X)

NAME

doupdate, refresh, wnoutrefresh, wrefresh - refresh windows and lines

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
int doupdate(void);
int refresh(void);
int wnoutrefresh(WINDOW *win);
int wrefresh(WINDOW *win);
```

DESCRIPTION

The `refresh()` and `wrefresh()` functions refresh the current or specified window. The functions position the terminal's cursor at the cursor position of the window, except that if the `leaveok()` mode has been enabled, they may leave the cursor at an arbitrary position.

The `wnoutrefresh()` function determines which parts of the terminal may need updating. The `doupdate()` function sends to the terminal the commands to perform any required changes.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

Refreshing an entire window is typically more efficient than refreshing several subwindows separately. An efficient sequence is to call `wnoutrefresh()` on each subwindow that has changed, followed by a call to `doupdate()`, which updates the terminal.

The `refresh()` or `wrefresh()` function (or `wnoutrefresh()` followed by `doupdate()`) must be called to send output to the terminal, as other Curses functions merely manipulate data structures.

SEE ALSO

`clearok`(3X), `curses`(3X), `redrawwin`(3X), `curses`(5).

Character User Interface Reference Manual

dupwin(3X)

NAME

`dupwin` - duplicate a window

SYNOPSIS

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

```
#include <curses.h>
WINDOW *dupwin(WINDOW *win);
```

DESCRIPTION

The `dupwin()` function creates a duplicate of the window *win*.

RETURN VALUE

Upon successful completion, `dupwin()` returns a pointer to the new window. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X), `derwin`(3X), `doupdate`(3X), `curses`(5).

Character User Interface Reference Manual

echo(3X)

NAME

`echo, noecho` - enable/disable terminal echo

SYNOPSIS

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

```
#include <curses.h>
int echo(void);
int noecho(void);
```

DESCRIPTION

The `echo()` function enables Echo mode for the current screen. The `noecho()` function disables Echo mode for the current screen. Initially, curses software echo mode is enabled and hardware echo mode of the tty driver is disabled. `echo()` and `noecho()` control software echo only. Hardware echo must remain disabled for the duration of the application, else the behaviour is undefined.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)`, `getch(3X)`, `curses(5)`.

Character User Interface Reference Manual

echochar(3X)

NAME

`echochar, wechochar -` echo single-byte character and renditionto a window and refresh

SYNOPSIS

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

```
#include <curses.h>
int echochar(const chtype ch);
int wechochar(WINDOW *win, const chtype ch);
```

DESCRIPTION

The `echochar()` function is equivalent to a call to `addch()` followed by a call to `refresh()`.

The `wechochar()` function is equivalent to a call to `waddch()` followed by a call to `wrefresh()`.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise they return `ERR`.

ERRORS

No errors are defined.

NOTES

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the `A_` prefix.

SEE ALSO

`addch`(3X), `curses`(3X), `doupdate`(3X), `echo_wchar`(3X), `curses`(5).

Character User Interface Reference Manual

echo_wchar(3X)

NAME

`echo_wchar, wecho_wchar -` write a complex character andimmediately refreshthe window

SYNOPSIS

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

```
#include <curses.h>
int echo_wchar(const cchar_t * wch);
int wecho_wchar(WINDOW * win, const cchar_t *wch);
```

DESCRIPTION

The `echo_wchar()` function is equivalent to calling `add_wch()` and then calling `refresh()`.

The `wecho_wchar()` function is equivalent to calling `wadd_wch()` and then calling `wrefresh()`.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`addch`(3X), `add_wch`(3X), `curses`(3X), `doupdate`(3X), `curses`(5).

Character User Interface Reference Manual

endwin(3X)

NAME

`endwin` - suspend Curses session

SYNOPSIS

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

```
#include <curses.h>
int endwin(void);
```

DESCRIPTION

The `endwin()` function restores the terminal after Curses activity by at least restoring the saved shell terminal mode, flushing any output to the terminal and moving the cursor to the first column of the last line of the screen. Refreshing a window resumes program mode. The application must call `endwin()` for each terminal being used before exiting. If `newterm()` is called more than once for the same terminal, the first screen created must be the last one for which `endwin()` is called.

RETURN VALUE

Upon successful completion, `endwin()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

NOTES

The `endwin()` function does not free storage associated with a screen, so `delscreen()` should be called after `endwin()` if a particular screen is no longer needed.

To leave Curses mode temporarily, portable applications should call `endwin()`. Subsequently, to return to Curses mode, they should call `doupdate()`, `refresh()` or `wrefresh()`.

SEE ALSO

`curses`(3X), `delscreen`(3X), `doupdate`(3X), `initscr`(3X), `isendwin`(3X), `curses`(5).

Character User Interface Reference Manual

erasechar(3X)

NAME

`erasechar, erasewchar, killchar, killwchar -` terminal environment query functions

SYNOPSIS

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

```
#include <curses.h>
char erasechar(void);
int erasewchar(wchar_t *ch);
char killchar(void);
int killwchar(wchar_t *ch);
```

DESCRIPTION

The `erasechar()` function returns the current erase character. The `erasewchar()` function stores the current erase character in the object pointed to by *ch*. If no erase character has been defined, the function will fail and the object pointed to by *ch* will not be changed.

The `killchar()` function returns the current line kill character. The `killwchar()` function stores the current line kill character in the object pointed to by *ch*. If no line kill character has been defined, the function will fail and the object pointed to by *ch* will not be changed.

RETURN VALUE

The `erasechar()` function returns the erase character and `killchar()` returns the line kill character. The return value is unspecified when these characters are multi-byte characters.

Upon successful completion, `erasewchar()` and `killwchar()` return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

The `erasechar()` and `killchar()` functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the `A_` prefix. Moreover, they do not reliably indicate cases in which the erase or line kill character, respectively, has not been defined. The `erasewchar()` and `killwchar()` functions overcome these limitations.

SEE ALSO

`tcgetattr`(2), `clearok`(3X), `curses`(3X), `delscreen`(3X), `curses`(5).

Character User Interface Reference Manual

filter(3X)

NAME

`filter -` disable use of certain terminal capabilities

SYNOPSIS

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

```
#include <curses.h>
void filter(void);
```

DESCRIPTION

The `filter()` function changes the algorithm for initialising terminal capabilities [see `terminfo(4)`] that assume that the terminal has more than one line. A subsequent call to `initscr()` or `newterm()` performs the following additional actions:

- Disable use of `clear`, `cud`, `cud1`, `cup`, `cuu1` and `vpa`.
- Set the value of the `home` string to the value of the `cr` string.
- Set `lines` equal to 1.

Any call to `filter()` must precede the call to `initscr()` or `newterm()`.

RETURN VALUE

The `filter()` function does not return a value.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)`, `initscr(3X)`, `terminfo(4)`, `curses(5)`.

Character User Interface Reference Manual

flash(3X)

NAME

`flash -` flash the screen

SYNOPSIS

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

```
#include <curses.h>
int flash(void);
```

DESCRIPTION

The `flash()` function alerts the user. It flashes the screen, or if that is not possible, it sounds the audible alarm on the terminal. If neither signal is possible, nothing happens.

RETURN VALUE

The `flash()` function always returns `OK`.

ERRORS

No errors are defined.

NOTES

Nearly all terminals have an audible alarm, but only some can flash the screen.

SEE ALSO

`beep`(3X) , `curses`(3X) , `curses`(5) .

Character User Interface Reference Manual

flushinp(3X)

NAME

`flushinp` - discard input

SYNOPSIS

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

```
#include <curses.h>
int flushinp(void);
```

DESCRIPTION

The `flushinp()` function discards (flushes) any characters in the input buffer associated with the current screen.

RETURN VALUE

The `flushinp()` function always returns `OK`.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X), `curses`(5).

Character User Interface Reference Manual

getbegyx(3X)

NAME

getbegyx, getmaxyx, getparyx, getyx - get cursorand window coordinates

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
void getbegyx(WINDOW *win, int y, int x);
void getmaxyx(WINDOW *win, int y, int x);
void getparyx(WINDOW *win, int y, int x);
void getyx(WINDOW *win, int y, int x);
```

DESCRIPTION

The `getyx()` macro stores the cursor position of the specified window in *y* and *x*.

The `getparyx()` macro, if the specified window is a subwindow, stores in *y* and *x* the coordinates of the window's origin relative to its parent window. Otherwise, – is stored in *y* and *x*.

The `getbegyx()` macro stores the absolute screen coordinates of the specified window's origin in *y* and *x*.

The `getmaxyx()` macro stores the number of rows of the specified window in *y* and stores the window's number of columns in *x*.

RETURN VALUE

No return values are defined.

ERRORS

No errors are defined.

NOTES

These interfaces are macros and "`&`" cannot be used before the *y* and *x* arguments.

Traditional implementations have often defined the following macros:

```
void getbegx(WINDOW *win, int x);
void getbegy(WINDOW *win, int y);
void getmaxx(WINDOW *win, int x);
void getmaxy(WINDOW *win, int y);
void getparx(WINDOW *win, int x);
void getpary(WINDOW *win, int y);
```

Although `getbegyx()`, `getmaxyx()` and `getparyx()` provide the required functionality, this does not preclude applications from defining these macros for their own use. For example, to implement

```
void getbegx(WINDOW *win, int x);
```

the macro would be

```
#define getbegx(_win,_x)    \
{                 \
  int _y;           \
 \
  getbegyx(_win,_y,_x);   \
}
```

SEE ALSO

`curses(3X)` , `curses(5)` .

Character User Interface Reference Manual

getcchar(3X)

NAME

`getcchar -` get a wide character string and rendition from acchar_t

SYNOPSIS

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

`#include <curses.h>`

`int getcchar(const cchar_t *` *wcval*, wchar_t **wch*, attr_t **attrs*,

`short *`*color_pair*, void **opts*);

DESCRIPTION

When *wch* is not a null pointer, the `getcchar()` function extracts information from a `cchar_t` defined by *wcval*, stores the character attributes in the object pointed to by *attrs*, stores the colour pair in the object pointed to by *color_pair*, and stores the wide character string referenced by *wcval* into the array pointed to by *wch*.

When *wch* is a null pointer, `getcchar()` obtains the number of wide characters in the object pointed to by *wcval* and does not change the objects pointed to by *attrs* or *color_pair*.

The *opts* argument is reserved for definition in a future edition of this document. Currently, the application must provide a null pointer as *opts*.

RETURN VALUE

When *wch* is a null pointer, `getcchar()` returns the number of wide characters referenced by *wcval*, including the null terminator.

When *wch* is not a null pointer, `getcchar()` returns `OK` upon successful completion, and `ERR` otherwise.

ERRORS

No errors are defined.

NOTES

The *wcval* argument may be a value generated by a call to `setcchar()` or by a function that has a `cchar_t` output argument. If *wcval* is constructed by any other means, the effect is unspecified.

SEE ALSO

`attroff`(3X), `can_change_color`(3X), `curses`(3X), `setcchar`(3X), `curses`(5).

Character User Interface Reference Manual

getch(3X)

NAME

`getch, wgetch, mvgetch, mvwgetch -` get asingle-byte character from the terminal

SYNOPSIS

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

```
#include <curses.h>
int getch(void);
int mvgetch(int y, int x);
int mvwgetch(WINDOW *win, int y, int x);
int wgetch(WINDOW *win);
```

DESCRIPTION

These functions read a single-byte character from the terminal associated with the current or specified window. The results are unspecified if the input is not a single-byte character. If `keypad()` is enabled, these functions respond to the pressing of a function key by returning the corresponding `KEY_` value defined in `<curses.h>`.

Processing of terminal input is subject to the general rules described in "X/Open Curses, Issue 4, Version 2, Section 3.5".

If echoing is enabled, then the character is echoed as though it were provided as an input argument to `addch()`, except for the following characters:

<backspace>, <left-arrow> and the current erase character:
: The input is interpreted as specified in "X/Open Curses, Issue 4, Version 2, Section 3.4.3" and then the character at the resulting cursor position is deleted as though `delch()` were called, except that if the cursor was originally in the first column of the line, then the user is alerted as though `beep()` were called.

Function keys:
: The user is alerted as though `beep()` were called. Information concerning the function keys is not returned to the caller.

If the current or specified window is not a pad, and it has been moved or modified since the last refresh operation, then it will be refreshed before another character is read.

RETURN VALUE

Upon successful completion `getch()`, `mvgetch()`, `mvwgetch()` and `wgetch()` return the single-byte character, `KEY_` value; otherwise `ERR` is returned. When in the *nodelay* mode and no data is available, `ERR` is returned.

ERRORS

No errors are defined.

NOTES

Applications should not define the escape key by itself as a single-character function.

When using these functions, *nocbreak* mode [`nocbreak()`] and *echo* mode [`echo()`] should not be used at the same time. Depending on the state of the terminal when each character is typed, the program may produce undesirable results.

SEE ALSO

`cbreak`(3X), `curses`(3X), `doupdate`(3X), `insch`(3X), `curses`(5).

Character User Interface Reference Manual

getnstr(3X)

NAME

`getnstr, getstr, mvgetnstr, mvgetstr, mvwgetnstr, mvwgetstr, wgetstr, wgetnstr` - get amulti-byte character string from the terminal

SYNOPSIS

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

```
#include <curses.h>
int getnstr(char *str, int n);
int getstr(char *str);
int mvgetnstr(int y, int x, char *str, int n);
int mvgetstr(int y, int x, char *str);
int mvwgetnstr(WINDOW *win, int y, int x, char *str, int n);
int mvwgetstr(WINDOW *win, int y, int x, char *str);
int wgetnstr(WINDOW *win, char *str, int n);
int wgetstr(WINDOW *win, char *str);
```

DESCRIPTION

The effect of `getstr()` is as though a series of calls to `getch()` were made, until a newline, carriage return or end-of-file is received. The resulting value is placed in the area pointed to by *str*. The string is then terminated with a null byte. The `getnstr()`, `mvgetnstr()`, `mvwgetnstr()` and `wgetnstr()` functions read at most *n* bytes, thus preventing a possible overflow of the input buffer. The user's erase and kill characters are interpreted, as well as any special keys (such as function keys, home key, clear key, and so on).

The `mvgetstr()` function is identical to `getstr()` except that it is as though it is a call to `move()` and then a series of calls to `getch()`. The `mvwgetstr()` function is identical to `getstr()` except it is as though a call to `wmove()` is made and then a series of calls to `wgetch()`. The `mvgetnstr()` function is identical to `getnstr()` except that it is as though it is a call to `move()` and then a series of calls to `getch()`. The `mvwgetnstr()` function is identical to `getnstr()` except it is as though a call to `wmove()` is made and then a series of calls to `wgetch()`.

The `getnstr()`, `wgetnstr()`, `mvgetnstr()` and `mvwgetnstr()` functions will only return the entire multi-byte sequence associated with a character. If the array is large enough to contain at least one character, the functions fill the array with complete characters. If the array is not large enough to contain any complete characters, the function fails.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

Reading a line that overflows the array pointed to by *str* with `getstr()`, `mvgetstr()`, `mvwgetstr()` or `wgetstr()` causes undefined results. The use of `getnstr()`, `mvgetnstr()`, `mvwgetnstr()` or `wgetnstr()`, respectively, is recommended.

SEE ALSO

`beep`(3X), `curses`(3X), `getch`(3X), `curses`(5).

Character User Interface Reference Manual

getn_wstr(3X)

NAME

`getn_wstr, get_wstr, mvgetn_wstr, mvget_wstr, mvwgetn_wstr, mvwget_wstr, wgetn_wstr, wget_wstr -` get an array of wide characters and function key codesfrom a terminal

SYNOPSIS

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

```
#include <curses.h>
int getn_wstr(wchar_t *wstr, int n);
int get_wstr(wchar_t *wstr);
int mvgetn_wstr(int y, int x, wchar_t *wstr, int n);
int mvget_wstr(int y, int x, wchar_t *wstr);
int mvwgetn_wstr(WINDOW *win, int y, int x, wchar_t *wstr, int n);
int mvwget_wstr(WINDOW *win, int y, int x, wchar_t *wstr);
int wgetn_wstr(WINDOW *win, wchar_t *wstr, int n);
int wget_wstr(WINDOW *win, wchar_t *wstr);
```

DESCRIPTION

The effect of `get_wstr()` is as though a series of calls to `get_wch()` were made, until a newline character, end-of-line character, or end-of-file character is processed. An end-of-file character is represented by `WEOF`, as defined in `<wchar.h>`. A newline or end-of-line is represented as its `wchar_t` value. In all instances, the end of the string is terminated by a null `wchar_t`. The resulting values are placed in the area pointed to by *wstr*.

The user's erase and kill characters are interpreted and affect the sequence of characters returned.

The effect of `wget_wstr()` is as though a series of calls to `wget_wch()` were made.

The effect of `mvget_wstr()` is as though a call to `move()` and then a series of calls to `get_wch()` were made. The effect of `mvwget_wstr()` is as though a call to `wmove()` and then a series of calls to `wget_wch()` were made. The effect of `mvget_nwstr()` is as though a call to `move()` and then a series of calls to `get_wch()` were made. The effect of `mvwget_nwstr()` is as though a call to `wmove()` and then a series of calls to `wget_wch()` were made.

The `getn_wstr()`, `mvgetn_wstr()`, `mvwgetn_wstr()` and `wgetn_wstr()` functions read at most *n* characters, letting the application prevent overflow of the input buffer.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

Reading a line that overflows the array pointed to by *wstr* with `get_wstr()`, `mvget_wstr()`, `mvwget_wstr()` or `wget_wstr()` causes undefined results. The use of `getn_wstr()`, `mvgetn_wstr()`, `mvwgetn_wstr()` or `wgetn_wstr()`, respectively, is recommended.

These functions cannot return `KEY_` values as there is no way to distinguish a `KEY_` value from a valid `wchar_t` value.

SEE ALSO

`curses`(3X) , `get_wch`(3X) , `getstr`(3X) , `curses`(5) , `wchar`(5) .

Character User Interface Reference Manual

getwin(3X)

NAME

`getwin, putwin -` dump window to, and reload window from, a file

SYNOPSIS

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

```
#include <curses.h>
WINDOW *getwin(FILE *filep);
int putwin(WINDOW *win, FILE *filep);
```

DESCRIPTION

The `getwin()` function reads window-related data stored in the file by `putwin()`. The function then creates and initialises a new window using that data.

The `putwin()` function writes all data associated with *win* into the `stdio` stream to which *filep* points, using an unspecified format. This information can be retrieved later using `getwin()`.

RETURN VALUE

Upon successful completion, `getwin()` returns a pointer to the window it created. Otherwise, it returns a null pointer.

Upon successful completion, `putwin()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)`, `scr_dump(3X)`, `curses(5)`.

Character User Interface Reference Manual

get_wch(3X)

NAME

get_wch, mvget_wch, mvwget_wch, wget_wch - get awide character from a terminal

SYNOPSIS

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

#include <curses.h>

int get_wch(wint_t *ch);

int mvget_wch(int *y*, int *x*, wint_t *ch);

int mvwget_wch(WINDOW *win, int *y*, int *x*, wint_t *ch);

int wget_wch(WINDOW *win, wint_t *ch);

DESCRIPTION

These functions read a character from the terminal associated with the current or specified window. If keypad() is enabled, these functions respond to the pressing of a function key by setting the object pointed to by *ch* to the corresponding KEY_ value defined in <curses.h> [see curses(5)] and returning KEY_CODE_YES.

Processing of terminal input is subject to the general rules described in "X/Open Curses, Issue 4, Version 2, Section 3.5".

If echoing is enabled, then the character is echoed as though it were provided as an input argument to add_wch(), except for the following characters:

<backspace>, <left-arrow> and the current erase character:

The input is interpreted as specified in "X/Open Curses, Issue 4, Version 2, Section 3.4.3" and then the character at the resulting cursor position is deleted as though delch() were called, except that if the cursor was originally in the first column of the line, then the user is alerted as though beep() were called.

Function keys:

The user is alerted as though beep() were called. Information concerning the function keys is not returned to the caller.

If the current or specified window is not a pad, and it has been moved or modified since the last refresh operation, then it will be refreshed before another character is read.

RETURN VALUE

When these functions successfully report the pressing of a function key, they return KEY_CODE_YES. When they successfully report a wide character, they return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

NOTES

Applications should not define the escape key by itself as a single-character function.

When using these functions, nocbreak() mode and echo() mode should not be used at the same time. Depending on the state of the terminal when each character is typed, the application may produce undesirable results.

SEE ALSO

beep(3X) , cbreak(3X) , curses(3X) , ins_wch(3X) , keypad(3X) , move(3X) , curses(5) , wchar(5) .

Character User Interface Reference Manual

halfdelay(3X)

NAME

`halfdelay -` control input character delay mode

SYNOPSIS

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

```
#include <curses.h>
int halfdelay(int tenths);
```

DESCRIPTION

The `halfdelay()` function sets the input mode for the current window to Half-Delay Mode and specifies *tenths* tenths of seconds as the half-delay interval. The *tenths* argument must be in a range from 1 up to and including 255.

RETURN VALUE

Upon successful completion, `halfdelay()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

NOTES

The application can call `nocbreak()` to leave Half-Delay mode.

SEE ALSO

`cbreak`(3X), `curses`(3X), `curses`(5).

Character User Interface Reference Manual

has_ic(3X)

NAME

has_ic, has_il - query functions for terminal insert and deletecapability

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
bool has_ic(void);
bool has_il(void);
```

DESCRIPTION

The has_ic() function indicates whether the terminal has insert- and delete-character capabilities.

The has_il() function indicates whether the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions.

RETURN VALUE

The has_ic() function returns TRUE if the terminal has insert- and delete-character capabilities. Otherwise, it returns FALSE.

The has_il() function returns TRUE if the terminal has insert- and delete-line capabilities. Otherwise, it returns FALSE.

ERRORS

No errors are defined.

NOTES

The has_il() function may be used to determine if it would be appropriate to turn on physical scrolling using scrollok().

SEE ALSO

curses(3X), curses(5).

Character User Interface Reference Manual

hline(3X)

NAME

`hline, mvhline, mvvline, mvwhline, mvwvline, vline, whline, wvline -` draw lines from single-bytecharacters and renditions

SYNOPSIS

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

`#include <curses.h>`

`int hline(chtype` *ch*, int *n*);

`int mvhline(int` *y*, int *x*, chtype *ch*, int *n*);

`int mvvline(int` *y*, int *x*, chtype *ch*, int *n*);

`int mvwhline(WINDOW *`*win*, int *y*, int *x*, chtype *ch*, int *n*);

`int mvwvline(WINDOW *`*win*, int *y*, int *x*, chtype *ch*, int *n*);

`int vline(chtype` *ch*, int *n*);

`int whline(WINDOW *`*win*, chtype *ch*, int *n*);

`int wvline(WINDOW *`*win*, chtype *ch*, int *n*);

DESCRIPTION

These functions draw a line in the current or specified window starting at the current or specified position, using *ch*. The line is at most *n* positions long, or as many as fit into the window.

These functions do not advance the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

The `hline()`, `mvhline()`, `mvwhline()` and `whline()` functions draw a line proceeding toward the last column of the same line.

The `vline()`, `mvvline()`, `mvwvline()` and `wvline()` functions draw a line proceeding toward the last line of the window.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the `A_` prefix.

SEE ALSO

`border`(3X), `box`(3X), `curses`(3X), `hline_set`(3X), `curses`(5).

hline_set(3X)

NAME

`hline_set, mvhline_set, mvvline_set, mvwhline_set, mvwvline_set, vline_set, whline_set, wvline_set -` draw lines from complex characters and renditions

SYNOPSIS

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

```
#include <curses.h>
int hline_set(const cchar_t * wch, int n);
int mvhline_set(int  y, int x, const cchar_t *wch, int n);
int mvvline_set(int  y, int x, const cchar_t *wch, int n);
int mvwhline_set(WINDOW * win, int y, int x, const cchar_t *wch, int n);
int mvwvline_set(WINDOW * win, int y, int x, const cchar_t *wch, int n);
int vline_set(const cchar_t * wch, int n);
int whline_set(WINDOW * win, const cchar_t *wch, int n);
int wvline_set(WINDOW * win, const cchar_t *wch, int n);
```

DESCRIPTION

These functions draw a line in the current or specified window starting at the current or specified position, using *ch*. The line is at most *n* positions long, or as many as fit into the window.

These functions do not advance the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

The `hline_set()`, `mvhline_set()`, `mvwhline_set()` and `whline_set()` functions draw a line proceeding toward the last column of the same line.

The `vline_set()`, `mvvline_set()`, `mvwvline_set()` and `wvline_set()` functions draw a line proceeding toward the last line of the window.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`border_set(3X)`, `curses(3X)`, `curses(5)`.

idcok(3X) - longname(3X)

idcok(3X)

NAME

`idcok -` enable or disable use of hardware insert- and delete-characterfeatures

SYNOPSIS

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

```
#include <curses.h>
```

`void idcok(WINDOW *`*win*, bool *bf*);

DESCRIPTION

The `idcok()` function specifies whether the implementation may use hardware insert- and delete-character features in *win* if the terminal is so equipped. If *bf* is `TRUE`, use of these features in *win* is enabled. If *bf* is `FALSE`, use of these features in *win* is disabled. The initial state is `TRUE`.

RETURN VALUE

The `idcok()` function does not return a value.

ERRORS

No errors are defined.

SEE ALSO

`clearok`(3X), `curses`(3X), `doupdate`(3X), `curses`(5).

Character User Interface Reference Manual

immedok(3X)

NAME

`immedok` - enable or disable immediate terminal refresh

SYNOPSIS

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

```
#include <curses.h>
void immedok(WINDOW *win, bool bf);
```

DESCRIPTION

The `immedok()` function specifies whether the screen is refreshed whenever the window pointed to by *win* is changed. If *bf* is `TRUE`, the window is implicitly refreshed on each such change. If *bf* is `FALSE`, the window is not implicitly refreshed. The initial state is `FALSE`.

RETURN VALUE

The `immedok()` function does not return a value.

ERRORS

No errors are defined.

NOTES

The `immedok()` function is useful for windows that are used as terminal emulators.

SEE ALSO

`clearok`(3X), `curses`(3X), `doupdate`(3X), `curses`(5).

Character User Interface Reference Manual

inch(3X)

NAME

`inch, mvinch, mvwinch, winch -` input a single-bytecharacter and rendition from a window

SYNOPSIS

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

```
#include <curses.h>
chtype inch(void);
chtype mvinch(int y, int x);
chtype mvwinch(WINDOW *win, int y, int x);
chtype winch(WINDOW *win);
```

DESCRIPTION

These functions return the character and rendition, of type `chtype`, at the current or specified position in the current or specified window.

RETURN VALUE

Upon successful completion, the functions return the specified character and rendition. Otherwise, they return `(chtype)ERR`.

ERRORS

No errors are defined.

NOTES

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the `A_` prefix.

SEE ALSO

`curses(3X)` , `curses(5)` .

Character User Interface Reference Manual

inchnstr(3X)

NAME

`inchnstr, inchstr, mvinchnstr, mvinchstr, mvwinchnstr, mvwinchstr, winchnstr, winchstr -` input an array of single-byte characters and renditionsfrom a window

SYNOPSIS

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

```
#include <curses.h>
int inchnstr(chtype *chstr, int n);
int inchstr(chtype *chstr);
int mvinchnstr(int y, int x, chtype *chstr, int n);
int mvinchstr(int y, int x, chtype *chstr);
int mvwinchnstr(WINDOW *win, int y, int x, chtype *chstr, int n);
int mvwinchstr(WINDOW *win, int y, int x, chtype *chstr);
int winchnstr(WINDOW *win, chtype *chstr, int n);
int winchstr(WINDOW *win, chtype *chstr);
```

DESCRIPTION

These functions place characters and renditions from the current or specified window into the array pointed to by *chstr*, starting at the current or specified position and ending at the end of the line.

The `inchnstr()`, `mvinchnstr()`, `mvwinchnstr()` and `winchnstr()` functions store at most *n* elements from the current or specified window into the array pointed to by *chstr*.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

Reading a line that overflows the array pointed to by *chstr* with `inchstr()`, `mvinchstr()`, `mvwinchstr()` or `winchstr()` causes undefined results. The use of `inchnstr()`, `mvinchnstr()`, `mvwinchnstr()` or `winchnstr()`, respectively, is recommended.

SEE ALSO

`curses(3X)`, `inch(3X)`, `curses(5)`.

Character User Interface Reference Manual

initscr(3X)

NAME

initscr, newterm - screen initialisation functions

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
WINDOW *initscr(void);
SCREEN *newterm(char *type, FILE *outfile, FILE *infile);
```

DESCRIPTION

The initscr() function determines the terminal type and initialises all implementation data structures. The TERM environment variable specifies the terminal type. The initscr() function also causes the first refresh operation to clear the screen. If errors occur, initscr() writes an appropriate error message to standard error and exits. The only functions that can be called before initscr() or newterm() are filter(), ripoffline(), slk_init(), use_env() and the functions whose prototypes are defined in <term.h>. Portable applications must not call initscr() twice.

The newterm() function can be called as many times as desired to attach a terminal device. The *type* argument points to a string specifying the terminal type, except that if *type* is a null pointer, the TERM environment variable is used. The *outfile* and *infile* arguments are file pointers for output to the terminal and input from the terminal, respectively. It is unspecified whether Curses modifies the buffering mode of these file pointers. The newterm() function should be called once for each terminal.

The initscr() function is equivalent to:

```
newterm(getenv("TERM"), stdout, stdin); return stdscr;
```

If the current disposition for the signals SIGINT, SIGQUIT or SIGTSTP is SIGDFL, then initscr() may also install a handler for the signal, which may remain in effect for the life of the process or until the process changes the disposition of the signal.

The initscr() and newterm() functions initialise the cur_term external variable.

RETURN VALUE

Upon successful completion, initscr() returns a pointer to stdscr. Otherwise, it does not return.

Upon successful completion, newterm() returns a pointer to the specified terminal. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

NOTES

A program that outputs to more than one terminal should use newterm() for each terminal instead of initscr(). A program that needs an indication of error conditions, so it can continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, would also use this function.

Applications should perform any required handling of the SIGINT, SIGQUIT or SIGTSTP signals before calling initscr().

SEE ALSO

curses(3X), delscreen(3X), doupdate(3X), del_curterm(3X), filter(3X), slk_attroff(3X), use_env(3X), curses(5), term(5).

Character User Interface Reference Manual

innstr(3X)

NAME

`innstr, instr, mvinnstr, mvinstr, mvwinnstr, mvwinstr, winnstr, winstr -` input a multi-byte characterstring from a window

SYNOPSIS

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

```
#include <curses.h>
int innstr(char *str, int n);
int instr(char *str);
int mvinnstr(int y, int x, char *str, int n);
int mvinstr(int y, int x, char *str);
int mvwinnstr(WINDOW *win, int y, int x, char *str, int n);
int mvwinstr(WINDOW *win, int y, int x, char *str);
int winnstr(WINDOW *win, char *str, int n);
int winstr(WINDOW *win, char *str);
```

DESCRIPTION

These functions place a string of characters from the current or specified window into the array pointed to by *str*, starting at the current or specified position and ending at the end of the line.

The `innstr()`, `mvinnstr()`, `mvwinnstr()` and `winnstr()` functions store at most *n* bytes in the string pointed to by *str*.

The `innstr()`, `mvinnstr()`, `mvwinnstr()` and `winnstr()` functions will only store the entire multi-byte sequence associated with a character. If the array is large enough to contain at least one character the array is filled with complete characters. If the array is not large enough to contain any complete characters, the function fails.

RETURN VALUE

Upon successful completion, `instr()`, `mvinstr()`, `mvwinstr()` and `winstr()` return `OK`.

Upon successful completion, `innstr()`, `mvinnstr()`, `mvwinnstr()` and `winnstr()` return the number of characters actually read into the string.

Otherwise, all these functions return `ERR`.

ERRORS

No errors are defined.

NOTES

Since multi-byte characters may be processed, there might not be a one-to-one correspondence between the number of column positions on the screen and the number of bytes returned.

These functions do not return rendition information.

Reading a line that overflows the array pointed to by *str* with `instr()`, `mvinstr()`, `mvwinstr()` or `winstr()` causes undefined results. The use of `innstr()`, `mvinnstr()`, `mvwinnstr()` or `winnstr()`, respectively, is recommended.

SEE ALSO

`curses(3X)` , `curses(5)` .

Character User Interface Reference Manual

innwstr(3X)

NAME

innwstr, inwstr, mvinnwstr, mvinwstr, mvwinnwstr, mvwinwstr, winnwstr, winwstr - input astring of wide characters from a window

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
int innwstr(wchar_t *wstr, int n);
int inwstr(wchar_t *wstr);
int mvinnwstr(int y, int x, wchar_t *wstr, int n);
int mvinwstr(int y, int x, wchar_t *wstr);
int mvwinnwstr(WINDOW *win, int y, int x, wchar_t *wstr, int n);
int mvwinwstr(WINDOW *win, int y, int x, wchar_t *wstr);
int winnwstr(WINDOW *win, wchar_t *wstr, int n);
int winwstr(WINDOW *win, wchar_t *wstr);
```

DESCRIPTION

These functions place a string of `wchar_t` characters from the current or specified window into the array pointed to by *wstr* starting at the current or specified cursor position and ending at the end of the line.

These functions will only store the entire wide character sequence associated with a spacing complex character. If the array is large enough to contain at least one complete spacing complex character, the array is filled with complete characters. If the array is not large enough to contain any complete characters this is an error.

The `innwstr()`, `mvinnwstr()`, `mvwinnwstr()` and `winnwstr()` functions store at most *n* characters in the array pointed to by *wstr*.

RETURN VALUE

Upon successful completion, `inwstr()`, `mvinwstr()`, `mvwinwstr()` and `winwstr()` return `OK`.

Upon successful completion, `innwstr()`, `mvinnwstr()`, `mvwinnwstr()` and `winnwstr()` return the number of characters actually read into the string.

Otherwise, all these functions return `ERR`.

ERRORS

No errors are defined.

NOTES

Reading a line that overflows the array pointed to by *wstr* with `inwstr()`, `mvinwstr()`, `mvwinwstr()` or `winwstr()` causes undefined results. The use of `innwstr()`, `mvinnwstr()`, `mvwinnwstr()` or `winnwstr()`, respectively, is recommended.

These functions do not return rendition information.

SEE ALSO

`curses(3X)` , `curses(5)` .

Character User Interface Reference Manual

insch(3X)

NAME

`insch, mvinsch, mvwinsch, winsch -` insert asingle-byte character and rendition into a window

SYNOPSIS

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

```
#include <curses.h>
int insch(chtype ch);
int mvinsch(int y, int x, chtype ch);
int mvwinsch(WINDOW *win, int y, int x, chtype ch);
int winsch(WINDOW *win, chtype ch);
```

DESCRIPTION

These functions insert the character and rendition from *ch* into the current or specified window at the current or specified position.

These functions do not perform wrapping. These functions do not advance the cursor position. These functions perform special-character processing, with the exception that if a newline is inserted into the last line of a window and scrolling is not enabled, the behavior is unspecified.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the `A_` prefix.

SEE ALSO

`curses(3X)` , `ins_wch(3X)` , `curses(5)` .

Character User Interface Reference Manual

insdelln(3X)

NAME

`insdelln, winsdelln -` delete or insert lines into a window

SYNOPSIS

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

```
#include <curses.h>
int insdelln(int n);
int winsdelln(WINDOW *win, int n);
```

DESCRIPTION

The `insdelln()` and `winsdelln()` functions perform the following actions:

- If *n* is positive, these functions insert *n* lines into the current or specified window before the current line. The *n* last lines are no longer displayed.
- If *n* is negative, these functions delete *n* lines from the current or specified window starting with the current line, and move the remaining lines toward the cursor. The last *n* lines are cleared.

The current cursor position remains the same.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X), `deleteln`(3X), `insertln`(3X), `curses`(5).

Character User Interface Reference Manual

insertln(3X)

NAME

`insertln, winsertln -` insert lines into a window

SYNOPSIS

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

```
#include <curses.h>
int insertln(void);
int winsertln(WINDOW *win);
```

DESCRIPTION

The `insertln()` and `winsertln()` functions insert a blank line before the current line in the current or specified window. The bottom line is no longer displayed. The cursor position does not change.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X), `insdelln`(3X), `curses`(5).

Character User Interface Reference Manual

insnstr(3X)

NAME

`insnstr, insstr, mvinsnstr, mvinsstr, mvwinsnstr, mvwinsstr, winsnstr, winsstr` - inserta multi-byte character string into a window

SYNOPSIS

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

```
#include <curses.h>
int insnstr(const char *str, int n);
int insstr(const char *str);
int mvinsnstr(int y, int x, const char *str, int n);
int mvinsstr(int y, int x, const char *str);
int mvwinsnstr(WINDOW *win, int y, int x, const char *str, int n);
int mvwinsstr(WINDOW *win, int y, int x, const char *str);
int winsnstr(WINDOW *win, const char *str, int n);
int winsstr(WINDOW *win, const char *str);
```

DESCRIPTION

These functions insert a character string (as many characters as will fit on the line) before the current or specified position in the current or specified window.

These functions do not advance the cursor position. These functions perform special-character processing. The `insnstr()` and `winsnstr()` functions perform wrapping. The `insstr()` and `winsstr()` functions do not perform wrapping.

The `insnstr()`, `mvinsnstr()`, `mvwinsnstr()` and `winsnstr()` functions insert at most *n* bytes. If *n* is less than 1, the entire string is inserted.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

Since the string may contain multi-byte characters, there might not be a one-to-one correspondence between the number of column positions occupied by the characters and the number of bytes in the string.

SEE ALSO

`curses(3X)` , `curses(5)` .

Character User Interface Reference Manual

ins_nwstr(3X)

NAME

`ins_nwstr, ins_wstr, mvins_nwstr, mvins_wstr, mvwins_nwstr, mvwins_wstr, wins_nwstr, wins_wstr -` insert a wide-character string into a window

SYNOPSIS

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

```
#include <curses.h>
int ins_nwstr(const wchar_t *wstr, int n);
int ins_wstr(const wchar_t *wstr);
int mvins_nwstr(int y, int x, const wchar_t *wstr, int n);
int mvins_wstr(int y, int x, const wchar_t *wstr);
int mvwins_nwstr(WINDOW *win, int y, int x, const wchar_t *wstr, int n);
int mvwins_wstr(WINDOW *win, int y, int x, const wchar_t *wstr);
int wins_nwstr(WINDOW *win, const wchar_t *wstr, int n);
int wins_wstr(WINDOW *win, const wchar_t *wstr);
```

DESCRIPTION

These functions insert a `wchar_t` character string (as many `wchar_t` characters as will fit on the line) in the current or specified window immediately before the current or specified position.

Any non-spacing characters in the string are associated with the first spacing character in the string that precedes the non-spacing characters. If the first character in the string is a non-spacing character, these functions will fail.

These functions do not perform wrapping. These functions do not advance the cursor position. These functions perform special-character processing.

The `ins_nwstr()`, `mvins_nwstr()`, `mvwins_nwstr()` and `wins_nwstr()` functions insert at most *n* `wchar_t` characters. If *n* is less than 1, then the entire string is inserted.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)`, `curses(5)`.

Character User Interface Reference Manual

ins_wch(3X)

NAME

`ins_wch, mvins_wch, mvwins_wch, wins_wch -` inserta complex character and rendition into a window

SYNOPSIS

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

```
#include <curses.h>
int ins_wch(const cchar_t * wch);
int wins_wch(WINDOW * win, const cchar_t *wch);
int mvins_wch(int  y, int x, const cchar_t *wch);
int mvwins_wch(WINDOW * win, int y, int x, const cchar_t *wch);
```

DESCRIPTION

These functions insert the complex character *wch* with its rendition in the current or specified window at the current or specified cursor position.

These functions do not perform wrapping. These functions do not advance the cursor position. These functions perform special-character processing, with the exception that if a newline is inserted into the last line of a window and scrolling is not enabled, the behavior is unspecified.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

For non-spacing characters, `add_wch()` can be used to add the non-spacing characters to a spacing complex character already in the window.

SEE ALSO

`add_wch`(3X) , `curses`(3X) , `curses`(5) .

Character User Interface Reference Manual

intrflush(3X)

NAME

`intrflush -` enable or disable flush on interrupt

SYNOPSIS

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

```
#include <curses.h>
int intrflush(WINDOW *win, bool bf);
```

DESCRIPTION

The `intrflush()` function specifies whether pressing an interrupt key (interrupt, suspend or quit) will flush the input buffer associated with the current screen. If the value of *bf* is `TRUE`, then flushing of the output buffer associated with the current screen will occur when an interrupt key (interrupt, suspend, or quit) is pressed. If the value of *bf* is `FALSE` then no flushing of the buffer will occur when an interrupt key is pressed. The default for the option is inherited from the display driver settings. The *win* argument is ignored.

RETURN VALUE

Upon successful completion, `intrflush()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

NOTES

The same effect is achieved outside Curses using the `NOFLSH` local mode flag specified in the XBD specification ("General Terminal Interface").

SEE ALSO

`curses(3X)` , `curses(5)` .

Character User Interface Reference Manual

in_wch(3X)

NAME

`in_wch, mvin_wch, mvwin_wch, win_wch -` extract acomplex character and rendition from a window

SYNOPSIS

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

```
#include <curses.h>
int in_wch(cchar_t * wcval);
int mvin_wch(int  y, int x, cchar_t *wcval);
int mvwin_wch(WINDOW * win, int y, int x, cchar_t *wcval);
int win_wch(WINDOW * win, cchar_t *wcval);
```

DESCRIPTION

These functions extract the complex character and rendition from the current or specified position in the current or specified window into the object pointed to by *wcval*.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)` , `curses(5)` .

Character User Interface Reference Manual

in_wchnstr(3X)

NAME

`in_wchnstr, in_wchstr, mvin_wchnstr, mvin_wchstr, mvwin_wchnstr, mvwin_wchstr, win_wchnstr, win_wchstr -` extract an array of complex characters and renditionsfrom a window

SYNOPSIS

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

```
#include <curses.h>
int in_wchnstr(cchar_t * wchstr, int n);
int in_wchstr(cchar_t * wchstr);
int mvin_wchnstr(int  y, int x, cchar_t *wchstr, int n);
int mvin_wchstr(int  y, int x, cchar_t *wchstr);
int mvwin_wchnstr(WINDOW * win, int y, int x, cchar_t *wchstr, int n);
int mvwin_wchstr(WINDOW * win, int y, int x, cchar_t *wchstr);
int win_wchnstr(WINDOW * win, cchar_t *wchstr, int n);
int win_wchstr(WINDOW * win, cchar_t *wchstr);
```

DESCRIPTION

These functions extract characters from the current or specified window, starting at the current or specified position and ending at the end of the line, and place them in the array pointed to by *wchstr*.

The `in_wchnstr()`, `mvin_wchnstr()`, `mvwin_wchnstr()` and `win_wchnstr()` fill the array with at most *n* `cchar_t` elements.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

Reading a line that overflows the array pointed to by *wchstr* with `in_wchstr()`, `mvin_wchstr()`, `mvwin_wchstr()` or `win_wchstr()` causes undefined results. The use of `in_wchnstr()`, `mvin_wchnstr()`, `mvwin_wchnstr()` or `win_wchnstr()`, respectively, is recommended.

SEE ALSO

`curses(3X)`, `in_wch(3X)`, `curses(5)`.

Character User Interface Reference Manual

isendwin(3X)

NAME

`isendwin -` determine whether a screen has been refreshed

SYNOPSIS

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

```
#include <curses.h>
bool isendwin(void);
```

DESCRIPTION

The `isendwin()` function indicates whether the screen has been refreshed since the last call to `endwin()`.

RETURN VALUE

The `isendwin()` function returns `TRUE` if `endwin()` has been called without any subsequent refresh. Otherwise, it returns `FALSE`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)`, `endwin(3X)`, `curses(5)`.

Character User Interface Reference Manual

is_linetouched(3X)

NAME

`is_linetouched, is_wintouched, touchline, touchwin, untouchwin, wtouchln -` window refresh control functions

SYNOPSIS

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

```
#include <curses.h>
bool is_linetouched(WINDOW *win, int line);
bool is_wintouched(WINDOW *win);
int touchline(WINDOW *win, int start, int count);
int touchwin(WINDOW *win);
int untouchwin(WINDOW *win);
int wtouchln(WINDOW *win, int y, int n, int changed);
```

DESCRIPTION

The `touchwin()` function touches the specified window (that is, marks it as having changed more recently than the last refresh operation). The `touchline()` function only touches *count* lines, beginning with line *start*.

The `untouchwin()` function marks all lines in the window as unchanged since the last refresh operation.

Calling `wtouchln()`, if *changed* is 1, touches *n* lines in the specified window, starting at line *y*. If *changed* is 0, `wtouchln()` marks such lines as unchanged since the last refresh operation.

The `is_wintouched()` function determines whether the specified window is touched. The `is_linetouched()` function determines whether line *line* of the specified window is touched.

RETURN VALUE

The `is_linetouched()` and `is_wintouched()` functions return `TRUE` if any of the specified lines, or the specified window, respectively, has been touched since the last refresh operation. Otherwise, they return `FALSE`.

Upon successful completion, the other functions return `OK`. Otherwise, they return `ERR`. Exceptions to this are noted in the preceding function descriptions.

ERRORS

No errors are defined.

NOTES

Calling `touchwin()` or `touchline()` is sometimes necessary when using overlapping windows, since a change to one window affects the other window, but the records of which lines have been changed in the other window do not reflect the change.

SEE ALSO

`curses(3X)`, `doupdate(3X)`, `curses(5)`.

Character User Interface Reference Manual

keyname(3X)

NAME

`keyname, key_name -` get name of key

SYNOPSIS

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

```
#include <curses.h>
char *keyname(int c);
char *key_name(wchar_t c);
```

DESCRIPTION

The `keyname()` and `key_name()` functions generate a character string whose value describes the key *c*. The *c* argument of `keyname()` can be an 8-bit character or a key code. The *c* argument of `key_name()` must be a wide character.

The string has a format according to the first applicable row in the following table:

Input	Format of Returned String
Visible character	The same character
Control character	^*X*
Meta-character (`keyname()` only)	M-*X*
Key value defined in `<curses.h>` (`keyname()` only)	KEY_*name*
None of the above	UNKNOWN KEY

The meta-character notation shown above is used only if meta-characters are enabled.

RETURN VALUE

Upon successful completion, `keyname()` returns a pointer to a string as described above. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

NOTES

The return value of `keyname()` and `key_name()` may point to a static area which is overwritten by a subsequent call to either of these functions.

Applications normally process meta-characters without storing them into a window. If an application stores meta-characters in a window and tries to retrieve them as wide characters, `keyname()` cannot detect meta-characters, since wide characters do not support meta-characters.

SEE ALSO

`curses(3X)`, `meta(3X)`, `curses(5)`.

Character User Interface Reference Manual

keypad(3X)

NAME

`keypad -` enable/disable abbreviation of function keys

SYNOPSIS

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

```
#include <curses.h>
int keypad(WINDOW *win, bool bf);
```

DESCRIPTION

The `keypad()` function controls keypad translation. If *bf* is `TRUE`, keypad translation is turned on. If *bf* is `FALSE`, keypad translation is turned off. The initial state is `FALSE`.

This function affects the behaviour of any function that provides keyboard input.

If the terminal in use requires a command to enable it to transmit distinctive codes when a function key is pressed, then after keypad translation is first enabled, the implementation transmits this command to the terminal before an affected input function tries to read any characters from that terminal.

RETURN VALUE

Upon successful completion, `keypad()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)` , `curses(5)` .

longname(3X)

NAME

`longname` - get verbose description of current terminal

SYNOPSIS

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

```
#include <curses.h>
char *longname(void);
```

DESCRIPTION

The `longname()` function generates a verbose description of the current terminal. The maximum length of a verbose description is 128 bytes. It is defined only after the call to `initscr()` or `newterm()`.

RETURN VALUE

Upon successful completion, `longname()` returns a pointer to the description specified above. Otherwise, it returns a null pointer on error.

ERRORS

No errors are defined.

NOTES

The return value of `longname()` may point to a static area which is overwritten by a subsequent call to `newterm()`.

SEE ALSO

`curses(3X)`, `initscr(3X)`, `curses(5)`.

meta(3X) - wunctrl(3X)

meta(3X)

NAME

`meta` - enable/disable meta-keys

SYNOPSIS

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

```
#include <curses.h>
int meta(WINDOW *win, bool bf);
```

DESCRIPTION

Initially, whether the terminal returns 7 or 8 significant bits on input depends on the control mode of the display driver (see the the XBD specification, "General Terminal Interface"). To force 8 bits to be returned, invoke `meta(win, TRUE)`. To force 7 bits to be returned, invoke `meta(win, FALSE)`. The *win* argument is always ignored. If the `terminfo(4)` capabilities `smm` (meta_on) and `rmm` (meta_off) are defined for the terminal, `smm` is sent to the terminal when `meta(win, TRUE)` is called and `rmm` is sent when `meta(win, FALSE)` is called.

RETURN VALUE

Upon successful completion, `meta()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

NOTES

The same effect is achieved outside Curses using the CS7 or CS8 control mode flag specified in the XBD specification ("General Terminal Interface").

The `meta()` function was designed for use with terminals with 7-bit character sets and a "meta" key that could be used to set the eighth bit.

SEE ALSO

`curses`(3X), `getch`(3X), `terminfo`(4), `curses`(5).

Character User Interface Reference Manual

move(3X)

NAME

`move, wmove -` window cursor location functions

SYNOPSIS

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

```
#include <curses.h>
int move(int y, int x);
int wmove(WINDOW *win, int y, int x);
```

DESCRIPTION

The `move()` and `wmove()` functions move the cursor associated with the current or specified window to (*y*, *x*) relative to the window's origin. This function does not move the terminal's cursor until the next refresh operation.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)` , `doupdate(3X)` , `curses(5)` .

Character User Interface Reference Manual

mv(3X)

NAME

mv - pointer page for functions with mv prefix

DESCRIPTION

Most cases in which a Curses function has the mv prefix indicate that the function takes *y* and *x* arguments and moves the cursor to that address as though move() were first called. (The corresponding functions without the mv prefix operate at the cursor position.)

The mvcur(), mvderwin() and mvwin() functions are exceptions to this rule, in that mv is not a prefix with the usual meaning and there are no corresponding functions without the mv prefix. These functions have entries under their own names.

In the mvprintw() and mvscanw() functions, mv is a prefix with the usual meaning, but the functions have entries under their own names because the mv function is the first function in the family of functions in alphabetical order.

The mv prefix is combined with a w prefix to produce Curses functions beginning with mvw.

The mv and mvw functions are discussed together with the corresponding functions that do not have these prefixes. They are found on the following entries:

Function		
mvaddch()	mvwaddch()	addch(3X)
mvaddchnstr()	mvwaddchnstr()	addchstr(3X)
mvaddchstr()	mvwaddchstr()	addchstr(3X)
mvaddnstr()	mvwaddnstr()	addnstr(3X)
mvaddstr()	mvwaddstr()	addnstr(3X)
mvaddnwstr()	mvwaddnwstr()	addnwstr(3X)
mvaddwstr()	mvwaddwstr()	addnwstr(3X)
mvadd_wch()	mvwadd_wch()	add_wch(3X)
mvadd_wchnstr()	mvwadd_wchnstr()	add_wchnstr(3X)
mvadd_wchstr()	mvwadd_wchstr()	add_wchnstr(3X)
mvchgat()	mvwchgat()	chgat(3X)
mvdelch()	mvwdelch()	delch(3X)
mvgetch()	mvwgetch()	getch(3X)
mvgetnstr()	mvwgetnstr()	getnstr(3X)
mvgetstr()	mvwgetstr()	getnstr(3X)
mvgetn_wstr()	mvwgetn_wstr()	getn_wstr(3X)
mvget_wch()	mvwget_wch()	get_wch(3X)
mvget_wstr()	mvwget_wstr()	getn_wstr(3X)
mvhline()	mvwhline()	hline(3X)
mvhline_set()	mvwhline_set()	hline_set(3X)
mvinch()	mvwinch()	inch(3X)
mvinchnstr()	mvwinchnstr()	inchnstr(3X)

mvinchstr()	mvwinchstr()	inchnstr(3X)
mvinnstr()	mvwinnstr()	innstr(3X)
mvinnwstr()	mvwinnwstr()	innwstr(3X)
mvinsch()	mvwinsch()	insch(3X)
mvinsnstr()	mvwinsnstr()	insnstr(3X)
mvinsstr()	mvwinsstr()	insnstr(3X)
mvinstr()	mvwinstr()	innstr(3X)
mvins_nwstr()	mvwins_nwstr()	ins_nwstr(3X)
mvins_wch()	mvwins_wch()	ins_wch(3X)
mvins_wstr()	mvwins_wstr()	ins_nwstr(3X)
mvinwstr()	mvwinwstr()	innwstr(3X)
mvin_wch()	mvwin_wch()	in_wch(3X)
mvin_wchnstr()	mvwin_wchnstr()	in_wchnstr(3X)
mvin_wchstr()	mvwin_wchstr()	in_wchnstr(3X)
mvprintw()	mvwprintw()	amvprintw(3X)
mvscanw()	mvwscanw()	mvscanw(3X)
mvvline()	mvwvline()	hline(3X)
mvvline_set()	mvwvline_set()	hline_set(3X)

SEE ALSO

curses(3X), w(3X).

Character User Interface Reference Manual

mvcur(3X)

NAME

`mvcur` - output cursor movement commands to the terminal

SYNOPSIS

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

```
#include <curses.h>
int mvcur(int oldrow, int oldcol, int newrow, int newcol);
```

DESCRIPTION

The `mvcur()` function outputs one or more commands to the terminal that move the terminal's cursor to (*newrow*, *newcol*), an absolute position on the terminal screen. The (*oldrow*, *oldcol*) arguments specify the former cursor position. Specifying the former position is necessary on terminals that do not provide coordinate-based movement commands. On terminals that provide these commands, Curses may select a more efficient way to move the cursor based on the former position. If (*newrow*, *newcol*) is not a valid address for the terminal in use, `mvcur()` fails. If (*oldrow*, *oldcol*) is the same as (*newrow*, *newcol*), then `mvcur()` succeeds without taking any action. If `mvcur()` outputs a cursor movement command, it updates its information concerning the location of the cursor on the terminal.

RETURN VALUE

Upon successful completion, `mvcur()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

NOTES

After use of `mvcur()`, the model Curses maintains of the state of the terminal might not match the actual state of the terminal. The application should touch and refresh the window before resuming conventional use of Curses.

SEE ALSO

`curses(3X)`, `doupdate(3X)`, `is_linetouched(3X)`, `curses(5)`.

Character User Interface Reference Manual

mvderwin(3X)

NAME

`mvderwin -` define window coordinate transformation

SYNOPSIS

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

```
#include <curses.h>
int mvderwin(WINDOW *win, int par_y, int par_x);
```

DESCRIPTION

The `mvderwin()` function specifies a mapping of characters. The function identifies a mapped area of the parent of the specified window, whose size is the same as the size of the specified window and whose origin is at (*par_y*, *par_x*) of the parent window.

- During any refresh of the specified window, the characters displayed in that window's display area of the terminal are taken from the mapped area.
- Any references to characters in the specified window obtain or modify characters in the mapped area.

That is, `mvderwin()` defines a coordinate transformation from each position in the mapped area to a corresponding position (same *y*, *x* offset from the origin) in the specified window.

RETURN VALUE

Upon successful completion, `mvderwin()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)`, `derwin(3X)`, `doupdate(3X)`, `dupwin(3X)`, `curses(5)`.

Character User Interface Reference Manual

mvprintw(3X)

NAME

`mvprintw, mvwprintw, printw, wprintw -` printformatted output in window

SYNOPSIS

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

```
#include <curses.h>
int mvprintw(int y, int x, char *fmt, ...);
int mvwprintw(WINDOW *win, int y, int x, char *fmt, ...);
int printw(char *fmt, ...);
int wprintw(WINDOW *win, char *fmt, ...);
```

DESCRIPTION

The `mvprintw()`, `mvwprintw()`, `printw()` and `wprintw()` functions are analogous to `printf()`. The effect of these functions is as though `sprintf()` were used to format the string, and then `waddstr()` were used to add that multi-byte string to the current or specified window at the current or specified cursor position.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`fprintf`(3S), `addnstr`(3X), `curses`(3X), `curses`(5).

Character User Interface Reference Manual

mvscanw(3X)

NAME

`mvscanw, mvwscanw, scanw, wscanw -` convertformatted input from a window

SYNOPSIS

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

```
#include <curses.h>
int mvscanw(int y, int x, char *fmt, ...);
int mvwscanw(WINDOW *win, int y, int x, char *fmt, ...);
int scanw(char *fmt, ...);
int wscanw(WINDOW *win, char *fmt, ...);
```

DESCRIPTION

These functions are similar to `scanf()`. Their effect is as though `mvwgetstr()` were called to get a multi-byte character string from the current or specified window at the current or specified cursor position, and then `sscanf()` were used to interpret and convert that string.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`wcstombs`(3C) , `fscanf`(3S) , `curses`(3X) , `getnstr`(3X) , `printw`(3X) , `curses`(5) .

Character User Interface Reference Manual

mvwin(3X)

NAME

`mvwin` - move window

SYNOPSIS

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

```
#include <curses.h>
int mvwin(WINDOW *win, int y, int x);
```

DESCRIPTION

The `mvwin()` function moves the specified window so that its origin is at position (*y*, *x*). If the move would cause any portion of the window to extend past any edge of the screen, the function fails and the window is not moved.

RETURN VALUE

Upon successful completion, `mvwin()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

NOTES

The application should not move subwindows by calling `mvwin()`.

SEE ALSO

`curses`(3X), `derwin`(3X), `doupdate`(3X), `is_linetouched`(3X), `curses`(5).

Character User Interface Reference Manual

napms(3X)

NAME

napms - suspend the calling process

SYNOPSIS

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

```
#include <curses.h>
int napms(int ms);
```

DESCRIPTION

The napms() function takes at least *ms* milliseconds to return.

RETURN VALUE

The napms() function returns OK.

ERRORS

No errors are defined.

NOTES

A more reliable method of achieving a timed delay is the usleep() function.

SEE ALSO

usleep(3), curses(3X), delay_output(3X), curses(5).

Character User Interface Reference Manual

newpad(3X)

NAME

newpad, pnoutrefresh, prefresh, subpad - padmanagement functions

SYNOPSIS

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

#include <curses.h>

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

int pnoutrefresh(WINDOW **pad*, int *pminrow*, int *pmincol*, int *sminrow*,□
int *smincol*, int *smaxrow*, int *smaxcol*);

int prefresh(WINDOW **pad*, int *pminrow*, int *pmincol*, int *sminrow*,□
□□□□□□ int *smincol*, int *smaxrow*, int *smaxcol*);

WINDOW *subpad(WINDOW **orig*, int *nlines*, int *ncols*, int *begin_y*,□
int *begin_x*);□

DESCRIPTION

The newpad() function creates a specialised WINDOW data structure representing a pad with *nlines* lines and *ncols* columns. A pad is like a window, except that it is not necessarily associated with a viewable part of the screen. Automatic refreshes of pads do not occur.

The subpad() function creates a subwindow within a pad with *nlines* lines and *ncols* columns. 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 affect both windows.

The prefresh() and pnoutrefresh() functions are analogous to wrefresh() and wnoutrefresh() except that they relate to pads instead of windows. The additional arguments indicate what part of the pad and screen are involved. The *pminrow* and *pmincol* arguments specify the origin of the rectangle to be displayed in the pad. The *sminrow*, *smincol*, *smaxrow* and *smaxcol* arguments specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad 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.

RETURN VALUE

Upon successful completion, the newpad() □and subpad() functions return a pointer to the pad data structure. Otherwise, they return a null pointer.

Upon successful completion, pnoutrefresh() and prefresh() return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

NOTES

To refresh a pad, call prefresh() or pnoutrefresh(), not wrefresh(). When porting code to use pads from WINDOWS, remember that these functions require additional arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display.

Although a subwindow and its parent pad may share memory representing characters in the pad, they need not share status information about what has changed in the pad. Therefore, after modifying a subwindow within a pad, it may be necessary to call touchwin() or touchline() on the pad before calling prefresh().

SEE ALSO

curses(3X), derwin(3X), doupdate(3X), is_linetouched(3X), curses(5).

Character User Interface Reference Manual

nl(3X)

NAME

`nl, nonl` - enable/disable newline translation

SYNOPSIS

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

```
#include <curses.h>
int nl(void);
int nonl(void);
```

DESCRIPTION

The `nl()` function enables a mode in which carriage return is translated to newline on input. The `nonl()` function disables the above translation. Initially, the above translation is enabled.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

The default translation adapts the terminal to environments in which newline is the line termination character. However, by disabling the translation with `nonl()`, the application can sense the pressing of the carriage return key.

SEE ALSO

`curses`(3X), `curses`(5).

Character User Interface Reference Manual

no(3X)

NAME

`no` - pointer page for functions with no prefix

DESCRIPTION

The `no` prefix indicates that a Curses function disables a mode. (The corresponding functions without the `no` prefix enable the same mode.)

The `no` functions are discussed together with the corresponding functions that do not have these prefixes.

The `nodelay()` function has an entry under its own name because there is no corresponding `delay()` function.

The `noqiflush()` and `notimeout()` functions have an entry under their own names because they precede the corresponding function without the `no` prefix in alphabetical order.

They are found on the following entries:

Function	Refer to
`nocbreak()`	`cbreak`(3X)
`noecho()`	`echo`(3X)
`nonl()`	`nl`(3X)
`noraw()`	`cbreak`(3X)

SEE ALSO

`curses`(3X) .

Character User Interface Reference Manual

nodelay(3X)

NAME

`nodelay -` enable or disable block during read

SYNOPSIS

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

```
#include <curses.h>
int nodelay(WINDOW *win, bool bf);
```

DESCRIPTION

The `nodelay()` function specifies whether Delay Mode or No Delay Mode is in effect for the screen associated with the specified window. If *bf* is `TRUE`, this screen is set to No Delay Mode. If *bf* is `FALSE`, this screen is set to Delay Mode. The initial state is `FALSE`.

RETURN VALUE

Upon successful completion, `nodelay()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X), `getch`(3X), `halfdelay`(3X), `curses`(5).

Character User Interface Reference Manual

noqiflush(3X)

NAME

`noqiflush, qiflush -` enable/disable queue flushing

SYNOPSIS

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

```
#include <curses.h>
void noqiflush(void);
void qiflush(void);
```

DESCRIPTION

The `qiflush()` function causes all output in the display driver queue to be flushed whenever an interrupt key (interrupt, suspend, or quit) is pressed. The `noqiflush()` causes no such flushing to occur. The default for the option is inherited from the display driver settings.

RETURN VALUE

These functions do not return a value.

ERRORS

No errors are defined.

NOTES

Calling `qiflush()` provides faster response to interrupts, but causes Curses to have the wrong idea of what is on the screen. The same effect is achieved outside Curses using the `NOFLSH` local mode flag specified in the XBD specification ("General Terminal Interface").

SEE ALSO

`curses(3X)`, `intrflush(3X)`, `curses(5)`.

Character User Interface Reference Manual

notimeout(3X)

NAME

`notimeout, timeout, wtimeout -` control blocking on input

SYNOPSIS

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

```
#include <curses.h>
int notimeout(WINDOW *win, bool bf);
void timeout(int delay);
void wtimeout(WINDOW *win, int delay);
```

DESCRIPTION

The `notimeout()` function specifies whether Timeout Mode or No Timeout Mode is in effect for the screen associated with the specified window. If *bf* is `TRUE`, this screen is set to No Timeout Mode. If *bf* is `FALSE`, this screen is set to Timeout Mode. The initial state is `FALSE`.

The `timeout()` and `wtimeout()` functions set blocking or non-blocking read for the current or specified window based on the value of *delay*:

- One or more blocking reads (indefinite waits for input) are used.
- One or more non-blocking reads are used. Any Curses input function will fail if every character of the requested string is not immediately available.
- Any Curses input function blocks for *delay* milliseconds and fails if there is still no input.

RETURN VALUE

Upon successful completion, the `notimeout()` function returns `OK`. Otherwise, it returns `ERR`.

The `timeout()` and `wtimeout()` functions do not return a value.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X), `getch`(3X), `halfdelay`(3X), `nodelay`(3X), `curses`(5).

Character User Interface Reference Manual

overlay(3X)

NAME

`overlay, overwrite -` copy overlapped windows

SYNOPSIS

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

```
#include <curses.h>
int overlay(const WINDOW *srcwin, WINDOW *dstwin);
int overwrite(const WINDOW *srcwin, WINDOW *dstwin);
```

DESCRIPTION

The `overlay()` and `overwrite()` functions overlay *srcwin* on top of *dstwin*. The *scrwin* and *dstwin* arguments need not be the same size; only text where the two windows overlap is copied.

The `overwrite()` function copies characters as though a sequence of `win_wch()` and `wadd_wch()` were performed with the destination window's attributes and background attributes cleared.

The `overlay()` function does the same thing, except that, whenever a character to be copied is the background character of the source window, `overlay()` does not copy the character but merely moves the destination cursor the width of the source background character.

If any portion of the overlaying window border is not the first column of a multi-column character then all the column positions will be replaced with the background character and rendition before the overlay is done. If the default background character is a multi-column character when this occurs, then these functions fail.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`copywin`(3X) , `curses`(3X) , `curses`(5) .

Character User Interface Reference Manual

pechochar(3X)

NAME

pechochar, pecho_wchar - write a character and rendition andimmediately refresh the pad

SYNOPSIS

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

```
#include <curses.h>
```

int pechochar(WINDOW *pad, chtype *ch*);

int pecho_wchar(WINDOW * *pad*, const cchar_t **wch*);

DESCRIPTION

The pechochar() and pecho_wchar() functions output one character to a pad and immediately refresh the pad. They are equivalent to a call to waddch() or wadd_wch(), respectively, followed by a call to prefresh(). The last location of the pad on the screen is reused for the arguments to prefresh().

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

NOTES

The pechochar() function is only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

SEE ALSO

curses(3X) , echochar(3X) , echo_char(3X), newpad(3X) , curses(5) .

Character User Interface Reference Manual

putp(3X)

NAME

putp, tputs - output commands to the terminal

SYNOPSIS

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

```
#include <term.h>
int putp(const char *str);
int tputs(const char *str, int affcnt, int (*putfunc)(int));
```

DESCRIPTION

These functions output commands contained in the terminfo(4) database to the terminal.

The putp() function is equivalent to tputs(str, 1, putchar). The output of putp() always goes to stdout, not to the *fildes* specified in setupterm().

The tputs() function outputs *str* to the terminal. The *str* argument must be a terminfo string variable or the return value from tgetstr(), tgoto(), tigetstr() or tparm(). The *affcnt* argument is the number of lines affected, or 1 if not applicable. If the terminfo database indicates that the terminal in use requires padding after any command in the generated string, tputs() inserts pad characters into the string that is sent to the terminal, at positions indicated by the terminfo database. The tputs() function outputs each character of the generated string by calling the user-supplied function *putfunc* (see below).

The user-supplied function *putfunc* (specified as an argument to tputs()) is either putchar() or some other function with the same prototype. The tputs() function ignores the return value of *putfunc*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

NOTES

Changing the terminal attributes using these functions may cause the renditions of characters within a curses window to be altered on some terminals.

After use of any of these functions, the model Curses maintains of the state of the terminal might not match the actual state of the terminal. The application should touch and refresh the window before resuming conventional use of Curses.

SEE ALSO

putchar(3S), curses(3X), doupdate(3X), is_linetouched(3X), tgetent(3X), tigetflag(3X), terminfo(4), term(5).

Character User Interface Reference Manual

redrawwin(3X)

NAME

`redrawwin, wredrawln -` line update status functions

SYNOPSIS

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

```
#include <curses.h>
int redrawwin(WINDOW *win);
int wredrawln(WINDOW *win, int beg_line, int num_lines);
```

DESCRIPTION

The `redrawwin()` and `wredrawln()` functions inform the implementation that some or all of the information physically displayed for the specified window may have been corrupted. The `redrawwin()` function marks the entire window; `wredrawln()` marks only *num_lines* lines starting at line number *beg_line*. The functions prevent the next refresh operation on that window from performing any optimisation based on assumptions about what is physically displayed there.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise they return `ERR`.

ERRORS

No errors are defined.

NOTES

The `redrawwin()` and `wredrawln()` functions could be used in a text editor to implement a command that redraws some or all of the screen.

SEE ALSO

`clearok`(3X), `curses`(3X), `doupdate`(3X), `curses`(5).

Character User Interface Reference Manual

resetty(3X)

NAME

`resetty, savetty -` save/restore terminal mode

SYNOPSIS

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

```
#include <curses.h>
int resetty(void);
int savetty(void);
```

DESCRIPTION

The `resetty()` function restores the program mode as of the most recent call to `savetty()`.

The `savetty()` function saves the state that would be put in place by a call to `reset_prog_mode()`.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses(3X)`, `def_prog_mode(3X)`, `curses(5)`.

Character User Interface Reference Manual

ripoffline(3X)

NAME

`ripoffline` - reserve a line for a dedicated purpose

SYNOPSIS

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

```
#include <curses.h>
int ripoffline(int line, int (*init)(WINDOW *win, int columns));
```

DESCRIPTION

The `ripoffline()` function reserves a screen line for use by the application.

Any call to `ripoffline()` must precede the call to `initscr()` or `newterm()`. If *line* is positive, one line is removed from the beginning of `stdscr`; if *line* is negative, one line is removed from the end. Removal occurs during the subsequent call to `initscr()` or `newterm()`. When the subsequent call is made, the function pointed to by *init* is called with two arguments: a `WINDOW` pointer to the one-line window that has been allocated and an integer with the number of columns in the window. The initialisation function cannot use the `LINES` and `COLS` external variables and cannot call `wrefresh()` or `doupdate()`, but may call `wnoutrefresh()`.

Up to five lines can be ripped off. Calls to `ripoffline()` above this limit have no effect but report success.

RETURN VALUE

The `ripoffline()` function returns `OK`.

ERRORS

No errors are defined.

NOTES

Calling `slk_init()` reduces the size of the screen by one line if `initscr()` eventually uses a line from `stdscr` to emulate the soft labels. If `slk_init()` rips off a line, it thereby reduces by one the number of lines an application can reserve by subsequent calls to `ripoffline()`. Thus, portable applications that use soft label functions should not call `ripoffline()` more than four times.

When `initscr()` or `newterm()` calls the initialisation function pointed to by *init*, the implementation may pass `NULL` for the `WINDOW` pointer argument *win*. This indicates inability to allocate a one-line window for the line that the call to `ripoffline()` ripped off. Portable applications should verify that *win* is not `NULL` before performing any operation on the window it represents.

SEE ALSO

`curses(3X)`, `doupdate(3X)`, `initscr(3X)`, `slk_attroff(3X)`, `stdscr(3X)`, `curses(5)`.

Character User Interface Reference Manual

scrl(3X)

NAME

`scrl, scroll, wscrl -` scroll a Curses window

SYNOPSIS

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

```
#include <curses.h>
int scrl(int n);
int scroll(WINDOW *win);
int wscrl(WINDOW *win, int n);
```

DESCRIPTION

The `scroll()` function scrolls *win* one line in the direction of the first line.

The `scrl()` and `wscrl()` functions scroll the current or specified window. If *n* is positive, the window scrolls *n* lines toward the first line. Otherwise, the window scrolls *–n* lines toward the last line.

These functions do not change the cursor position. If scrolling is disabled for the current or specified window, these functions have no effect. The interaction of these functions with `setsccreg()` is currently unspecified.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X) , `curses`(5) .

Character User Interface Reference Manual

scr_dump(3X)

NAME

`scr_dump, scr_init, scr_restore, scr_set -` screenfile input/output functions

SYNOPSIS

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

```
#include <curses.h>
int scr_dump(const char *filename);
int scr_init(const char *filename);
int scr_restore(const char *filename);
int scr_set(const char *filename);
```

DESCRIPTION

The `scr_dump()` function writes the current contents of the virtual screen to the file named by *filename* in an unspecified format.

The `scr_restore()` function sets the virtual screen to the contents of the file named by *filename*, which must have been written using `scr_dump()`. The next refresh operation restores the screen to the way it looked in the dump file.

The `scr_init()` function reads the contents of the file named by *filename* and uses them to initialise the Curses data structures to what the terminal currently has on its screen. The next refresh operation bases any updates on this information, unless either of the following conditions is true:

- The terminal has been written to since the virtual screen was dumped to *filename*
- The `terminfo` capabilities `rmcup` and `nrrmc` are defined for the current terminal.

The `scr_set()` function is a combination of `scr_restore()` and `scr_init()`. It tells the program that the information in the file named by *filename* is what is currently on the screen, and also what the program wants on the screen. This can be thought of as a screen inheritance function.

RETURN VALUE

On successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

The `scr_init()` function is called after `initscr()` or a `system()` call to share the screen with another process that has done a `scr_dump()` after its `endwin()` call.

To read a window from a file, call `getwin()`; to write a window to a file, call `putwin()`.

SEE ALSO

`open(2)`, `read(2)`, `write(2)`, `curses(3X)`, `delscreen(3X)`, `doupdate(3X)`, `endwin(3X)`, `getwin(3X)`, `terminfo(4)`, `curses(5)`.

Character User Interface Reference Manual

setcchar(3X)

NAME

`setcchar -` set cchar_t from a wide character string andrendition

SYNOPSIS

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

`#include <curses.h>`

`int setcchar(cchar_t *` *wcval*, const wchar_t **wch*, const attr_t *attrs*,

`short` *color_pair*, const void **opts*);

DESCRIPTION

The `setcchar()` function initialises the object pointed to by *wcval* according to the character attributes in *attrs*, the colour pair in *color_pair* and the wide character string pointed to by *wch*.

The *opts* argument is reserved for definition in a future edition of this manual page. Currently, the application must provide a null pointer as *opts*.

RETURN VALUE

Upon successful completion, `setcchar()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`attroff`(3X), `can_change_color`(3X), `curses`(3X), `getcchar`(3X), `curses`(5).

Character User Interface Reference Manual

set_term(3X)

NAME

`set_term -` switch between screens

SYNOPSIS

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

```
#include <curses.h>
SCREEN *set_term(SCREEN *new);
```

DESCRIPTION

The `set_term()` function switches between different screens. The *new* argument specifies the new current screen.

RETURN VALUE

Upon successful completion, `set_term()` returns a pointer to the previous screen. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

NOTES

This is the only function that manipulates `SCREEN` pointers; all other functions affect only the current screen.

SEE ALSO

`curses`(3X), `initscr`(3X), `curses`(5).

Character User Interface Reference Manual

slk_attroff(3X)

NAME

slk_attroff, slk_attr_off, slk_attron, slk_attr_on, slk_attrset, slk_attr_set, slk_clear, slk_color, slk_init, slk_label, slk_noutrefresh, slk_refresh, slk_restore, slk_set, slk_touch, slk_wset - softlabel functions

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
int slk_attroff(const chtype attrs);
int slk_attr_off(const attr_t attrs, void *opts);
int slk_attron(const chtype attrs);
int slk_attr_on(const attr_t attrs, void *opts);
int slk_attrset(const chtype attrs);
int slk_attr_set(const attr_t attrs, short color_pair_number, void *opts);
int slk_clear(void);
int slk_color(short color_pair_number);
int slk_init(int fmt);
char *slk_label(int labnum);
int slk_noutrefresh(void);
int slk_refresh(void);
int slk_restore(void);
int slk_set(int labnum, const char *label, int justify);
int slk_touch(void);
int slk_wset(int labnum, const wchar_t *label, int justify);
```

DESCRIPTION

The Curses interface manipulates the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, Curses takes over the bottom line of `stdscr`, reducing the size of `stdscr` and the value of the `LINES` external variable. There can be up to eight labels of up to eight display columns each.

To use soft labels, `slk_init()` must be called before `initscr()`, `newterm()` or `ripoffline()` is called. If `initscr()` eventually uses a line from `stdscr` to emulate the soft labels, then *fmt* determines how the labels are arranged on the screen. Setting *fmt* to 0 indicates a 3-2-3 arrangement of the labels; 1 indicates a 4-4 arrangement. Other values for *fmt* are unspecified.

The `slk_init()` function has the effect of calling `ripoffline()` to reserve one screen line to accommodate the requested format.

The `slk_set()` and `slk_wset()` functions specify the text of soft label number *labnum*, within the range from 1 to and including 8. The *label* argument is the string to be put on the label. With `slk_set()`, and `slk_wset()`, the width of the label is limited to eight column positions. A null string or a null pointer specifies a blank label. The *justify* argument can have the following values to indicate how to justify *label* within the space reserved for it:

`0` Align the start of *label* with the start of the space

`1` Center *label* within the space

`2` Align the end of *label* with the end of the space

The `slk_refresh()` and `slk_noutrefresh()` functions correspond to the `wrefresh()` and

`wnoutrefresh()` functions.

The `slk_label()` function obtains soft label number *labnum*.

The `slk_clear()` function immediately clears the soft labels from the screen.

The `slk_restore()` function immediately restores the soft labels to the screen after a call to `slk_clear()`.

The `slk_touch()` function forces all the soft labels to be output the next time `slk_noutrefresh()` or `slk_refresh()` is called.

The `slk_attron()`, `slk_attrset()` and `slk_attroff()` functions correspond to `attron()`, `attrset()`, and `attroff()`. They have an effect only if soft labels are simulated on the bottom line of the screen.

The `slk_attr_off()`, `slk_attr_on()`, `slk_attr_set()`, and `slk_color()` functions correspond to `slk_attroff()`, `slk_attrond()`, `slk_attrset()` and `color_set()` and thus support the attribute constants with the `WA_` prefix and colour.

The *opts* argument is reserved for definition in a future edition of this manual page. Currently, the application must provide a null pointer as *opts*.

RETURN VALUE

Upon successful completion, `slk_label()` returns the requested label with leading and trailing blanks stripped. Otherwise, it returns a null pointer.

Upon successful completion, the other functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

NOTES

When using multi-byte character sets, applications should check the width of the string by calling `mbstowcs()` and then `wcswidth()` before calling `slk_set()`. When using wide characters, applications should check the width of the string by calling `wcswidth()` before calling `slk_set()`.

Since the number of columns that a wide character string will occupy is codeset-specific, call `wcwidth()` and `wcswidth()` to check the number of column positions in the string before calling `slk_wset()`.

Most applications would use `slk_noutrefresh()` because a `wrefresh()` is likely to follow soon.

SEE ALSO

`mbstowcs(3C)`, `wcswidth(3C)`, `attr_get(3X)`, `attroff(3X)`, `curses(3X)`, `delscreen(3X)`, `ripoffline(3X)`, `stdscr(3X)`, `curses(5)`.

Character User Interface Reference Manual

standend(3X)

NAME

`standend, standout, wstandend, wstandout` - set andclear window attributes

SYNOPSIS

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

```
#include <curses.h>
int standend(void);
int standout(void);
int wstandend(WINDOW *win);
int wstandout(WINDOW *win);
```

DESCRIPTION

The `standend()` and `wstandend()` functions turn off all attributes of the current or specified window.

The `standout()` and `wstandout()` functions turn on the standout attribute of the current or specified window.

RETURN VALUE

These functions always return 1.

ERRORS

No errors are defined.

SEE ALSO

`attroff`(3X), `attr_get`(3X), `curses`(3X), `curses`(5).

Character User Interface Reference Manual

stdscr(3X)

NAME

`stdscr` - default window

SYNOPSIS

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

```
#include <curses.h>
extern WINDOW *stdscr;
```

DESCRIPTION

The external variable `stdscr` specifies the default window used by functions that do not specify a window using an argument of type `WINDOW *`. Other windows may be created using `newwin()`.

SEE ALSO

`curses(3X)`, `derwin(3X)`, `curses(5)`.

Character User Interface Reference Manual

syncok(3X)

NAME

`syncok, wcursyncup, wsyncdown, wsyncup -` synchronisea window with its parents or children

SYNOPSIS

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

```
#include <curses.h>
int syncok(WINDOW *win, bool bf);
void wcursyncup(WINDOW *win);
void wsyncdown(WINDOW *win);
void wsyncup(WINDOW *win);
```

DESCRIPTION

The `syncok()` function determines whether all ancestors of the specified window are implicitly touched whenever there is a change in the window. If *bf* is `TRUE`, such implicit touching occurs. If *bf* is `FALSE`, such implicit touching does not occur. The initial state is `FALSE`.

The `wcursyncup()` function updates the current cursor position of the ancestors of *win* to reflect the current cursor position of *win*.

The `wsyncdown()` function touches *win* if any ancestor window has been touched.

The `wsyncup()` function unconditionally touches all ancestors of *win*.

RETURN VALUE

Upon successful completion, `syncok()` returns `OK`. Otherwise, it returns `ERR`.

The other functions do not return a value.

ERRORS

No errors are defined.

NOTES

Applications seldom call `wsyncdown()` because it is called by all refresh operations.

SEE ALSO

`curses`(3X), `doupdate`(3X), `is_linetouched`(3X), `curses`(5).

Character User Interface Reference Manual

tam(3X)

NAME

`tam` - TAM transition libraries

SYNOPSIS

```
#include <tam.h>
cc –I /usr/add-on/include/tam [ flags ] files
–ltam –lcurses [libraries]
```

DESCRIPTION

These routines are used to port UNIX PC character-based TAM programs to the 3B processor line so that they will run using any terminal supported by `curses(3X)`, the low-level ETI library. Once a TAM program has been changed to remove machine-specific code, it can be recompiled with the standard TAM header file <`tam.h`> and linked with the TAM transition and `curses(3X)` libraries.

Note that TAM will probably not be supported in future releases.

FUNCTIONS

The following is a list of TAM routines supplied in the transition library. Those routines marked with a pound sign (#) are macros and do not return a value.

`addch (c)#`
See `curses(3X)`.

```
char c;
addstr (s)#
char *s;
int adf_gttok (ptr, tbl)
char *ptr;
struct s_kwtbl *tbl;
char *adf_gtwrd (sptr, dptr)
char *sptr, *dptr;
char *adf_gtxcd (sptr, dptr)
char *sptr, *dptr;
int attroff (attrs)
```

See `curses(3X)`.

```
long attrs;
int attron(attrs)
long attrs;
int baudrate()
int beep()
int cbreak()
int clear()
clearok (dummy, dummy)#
int dummy;
int clrtobot()
int clrtoeol()
int delch()
int deleteln()
int echo()
int endwin()
erase()#
int exhelp (hfile, htitle)
char *hfile, *htitle;
int fixterm()
```

See `curses(3X)`.

```
flash()#
int flushinp()
int form (form, op)
     form_t *form;
int op;
int getch()
```
See curses(3X).
```
getyx(win, r, c)#
int win, r, c;
int initscr()
int insch(ch)
char ch;
int insertln()
int iswind()
```
Always returns 0.
```
char *kcodemap (code)
```
See curses(3X).
```
unsigned char code;
int keypad (dummy, flag)
int dummy, flag;
leaveok (dummy, dummy)#
int dummy;
int menu (menu, op)
     menu_t *menu;
int op;
int message (mtype, hfile, htitle, format [, arg ...]
int mtype;
char *hfile, *htitle, *format;
move(r, c)#
```
See curses(3X).
```
int r, c;
mvaddch (r, c, ch)#
int r, c;
char ch;
mvaddstr (r, c, s)#
int r, c;
char *s;
unsigned long mvinch(r, c)
int r, c;
nl()#
```
NOT SUPPORTED
```
int nocbreak()
int nodelay (dummy, bool)
int dummy, bool;
int noecho()
nonl()#
```
NOT SUPPORTED
```
int pb_check (stream)
FILE *stream;
int pb_empty (stream)
FILE *stream;
int pb_gbuf (ptr, n, fn, stream)
char *ptr;
int n;
int (*fn) ();
FILE *stream;
```

```
char *pb_gets (ptr, n, stream)
char *ptr;
int n;
FILE *stream;
char *pb_name()
FILE *pb_open()
int pb_puts (ptr, stream)
char *ptr;
FILE *stream;
int pb_seek (stream)
FILE *stream;
int pb_weof (stream)
FILE *stream;
int printw (fmt[, arg1 ... argn])
```

See curses(3X).

```
char *fmt;
refresh()#
int resetterm()
int resetty()
int savetty()
int track (w, trk, op, butptr, whyptr)
```

See wgetc().

```
int w, op, *butptr, *whyptr;
track_t *trk;
int wcmd (wn, cp)
```

Outputs a null-terminated

```
short wn;
```

string to the entry/echo line.

```
char *cp;
int wcreate (row, col, height, width, flags)
```

Creates a window.

```
short row, col, height, width;
unsigned short flags;
int wdelete (wn)
```

Deletes the specified window.

```
short wn;
void wexit(ret)
int ret;
int wgetc (wn)
short wn;
int wgetmouse (wn, ms)
```

no-op; returns 0.

```
short wn;
struct umdata *ms;
int wgetpos (wn, rowp, colp)
```

Gets the current position (row,

```
short wn;
```

column) of the cursor in the

```
int *rowp, *colp;
```

specified window (*wn*).

```
int wgetsel()
```

Returns the currently selected window.

```
int wgetstat (wn, wstatp)
```

Returns the information in

```
short wn;
```

`WSTAT` for a window.

```
WSTAT *wstatp;
int wgoto (wn, row, col)
```

Moves the window's cursor to

```
short wn, row, col;
```

aspecified row, column.

```
void wicoff (wn, row, col, icp)
```

no-op; returns 0.

```
short wn, row, col;
struct icon *icp;
void wicon (wn, row, col, icp)
```

no-op; returns 0.

```
short wn, row, col;
struct icon *icp;
int wind (type, height, width, flags, pfont)
int type, height, width;
short flags;
char *pfont[];
void winit()
```

Sets up the process for window access.

```
int wlabel (wn, cp)
```

Outputs a null-terminated

```
short wn;
```

string to the window label area.

```
char *cp;
int wndelay (wn, bool)
int wn, bool;
void wnl (wn, flag)
short wn;
int flag;
int wpostwait()
```

Reverses the effects of `wprexec()`.

```
int wprexec()
```

Performs the appropriate actions for passing a window to a child process.

```
int wprintf (wn, fmt[, arg1 ... argn])
short wn;
char *fmt;
int wprompt (wn, cp)
```

Outputs a null-terminated

```
short wn;
```

string to the prompt line.

```
char *cp;
int wputc (wn, c)
```

Outputs a character

```
short wn;
```

to a window (*wn*).

```
char c;
int wputs (wn, cp)
```

Outputs a character string

```
short wn;
```

to a window.

```
char *cp;
int wrastop (w, srcbase, srcwidth, dstbase
```

NOT SUPPORTED

```
dstwidth, srcx, srcy, dstx,
dsty, width, height, srcop,
dstop, pattern)
int w;
unsigned short *srcbase, *dstbase, *pattern;
unsigned short srcwidth, dswidth, width, height;
unsigned short srcx, srcy, dstx, dsty;
char srcop, dstop;
int wreadmouse (wn, xp, yp, bp, rp)
```

no-op; returns 0.

```
short wn;
int *xp, *yp, *bp, *rp;
int wrefresh (wn)
```

Flushes all output

```
short wn;
```

to the window.

```
int wselect (wn)
```

Selects the specified window

```
short wn;
```

as the current or active one.

```
int wsetmouse (wn, ms)
```

no-op; returns 0.

```
short wn;
struct umdata *ms;
int wsetstat (wn, wstatp)
```

Sets the status for a window.

```
short wn;
WSTAT *wstatp;
int wslk (wn, 0, slong1, slong2, sshort)
```

Writes a null-terminated

```
short wn;
```

string to a set of

```
char *slong1, *slong2, *sshort;
```

soft-labeled keys (SLK).

```
int wslk (wn, kn, llabel, slabel)
```

Writes a null-terminated

```
short wn, kn;
```

stringto a screen-labeled key.

```
char *llabel, *slabel;
```

The alternate form writes all the screen-labeled keys at once more efficiently.

```
int wuser (wn, cp)
```

NOT SUPPORTED

```
short wn;
     char *cp;
```

SEE ALSO

curses(3X) .

Character User Interface Reference Manual

termattrs(3X)

NAME

`termattrs -` get supported terminal video attributes

SYNOPSIS

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

```
#include <curses.h>
chtype termattrs(void);
attr_t term_attrs(void);
```

DESCRIPTION

The `termattrs()` function extracts the video attributes of the current terminal which is supported by the `chtype` data type.

The `term_attrs()` function extracts information for the video attributes of the current terminal which is supported for a `cchar_t` data type.

RETURN VALUE

The `termattrs()` function returns a logical OR of `A_` values of all video attributes supported by the terminal.

The `term_attrs()` function returns a logical OR of `WA_` values of all video attributes supported by the terminal.

ERRORS

No errors are defined.

SEE ALSO

`attroff`(3X), `attr_get`(3X), `curses`(3X), `curses`(5).

Character User Interface Reference Manual

termname(3X)

NAME

`termname` - get terminal name

SYNOPSIS

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

```
#include <curses.h>
char *termname(void);
```

DESCRIPTION

The `termname()` function obtains the terminal name as recorded by `setupterm()`.

RETURN VALUE

The `termname()` function returns a pointer to the terminal name.

ERRORS

No errors are defined.

SEE ALSO

`getenv`(3C), `curses`(3X), `del_curterm`(3X), `initscr`(3X), `curses`(5).

Character User Interface Reference Manual

tgetent(3X)

NAME

`tgetent, tgetflag, tgetnum, tgetstr, tgoto -` termcap database emulation

SYNOPSIS

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

```
#include <term.h>
int tgetent(char *bp, const char *name);
int tgetflag(char id[2]);
int tgetnum(char id[2]);
char *tgetstr(char id[2], char **area);
char *tgoto(char *cap, int col, int row);
```

DESCRIPTION

The `tgetent()` function looks up the `termcap` entry for *name*. The emulation ignores the buffer pointer *bp*.

The `tgetflag()` function gets the boolean entry for *id*.

The `tgetnum()` function gets the numeric entry for *id*.

The `tgetstr()` function gets the string entry for *id*. If *area* is not a null pointer and does not point to a null pointer, `tgetstr()` copies the string entry into the buffer pointed to by **area* and advances the variable pointed to by *area* to the first byte after the copy of the string entry.

The `tgoto()` function instantiates the parameters *col* and *row* into the capability *cap* and returns a pointer to the resulting string.

All of the information available in the `terminfo(4)` database need not be available through these functions.

RETURN VALUE

Upon successful completion, functions that return an integer return `OK`. Otherwise, they return `ERR`.

Functions that return pointers return a null pointer on error.

ERRORS

No errors are defined.

NOTES

These functions are included as a conversion aid for programs that use the `termcap` library. Their arguments are the same and the functions are emulated using the `terminfo` database.

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the `A_` prefix.

Any terminal capabilities from the `terminfo` database that cannot be retrieved using these interfaces can be retrieved using the interfaces described on the `tigetflag(3X)` manual page.

Portable applications must use `tputs()` to output the strings returned by `tgetstr()` and `tgoto()`.

SEE ALSO

`putc(3S)`, `curses(3X)`, `setupterm(3X)`, `tigetflag(3X)`, `terminfo(4)`, `term(5)`.

Character User Interface Reference Manual

tigetflag(3X)

NAME

`tigetflag, tigetnum, tigetstr, tparm -` retrievecapabilities from the terminfo database

SYNOPSIS

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

```
#include <term.h>
int tigetflag(char *capname);
int tigetnum(char *capname);
char *tigetstr(char *capname);
char *tparm(char *cap, long p1, long p2, long p3, long p4,□
□□□□□ long p5, long p6, long p7, long p8, long p9);□
```

DESCRIPTION

The `tigetflag()`, `tigetnum()`, and `tigetstr()` functions obtain boolean, numeric and string capabilities, respectively, from the selected record of the `terminfo` database. For each capability, the value to use as *capname* appears in the **Capname** column in the table in `terminfo(4)`.

The `tparm()` function takes as *cap* a string capability. If *cap* is parameterised (as described in "X/Open Curses, Issue 4, Version 2, Section A.1.2"), `tparm()` resolves the parameterisation. If the parameterised string refers to parameters `%p1` through `%p9`, then `tparm()` substitutes the values of *p1* through *p9*, respectively.

RETURN VALUE

Upon successful completion, `tigetflg()`, `tigetnum()` and `tigetstr()` return the specified capability. The `tigetflag()` function returns –2 if *capname* is not a boolean capability. The `tigetnum()` function returns –2 if *capname* is not a numeric capability. The `tigetstr()` function returns (`char *`)–1 if *capname* is not a string capability.

Upon successful completion, `tparm()` returns *str* with parameterisation resolved. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

NOTES

For parameterised string capabilities, the application should pass the return value from `tigetstr()` to `tparm()`, as described above.

Applications intending to send terminal capabilities directly to the terminal (which should only be done using `tputs()` or `putp()`) instead of using Curses, normally should obey the following rules:

- Call `reset_shell_mode()` to restore the display modes before exiting.
- If using cursor addressing, output `enter_ca_mode` upon startup and output `exit_ca_mode` before exiting.
- If using shell escapes, output `exit_ca_mode` and call `reset_shell_mode()` before calling the shell; call `reset_prog_mode()` and output `enter_ca_mode` after returning from the shell.

All parameterised terminal capabilities defined in `terminfo(4)` can be passed to `tparm()`. Some implementations create their own capabilities, create capabilities for non-terminal devices, and redefine the capabilities in `terminfo(4)`. These practices are non-conforming because it may be that `tparm()` cannot parse these user-defined strings.

SEE ALSO

curses(3X), def_prog_mode(3X), tgetent(3X), putp(3X), terminfo(4), term(5).

Character User Interface Reference Manual

typeahead(3X)

NAME

`typeahead` - control checking for typeahead

SYNOPSIS

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

```
#include <curses.h>
int typeahead(int fildes);
```

DESCRIPTION

The `typeahead()` function controls the detection of typeahead during a refresh, based on the value of *fildes*:

- If *fildes* is a valid file descriptor, typeahead is enabled during refresh; Curses periodically checks *fildes* for input and aborts the refresh if any character is available. (This is the initial setting, and the typeahead file descriptor corresponds to the input file associated with the screen created by `initscr()` or `newterm()`.) The value of *fildes* need not be the file descriptor on which the refresh is occurring.
- If *fildes* is –1, Curses does not check for typeahead during refresh.

RETURN VALUE

Upon successful completion, `typeahead()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X), `doupdate`(3X), `getch`(3X), `initscr`(3X), `curses`(5).

Character User Interface Reference Manual

unctrl(3X)

NAME

`unctrl` - generate printable representation of a character

SYNOPSIS

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

```
#include <unctrl.h>
char *unctrl(chtype c);
```

DESCRIPTION

The `unctrl()` function generates a character string that is a printable representation of c. If c is a control character, it is converted to the $\hat{}X$ notation. If c contains rendition information, the effect is undefined.

RETURN VALUE

Upon successful completion, `unctrl()` returns the generated string. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X), `keyname`(3X), `wunctrl`(3X), `unctrl`(5).

Character User Interface Reference Manual

ungetch(3X)

NAME

ungetch, unget_wch - push a character onto the input queue

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
int ungetch(int ch);
int unget_wch(const wchar_t wch);
```

DESCRIPTION

The ungetch() function pushes the single-byte character *ch* onto the head of the input queue.

The unget_wch() function pushes the wide character *wch* onto the head of the input queue.

One character of push-back is guaranteed. The result of successive calls without an intervening call to getch() or get_wch() are unspecified.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

SEE ALSO

curses(3X), getch(3X), get_wch(3X), curses(5).

Character User Interface Reference Manual

use_env(3X)

NAME

use_env - specify source of screen size information

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
void use_env(bool boolval);
```

DESCRIPTION

The use_env() function specifies the technique by which the implementation determines the size of the screen. If *boolval* is FALSE, the implementation uses the values of *lines* and *columns* specified in the terminfo database. If *boolval* is TRUE, the implementation uses the LINES and COLUMNS environment variables. The initial value is TRUE.

Any call to use_env() must precede calls to initscr(), newterm() or setupterm().

RETURN VALUE

The function does not return a value.

ERRORS

No errors are defined.

SEE ALSO

curses(3X), del_curterm(3X), initscr(3X), curses(5).

Character User Interface Reference Manual

vidattr(3X)

NAME

vidattr, vid_attr, vidputs, vid_puts - outputattributes to the terminal

SYNOPSIS

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

#include <curses.h>

int vidattr(chtype *attr*);

int vid_attr(attr_t *attr*, short *color_pair_number*, void **opt*);

int vidputs(chtype *attr*, int (**putfunc*)(int));

int vid_puts(attr_t *attr*, short *color_pair_number*, void **opt*, int□
□
□□□□□ (**putfunc*)(int));□

DESCRIPTION

These functions output commands to the terminal that change the terminal's attributes.

If the terminfo database indicates that the terminal in use can display characters in the rendition specified by *attr*, then vidattr() outputs one or more commands to request that the terminal display subsequent characters in that rendition. The function outputs by calling putchar(). The vidattr() function neither relies on nor updates the model which Curses maintains of the prior rendition mode.

The vidputs() function computes the same terminal output string that vidattr() does, based on *attr*, but vidputs() outputs by calling the user-supplied function *putfunc*. The vid_attr() and vid_puts() functions correspond to vidattr() and vidputs() respectively, but take a set of arguments, one of type attr_t for the attributes, short for the colour pair number and a void□*, and thus support the attribute constants with the WA_ prefix.

The *opts* argument is reserved for definition in a future edition of this manual page. Currently, the application must provide a null pointer as *opts*.

The user-supplied function *putfunc* (which can be specified as an argument to either vidputs() or vid_puts()) is either putchar() or some other function with the same prototype. Both the vidputs() and the vid_puts() function ignore the return value of *putfunc*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

NOTES

After use of any of these functions, the model Curses maintains of the state of the terminal might not match the actual state of the terminal. The application should touch and refresh the window before resuming conventional use of Curses.

Use of these functions requires that the application contain so much information about a particular class of terminal that it defeats the purpose of using Curses.

On some terminals, a command to change rendition conceptually occupies space in the screen buffer (with or without width). Thus, a command to set the terminal to a new rendition would change the rendition of some characters already displayed.

SEE ALSO

putchar(3S), putwchar(3S), curses(3X), doupdate(3X), is_linetouched(3X), tigetflag(3X), curses(5).

Character User Interface Reference Manual

vwprintw(3X)

NAME

vwprintw - print formatted output in window

SYNOPSIS

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

```
#include <varargs.h>
#include <curses.h>
int vwprintw(WINDOW *, char *, va_list varglist);
```

DESCRIPTION

The vwprintw() function achieves the same effect as wprintw() using a variable argument list. The third argument is a va_list, as defined in <varargs.h> [see varargs(5)].

RETURN VALUE

Upon successful completion, vwprintw() returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

NOTES

The vwprintw() function is deprecated because it relies on deprecated functions in the XSH specification. The vw_printw() function is preferred. The use of the vwprintw() and the vw_printw() functions in the same file will not work, due to the requirement to include varargs.h and stdarg.h which both contain definitions of va_list.

SEE ALSO

fprintf(3S), curses(3X), mvprintw(3X), vw_printw(3X), curses(5), varargs(5).

Character User Interface Reference Manual

vwscanw(3X)

NAME

`vwscanw` - convert formatted input from a window

SYNOPSIS

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

```
#include <varargs.h>
#include <curses.h>
int vwscanw(WINDOW *, char *, va_list varglist);
```

DESCRIPTION

The `vwscanw()` function achieves the same effect as `wscanw()` using a variable argument list. The third argument is a `va_list`, as defined in `<varargs.h>` [see `varargs(5)`].

RETURN VALUE

Upon successful completion, `vwscanw()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

NOTES

The `vwscanw()` function is deprecated because it relies on deprecated functions in the XSH specification. The `vw_scanw()` function is preferred. The use of the `vwscanw()` and the `vw_scanw()` functions in the same file will not work, due to the requirement to include `varargs.h` and `stdarg.h` which both contain definitions of `va_list`.

SEE ALSO

`fscanf(3S)`, `curses(3X)`, `mvscanw(3X)`, `vw_scanw(3X)`, `curses(5)`, `varargs(5)`.

Character User Interface Reference Manual

vw_printw(3X)

NAME

`vw_printw -` print formatted output in window

SYNOPSIS

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

```
#include <stdarg.h>
#include <curses.h>
int vw_printw(WINDOW *, char *, va_list varglist);
```

DESCRIPTION

The `vw_printw()` function achieves the same effect as `wprintw()` using a variable argument list. The third argument is a `va_list`, as defined in `<stdarg.h>` [see `stdarg(5)`].

RETURN VALUE

Upon successful completion, `vw_printw()` returns `OK`. Otherwise, it returns `ERR`.

ERRORS

No errors are defined.

NOTES

The `vw_printw()` function is preferred over `vwprintw()`. The use of the `vwprintw()` and the `vw_printw()` functions in the same file will not work, due to the requirement to include `varargs.h` and `stdarg.h` which both contain definitions of `va_list`.

SEE ALSO

`fprintf(3S)`, `curses(3X)`, `mvprintw(3X)`, `curses(5)`, `stdarg(5)`.

Character User Interface Reference Manual

vw_scanw(3X)

NAME

vw_scanw - convert formatted input from a window

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <stdarg.h>
#include <curses.h>
int vw_scanw(WINDOW *, char *, va_list varflist);
```

DESCRIPTION

The vw_scanw() function achieves the same effect as wscanw() using a variable argument list. The third argument is a va_list, as defined in <stdarg.h> [see stdarg(5)].

RETURN VALUE

Upon successful completion, vw_scanw() returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

NOTES

The vw_scanw() function is preferred over vwscanw(). The use of the vwscanw() and the vw_scanw() functions in the same file will not work, due to the requirement to include varargs.h and stdarg.h which both contain definitions of va_list.

SEE ALSO

fscanf(3S), curses(3X), mvscanw(3X), curses(5), stdarg(5).

Character User Interface Reference Manual

w(3X)

NAME

`w` - pointer page for functions with w prefix

DESCRIPTION

Most uses of the `w` prefix indicate that a Curses function takes a *win* argument that specifies the affected window. (The corresponding functions without the `w` prefix operate on the current window.)

The `wunctrl()` function is an exception to this rule and has an entry under its own name.

The `w` functions are discussed together with the corresponding functions without the `w` prefix. They are found on the following entries:

Function	Refer to
`waddch()`	`addch`(3X)
`waddchnstr()`	`addchstr`(3X)
`waddchstr()`	`addchstr`(3X)
`waddnstr()`	`addnstr`(3X)
`waddstr()`	`addnstr`(3X)
`waddnwstr()`	`addnwstr`(3X)
`waddwstr()`	`addnwstr`(3X)
`wadd_wch()`	`add_wch`(3X)
`wadd_wchnstr()`	`add_wchnstr`(3X)
`wadd_wchstr()`	`add_wchnstr`(3X)
`wattroff()`	`attroff`(3X)
`wattron()`	`attroff`(3X)
`wattrset()`	`attroff`(3X)
`wattr_get()`	`attr_get`(3X)
`wattr_off()`	`attr_get`(3X)
`wattr_on()`	`attr_get`(3X)
`wattr_set()`	`attr_get`(3X)
`wbkgd()`	`bkgd`(3X)
`wbkgdset()`	`bkgd`(3X)
`wbkgrnd()`	`bkgrnd`(3X)
`wbkgrndset()`	`bkgrnd`(3X)
`wborder()`	`border`(3X)
`wborder_set()`	`border_set`(3X)
`wchgat()`	`chgat`(3X)
`wclear()`	`clear`(3X)
`wclrtobot()`	`clrtobot`(3X)
`wclrtoeol()`	`clrtoeol`(3X)

wcursyncup() *	syncok(3X)
wdelch()	delch(3X)
wdeleteln()	deleteln(3X)
wechochar()	echochar(3X)
wecho_wchar()	echo_wchar(3X)
werase()	clear(3X)
wgetbkgrnd()	bkgrnd(3X)
wgetch()	getch(3X)
wgetnstr()	getnstr(3X)
wgetn_wstr()	getn_wstr(3X)
wgetstr()	getnstr(3X)
wget_wch()	get_wch(3X)
wget_wstr()	getn_wstr(3X)
whline()	hline(3X)
whline_set()	hline_set(3X)
winch()	inch(3X)
winchnstr()	inchnstr(3X)
winchstr()	inchnstr(3X)
winnstr()	innstr(3X)
winnwstr()	innwstr(3X)
winsch()	insch(3X)
winsdelln()	insdelln(3X)
winsertln()	insertln(3X)
winsnstr()	insnstr(3X)
winsstr()	insnstr(3X)
winstr()	innstr(3X)
wins_nwstr()	ins_nwstr(3X)
wins_wch()	ins_wch(3X)
wins_wstr()	ins_nwstr(3X)
winwstr()	innwstr(3X)
win_wch()	in_wch(3X)
win_wchnstr()	in_wchnstr(3X)
win_wchstr()	in_wchnstr(3X)
wmove()	move(3X)
wnoutrefresh() *	doupdate(3X)
wprintw()	mvprintw(3X)
wredrawln()	redrawln(3X)
wrefresh()	doupdate(3X)

wscanw()	mvscanw(3X)
wscrl()	scrl(3X)
wsetscrreg()	clearok(3X)
wstandend()	standend(3X)
wstandout()	standend(3X)
wsyncdown() *	syncok(3X)
wsyncup() *	syncok(3X)
wtimeout()	notimeout(3X)
wtouchln() *	is_linetouch(3X)
wvline()	hline(3X)
wvline_set()	hline_set(3X)

* = There is no corresponding function without the w prefix.

SEE ALSO

curses(3X) .

Character User Interface Reference Manual

wunctrl(3X)

NAME

`wunctrl -` generate printable representation of a wide character

SYNOPSIS

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

```
#include <curses.h>
wchar_t *wunctrl(cchar_t * wc);
```

DESCRIPTION

The `wunctrl()` function generates a wide character string that is a printable representation of the wide character *wc*.

This function also performs the following processing on the input argument:

- Control characters are converted to the ^*X* notation.
- Any rendition information is removed.

RETURN VALUE

Upon successful completion, `wunctrl()` returns the generated string. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

SEE ALSO

`curses`(3X), `keyname`(3X), `unctrl`(3X), `curses`(5).

Character User Interface Reference Manual

COLS(3X)

NAME

`COLS` - number of columns on terminal screen

SYNOPSIS

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

```
#include <curses.h>
extern int COLS;
```

DESCRIPTION

The external variable `COLS` indicates the number of columns on the terminal screen.

SEE ALSO

`curses`(3X), `initscr`(3X), `curses`(5).

Character User Interface Reference Manual

LINES(3X)

NAME

LINES - number of lines on terminal screen

SYNOPSIS

```
cc [ flag ... ] file ... -lcurses [ library ... ]
#include <curses.h>
extern int LINES;
```

DESCRIPTION

The external variable LINES indicates the number of lines on the terminal screen.

SEE ALSO

curses(3X), initscr(3X), curses(5).

File Formats(4)

Character User Interface Reference Manual

term(4)

NAME

term - format of compiled terminfo files

SYNOPSIS

```
/usr/lib/share/terminfo/?/*
```

DESCRIPTION

Compiled terminfo(4) description files are placed under the directory /usr/share/lib/terminfo. In order to avoid a linear search of a huge UNIX system directory, a two-level scheme is used: /usr/share/lib/terminfo/*c*/*name* where *name* is the name of the terminal, and *c* is the first character of *name*. Thus, 97801 can be found in the file /usr/share/lib/terminfo/9/97801. Synonyms for the same terminal are implemented by multiple links to the same compiled file.

The format has been chosen so that it is the same on all hardware. An 8-bit byte is assumed, but no assumptions about byte ordering or sign extension are made. Thus, these binary terminfo files can be transported to other hardware with 8-bit bytes.

Short integers are stored in two 8-bit bytes. The first byte contains the least significant 8 bits of the value, and the second byte contains the most significant 8 bits. (Thus, the value represented is 256**second*+*first*.) The value -1 is represented by 0377,0377, and the value -2 is represented by 0376,0377; other negative values are illegal. The -1 generally means that a capability is missing from this terminal. The -2 means that the capability has been cancelled in the terminfo source and also is to be considered missing.

The compiled file is created from the source file descriptions of the terminals [see the -I option of infocmp(1M)] by using the terminfo compiler, tic(1M), and read by the routine setupterm(3X). The file is divided into six parts in the following order: the header, terminal names, boolean flags, numbers, strings, and string table.

The header section begins the file. This section contains six short integers in the format described below. These integers are:

1. the magic number (octal 0432);
2. the size, in bytes, of the names section;
3. the number of bytes in the boolean section;

4. the number of short integers in the numbers section;
5. the number of offsets (short integers) in the strings section;
6. the size, in bytes, of the string table.

The terminal names section comes next. It contains the first line of the `terminfo` description, listing the various names for the terminal, separated by the bar (|) character [see `term_names(5)`]. The section is terminated with an `ASCII NUL` character.

The boolean flags have one byte for each flag. This byte is either `0` or `1` as the flag is present or absent. The value of `2` means that the flag has been cancelled. The capabilities are in the same order as the file `<term.h>`.

Between the boolean section and the number section, a null byte is inserted, if necessary, to ensure that the number section begins on an even byte offset. All short integers are aligned on a short word boundary.

The numbers section is similar to the boolean flags section. Each capability takes up two bytes, and is stored as a short integer. If the value represented is `−1` or `−2`, the capability is taken to be missing.

The strings section is also similar. Each capability is stored as a short integer, in the format above. A value of `−1` or `−2` means the capability is missing. Otherwise, the value is taken as an offset from the beginning of the string table. Special characters in ^X or \c notation are stored in their interpreted form, not the printing representation. Padding information ($<nn>) and parameter information (%x) are stored intact in uninterpreted form.

The final section is the string table. It contains all the values of string capabilities referenced in the string section. Each string is null terminated.

Note that it is possible for `setupterm()` to expect a different set of capabilities than are actually present in the file. Either the database may have been updated since `setupterm()` has been recompiled (resulting in extra unrecognized entries in the file) or the program may have been recompiled more recently than the database was updated (resulting in missing entries). The routine `setupterm()` must be prepared for both possibilities – this is why the numbers and sizes are included. Also, new capabilities must always be added at the end of the lists of boolean, number, and string capabilities.

As an example, here is terminal information on the 97801 terminal as output by the `infocmp −I 97801` command:

```
standard|97801|97808,
 am, hs, npc,
 cols#80, lines#24,
 acsc=+K\,L.N-Mf?jEkClBmDnJqAtFuGvIwHx@~\,, bel=^G,
 blink=\E[5m, cbt=\E[Z, civis=\E[6p, clear=\E[H\E[2J,
 cnorm=\E[7p, cr=\r, csr=\E[%i%p1%d;%p2%dr,
 cub=\E[%p1%dD, cub1=\b, cud=\E[%p1%dB, cud1=\E[B,
 cuf=\E[%p1%dC, cuf1=\E[C, cup=\E[%i%p1%d;%p2%dH,
 cuu=\E[%p1%dA, cuu1=\E[A, dch=\E[%p1%dP, dch1=\E[P,
 dim=\E[2m, dl=\E[%p1%dM, dl1=\E[M,
 dsl=\E[s\E[25;1H\E[K\E[u, ed=\E[0J, el=\E[0K\E[0m,
 fsl=\E[u, home=\E[H, ht=\t, ich=\E[%p1%d@, ich1=\E[@,
 il=\E[%p1%dL, il1=\E[L, ind=\E[S, indn=\E[%p1%dS,
 invis=\E[8m, is2=\E[0u\E[H\E[2J\E[1u\E[7p, kbs=\b,
 kcbt=\E[Z, kcub1=\E[D, kcud1=\E[B, kcuf1=\E[C,
 kcuu1=\E[A, kdch1=\E[P, kdl1=\E[M, kdw=\Ep, kf1=\E@,
 kf10=\EJ, kf11=\EK, kf12=\EL, kf13=\EM, kf14=\EN,
 kf15=\EO, kf16=\EP, kf17=\E0, kf18=\E_, kf19=\Ed,
 kf2=\EA, kf20=\ET, kf21=\EV, kf22=\EX, kf23=\E\s,
 kf24=\E;, kf25=\E", kf26=\E#, kf27=\E$, kf28=\E%,
 kf29=\E&, kf3=\EB, kf30=\E', kf31=\E<, kf32=\E=,
 kf33=\E*, kf34=\E+, kf35=\E\,, kf36=\E-, kf37=\E.,
 kf38=\E/, kf39=\E1, kf4=\EC, kf40=\E2, kf41=\E3,
 kf42=\EU, kf43=\EW, kf44=\EY, kf5=\ED, kf6=\EF,
 kf7=\EG, kf8=\EH, kf9=\EI, khlp=\E>, khome=\E[H,
```

```
kich1=\E[@, kil1=\E[L, kind=\E[T, kiw=\Eo, kmenu=\n,
kmode=\E4, kprt=\Eg, kri=\E[S, krst=\Em, krst=\Em,
lf0=\E>, lf2=^D, nel=\EE, pctrm=USE_TERM:s97801pc:,
rc=\E[u, rev=\E[7m, ri=\E[T, rin=\E[%p1%dT, rmacs=^O,
rmso=\E[0m, rmul=\E[0m, sbt=\E9, sc=\E[s,
sgr=\E[0%?%p1%t;7%;%?%p2%t;4%;%?%p3%t;7%;%?%p4%t;5%;%?%p5%t;\
    2%;%?%p6%t;7%;%?%p7%t;8m%em%;%?%p9%t^N%e^O%;,
sgr0=\E[0m^O, sht=\E:, smacs=^N,
smcup=\E[1;24r\E[m^O\E)w, smso=\E[7m, smul=\E[4m,
tsl=\E[s\E[25;1H,
```

And here is an octal dump of the `term` file, produced by the `od –c /usr/share/lib/terminfo/9/97801` command:

```
0000000 032 001 025  \0   %  \0   !  \0 211 001 324 002   s   t   a   n
0000020   d   a   r   d   |   9   7   8   0   1   |   9   7   8   0   8
0000040  \0  \0 001  \0  \0  \0  \0  \0  \0  \0 001  \0  \0  \0  \0  \0
0000060  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0 001  \0  \0  \0  \0  \0
0000100  \0  \0  \0  \0  \0  \0   P  \0 377 377 030  \0 377 377 377 377
0000120 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0000200 377 377 377 377 377 377 377 377   b  \0   ^  \0   \  \0   v 001
0000220 377 377 221  \0 203  \0 214  \0 377 377 377 377   f  \0   {  \0
0000240 362  \0   P 001   `  \0 377 377   U 001   w  \0 377 377 177  \0
0000260 377 377   b 001 231  \0   <  \0 377 377   3 001 214 001 377 377
0000300   A 001 377 377 207 001 377 377   # 001 377 377   ( 001   7 001
0000320   j 001 377 377   5 001   - 001 377 377 377 377 377 377   < 001
0000340   o 001 377 377 377 377   M  \0 377 377   (  \0 377 377 377 377
0000360   Z 001 241  \0 377 377   @ 002 377 377 377 377 377 377   f 001
0000400 235  \0 225 001 377 377 377 377 377 377 377 377 245 001 300 001
0000420 250 001 253 001 256 001 261 001 264 001 267 001 272 001 275 001
0000440 241 001   ^ 001 245  \0 235 001 377 377 377 377 377 377 231 001
0000460 265  \0 255  \0 377 377 221 001 377 377 377 377   * 002 377 377
0000500 377 377   - 002 377 377 377 377 377 377 377 377 377 377 377 377
0000520 377 377 377 377 377 377 313  \0 377 377 032 001  \0 001 316  \0
0000540 021 001 271  \0 366  \0 340  \0 351  \0 302  \0 327  \0 377 377
0000560 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
0000600 377 377 377 377  \r 001 377 377  \t 001 261  \0 251  \0   O 002
0000620 377 377 377 377   t 001   Q  \0 377 377 377 377 377 377 377 377
0000640 377 377 377 377 377 377 377 377 377 377 377 377 261 002 377 377
0000660   B 002 377 377 377 377 377 377 377 377 377 377 377 377 377 377
0000700 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
0000720 377 377 377 377 377 377 377 377   2 002 377 377 377 377 377 377
0000740 377 377 377 377 377 377 377 377   / 002 377 377 377 377 377 377
0000760 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0001060 377 377 377 377 377 377 377 377 303 001 306 001 311 001 314 001
0001100 317 001 322 001 325 001 330 001 333 001 336 001 341 001 344 001
0001120 347 001 352 001 355 001 360 001 363 001 366 001 371 001 374 001
0001140  \0 002 003 002 006 002  \t 002  \f 002 017 002 022 002 025 002
0001160 030 002 033 002 036 002   ! 002   $ 002   ' 002 377 377 377 377
0001200 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0001600 377 377 377 377 377 377 025  \0 377 377 377 377   ; 002   8 002
0001620   5 002   > 002   F 002   I 002   L 002   s   t   a   n   d   a
0001640   r   d   |   9   7   8   0   1   |   9   7   8   0   8  \0   U
0001660   S   E   _   T   E   R   M   :   s   9   7   8   0   1   p   c
0001700   :  \0 033   [   0   u 033   [   H 033   [   2   J 033   [   1
0001720   u 033   [   7   p  \0 033   [   s 033   [   2   5   ;   1   H
0001740 033   [   K 033   [   u  \0 033   [   u  \0 033   [   s 033   [
```

```
0001760   2   5   ;   1   H  \0  \r  \0 007  \0  \b  \0 033   [   Z  \0
0002000 033   [   %   i   %   p   1   %   d   ;   %   p   2   %   d   H
0002020  \0 033   [   C  \0 033   [   B  \0 033   [   A  \0 033   [   0
0002040   K 033   [   0   m  \0 033   [   0   J  \0 033   [   H 033   [
0002060   2   J  \0 033   [   M  \0 033   [   M  \0 033   [   L  \0 033
0002100   [   L  \0 033   [   T  \0 033   [   S  \0 033   [   S  \0 033
0002120   [   T  \0 033   [   %   p   1   %   d   S  \0 033   [   %   p
0002140   1   %   d   T  \0 033   E  \0 033   [   %   p   1   %   d   B
0002160  \0 033   [   %   p   1   %   d   A  \0 033   [   %   p   1   %
0002200   d   D  \0 033   [   %   p   1   %   d   C  \0 033   [   H  \0
0002220 033   [   %   p   1   %   d   L  \0  \0 033   [   %   p   1   %
0002240   d   M  \0 033   [   s  \0 033   [   u  \0 033   [   %   p   1
0002260   %   d   @  \0 033   [   %   p   1   %   d   P  \0 033   [   8
0002300   m  \0 033   [   7   m  \0 033   [   0   m 017  \0 016  \0 017
0002320  \0 033   [   7   m  \0 033   [   0   m  \0 033   [   1   ;   2
0002340   4   r 033   [   m 017 033   )   w  \0 033   [   6   p  \0 033
0002360   [   7   p  \0 033   [   @  \0 033   [   @  \0 033   [   P  \0
0002400 033   [   P  \0 033   [   4   m  \0 033   [   0   m  \0  \t  \0
0002420 033   [   %   i   %   p   1   %   d   ;   %   p   2   %   d   r
0002440  \0 033   [   2   m  \0 033   [   5   m  \0 033   [   A  \0 033
0002460   [   B  \0 033   [   C  \0 033   [   D  \0 033   [   H  \0 033
0002500   @  \0 033   A  \0 033   B  \0 033   C  \0 033   D  \0 033   F
0002520  \0 033   G  \0 033   H  \0 033   I  \0 033   J  \0 033   K  \0
0002540 033   L  \0 033   M  \0 033   N  \0 033   O  \0 033   P  \0 033
0002560   0  \0 033   _  \0 033   d  \0 033   T  \0 033   V  \0 033   X
0002600  \0 033   `  \0 033   ;  \0 033   "  \0 033   #  \0 033   $  \0
0002620 033   %  \0 033   &  \0 033   '  \0  \0 033   <  \0 033   =  \0
0002640 033   *  \0 033   +  \0 033   ,  \0 033   -  \0 033   .  \0 033
0002660   /  \0 033   1  \0 033   2  \0 033   3  \0 033   U  \0 033   W
0002700  \0 033   Y  \0 033   >  \0 004  \0 033   g  \0 033   >  \0 033
0002720   m  \0 033   9  \0 033   :  \0  \n  \0  \b  \0 033   [   Z  \0
0002740 033   4  \0 033   p  \0 033   o  \0 033   [   0   %   ?   %   p
0002760   1   %   t   ;   7   %   ;   %   ?   %   p   2   %   t   ;   4
0003000   %   ;   %   ?   %   p   3   %   t   ;   7   %   ;   %   ?   %
0003020   p   4   %   t   ;   5   %   ;   %   ?   %   p   5   %   t   ;
0003040   2   %   ;   %   ?   %   p   6   %   t   ;   7   %   ;   %   ?
0003060   %   p   7   %   t   ;   8   m   %   e   m   %   ;   %   ?   %
0003100   p   9   %   t 016   %   e 017   %   ;  \0   +   K   ,   L   .
0003120   N   -   M   f   ?   j   E   k   C   l   B   m   D   n   J   q
0003140   A   t   F   u   G   v   I   w   H   x   @   ~   ,  \0
0003156
```

Some limitations: total compiled entries cannot exceed 4096 bytes; all entries in the name field cannot exceed 128 bytes.

FILES

`/usr/share/lib/terminfo/?/*`
: compiled terminal description database

`/usr/include/term.h`
: `terminfo` header file

SEE ALSO

`infocmp(1M)`, `tic(1M)`, `curses(3X)`, `setupterm(3X)`, `terminfo(4)`, `term(5)`, `term_names(5)`.

Miscellaneous Facilities(5)

curses(5)

NAME

curses - definitions for screen handling and optimisation functions

SYNOPSIS

```
#include <curses.h>
```

DESCRIPTION

Objects

The <curses.h> header provides a declaration for COLOR_PAIRS, COLORS, COLS, curscr, LINES and stdscr.

Constants

The following constants are defined:

EOF Function return value for end-of-file

ERR Function return value for failure

FALSE Boolean false value

OK Function return value for success

TRUE Boolean true value

WEOF Wide-character function return value for end-of-file, as defined in wchar(5) .

The following constant is defined if the implementation supports the indicated revision of the X/Open Curses specification:

_XOPEN_CURSES
: X/Open Curses, Issue 4, Version 2, July 1996, (ISBN: 1-85912-171-3, C610).

Data Types

The following data types are defined through typedef:

attr_t
: An OR-ed set of attributes

bool Boolean data type

chtype
: A character, attributes and a colour-pair

SCREEN
: An opaque terminal representation

wchar_t
: As described in stddef(5)

cchar_t
: As described in wchar(5)

wint_t
: References a string of wide characters

WINDOW
: An opaque window representation

These data types are described in more detail in "X/Open Curses, Issue 4, Version 2, July 1996, (ISBN: 1-85912-171-3, C610), Section 2.4".

The inclusion of <curses.h> may make visible all symbols from the headers stdio(5) , term(5) , termios(5) and wchar(5) .

Attribute Bits

The following symbolic constants are used to manipulate objects of type `attr_t`:

`WA_ALTCHARSET`	Alternate character set
`WA_BLINK`	Blinking
`WA_BOLD`	Extra bright or bold
`WA_DIM`	Half bright
`WA_HORIZONTAL`	Horizontal highlight
`WA_INVIS`	Invisible
`WA_LEFT`	Left highlight
`WA_LOW`	Low highlight
`WA_PROTECT`	Protected
`WA_REVERSE`	Reverse video
`WA_RIGHT`	Right highlight
`WA_STANDOUT`	Best highlighting mode of the terminal
`WA_TOP`	Top highlight
`WA_UNDERLINE`	Underlining
`WA_VERTICAL`	Vertical highlight

These attribute flags shall be distinct.

The following symbolic constants are used to manipulate attribute bits in objects of type `chtype`:

`A_ALTCHARSET`	Alternate character set
`A_BLINK`	Blinking
`A_BOLD`	Extra bright or bold
`A_DIM`	Half bright
`A_INVIS`	Invisible
`A_PROTECT`	Protected
`A_REVERSE`	Reverse video
`A_STANDOUT`	Best highlighting mode of the terminal
`A_UNDERLINE`	Underlining

These attribute flags need not be distinct except when `_XOPEN_CURSES` is defined and the application sets `_XOPEN_SOURCE_EXTENDED` to 1.

The following symbolic constants can be used as bit-masks to extract the components of a `chtype`:

`A_ATTRIBUTES`	Bit-mask to extract attributes
`A_CHARTEXT`	Bit-mask to extract a character
`A_COLOR`	Bit-mask to extract colour-pair information

Line-drawing Constants

The `<curses.h>` header defines the symbolic constants shown in the leftmost two columns of the following table for use in drawing lines. The symbolic constants that begin with `ACS_` are `char` constants. The symbolic constants that begin with `WACS_` are `cchar_t` constants for use with the wide-character interfaces that take a pointer to a `cchar_t`.

In the POSIX locale, the characters shown in the "POSIX Locale Default" column are used when the terminal database does not specify a value using the `acsc` capability as described in `terminfo(4)` , section "Line Graphics".

		POSIX Locale	
char Constant	cchar_t Constant	Default	Glyph Description
ACS_ULCORNER	WACS_ULCORNER	+	upper left-hand corner
ACS_LLCORNER	WACS_LLCORNER	+	lower left-hand corner
ACS_URCORNER	WACS_URCORNER	+	upper right-hand corner
ACS_LRCORNER	WACS_LRCORNER	+	lower right-hand corner
□			
ACS_RTEE	WACS_RTEE	+	right tee (–|)
ACS_LTEE	WACS_LTEE	+	left tee (\z|–)
ACS_BTEE	WACS_BTEE	+	bottom tee ()
ACS_TTEE	WACS_TTEE	+	top tee ()
ACS_HLINE	WACS_HLINE	–	horizontal line
ACS_VLINE	WACS_VLINE	|	vertical line
ACS_PLUS	WACS_PLUS	+	plus
ACS_S1	WACS_S1	-	scan line 1
ACS_S9	WACS_S9		scan line 9
ACS_DIAMOND	WACS_DIAMOND	+	diamond
ACS_CKBOARD	WACS_CKBOARD	:	checker board (stipple)
ACS_DEGREE	WACS_DEGREE	'	degree symbol
ACS_PLMINUS	WACS_PLMINUS	#	plus/minus
ACS_BULLET	WACS_BULLET	o	bullet
ACS_LARROW	WACS_LARROW	<	arrow pointing left
ACS_RARROW	WACS_RARROW	>	arrow pointing right
□			
ACS_DARROW	WACS_DARROW	/	arrow pointing down
ACS_UARROW	WACS_UARROW	[Yuml]	arrow pointing up
ACS_BOARD	WACS_BOARD	#	board of squares
ACS_LANTERN	WACS_LANTERN	#	lantern symbol
ACS_BLOCK	WACS_BLOCK	#	solid square block

Colour-related Macros

The following colour-related macros are defined:

COLOR_BLACK□

□

COLOR_BLUE□

□

COLOR_GREEN□

□
COLOR_CYAN□
□
COLOR_RED□
□
COLOR_MAGENTA□
□
COLOR_YELLOW□
□
COLOR_WHITE

Coordinate-related Macros

The following coordinate-related macros are defined:

void getbegyx(WINDOW *win, int y, int x);□
void getmaxyx(WINDOW *win, int y, int x);□
void getparyx(WINDOW *win, int y, int x);□
void getyx(WINDOW *win, int y, int x);

Key Codes

The following symbolic constants representing function key values are defined:

Key Code	Description
KEY_CODE_YES	Used to indicate that a wchar_t variable contains a key code
KEY_BREAK	Break key
KEY_DOWN	Down arrow key
KEY_UP	Up arrow key
KEY_LEFT	Left arrow key
KEY_RIGHT	Right arrow key
KEY_HOME	Home key
KEY_BACKSPACE	Backspace
KEY_F0	Function keys; space for 64 keys is reserved
KEY_F(*n*)	For 0£|*n*£63
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_EIC	Exit insert char mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page

KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send
KEY_SRESET	Soft (partial) reset
KEY_RESET	Reset or hard reset
KEY_PRINT	Print or copy
KEY_LL	Home down or bottom
KEY_A1	Upper left of keypad
KEY_A3	Upper right of keypad
KEY_B2	Center of keypad
KEY_C1	Lower left of keypad
KEY_C3	Lower right of keypad

The virtual keypad is a 3-by-3 keypad arranged as follows:

A1	UP	A3
LEFT	B2	RIGHT
C1	DOWN	C3

Each legend, such as `A1`, corresponds to a symbolic constant for a key code from the preceding table, such as `KEY_A1`. The following symbolic constants representing function key values are also defined:

Key Code	Description
KEY_BTAB	Back tab key
KEY_BEG	Beginning key
KEY_CANCEL	Cancel key
KEY_CLOSE	Close key
KEY_COMMAND	Cmd (command) key
KEY_COPY	Copy key
KEY_CREATE	Create key
KEY_END	End key
KEY_EXIT	Exit key
KEY_FIND	Find key
KEY_HELP	Help key
KEY_MARK	Mark key
KEY_MESSAGE	Message key
KEY_MOVE	Move key
KEY_NEXT	Next object key
KEY_OPEN	Open key
KEY_OPTIONS	Options key

KEY_PREVIOUS	Previous object key
KEY_REDO	Redo key
KEY_REFERENCE	Reference key
KEY_REFRESH	Refresh key
KEY_REPLACE	Replace key
KEY_RESTART	Restart key
KEY_RESUME	Resume key
KEY_SAVE	Save key
KEY_SBEG	Shifted beginning key
KEY_SCANCEL	Shifted cancel key
KEY_SCOMMAND	Shifted command key
KEY_SCOPY	Shifted copy key
KEY_SCREATE	Shifted create key
KEY_SDC	Shifted delete char key
KEY_SDL	Shifted delete line key
KEY_SELECT	Select key
KEY_SEND	Shifted end key
KEY_SEOL	Shifted clear line key
KEY_SEXIT	Shifted exit key
KEY_SFIND	Shifted find key
KEY_SHELP	Shifted help key
KEY_SHOME	Shifted home key
KEY_SIC	Shifted input key
KEY_SLEFT	Shifted left arrow key
KEY_SMESSAGE	Shifted message key
KEY_SMOVE	Shifted move key
KEY_SNEXT	Shifted next key
KEY_SOPTIONS	Shifted options key
KEY_SPREVIOUS	Shifted prev key
KEY_SPRINT	Shifted print key
KEY_SREDO	Shifted redo key
KEY_SREPLACE	Shifted replace key
KEY_SRIGHT	Shifted right arrow
KEY_SRSUME	Shifted resume key
KEY_SSAVE	Shifted save key
KEY_SSUSPEND	Shifted suspend key
KEY_SUNDO	Shifted undo key
KEY_SUSPEND	Suspend key

KEY_UNDO	Undo key

Function Prototypes

The following are declared as functions, and may also be defined as macros:

```
int  addch(const chtype);□
int  addchnstr(const chtype *, int);□
int  addchstr(const chtype *);□
int  addnstr(const char *, int);□
int  addnwstr(const wchar_t *, int);□
int  addstr(const char *);□
int  add_wch(const cchar_t *);□
int  add_wchnstr(const cchar_t *, int);□
int  add_wchstr(const cchar_t *);□
int  addwstr(const wchar_t *);□
int  attroff(int);□
int  attron(int);□
int  attrset(int);□
int  attr_get(attr_t *, short *, void *);□
int  attr_off(attr_t, void *);□
int  attr_on(attr_t, void *);□
int  attr_set(attr_t, short, void *);□
int  baudrate(void);□
int
 beep(void);□
int  bkgd(chtype);□
void  bkgdset(chtype);□
int  bkgrnd(const cchar_t *);□
void  bkgrndset(const cchar_t *);□
int  border(chtype, chtype, chtype, chtype, chtype, chtype, chtype,□
       chtype);□
int  border_set(const cchar_t *, const cchar_t *, const cchar_t *,□
         const cchar_t *, const cchar_t *, const cchar_t *,□
         const cchar_t *, const cchar_t *);□
int  box(WINDOW *, chtype, chtype);□
int  box_set(WINDOW *, const cchar_t *, const cchar_t *);□
bool  can_change_color(void);□
int  cbreak(void);□
int  chgat(int, attr_t, short, const void *);□
int  clearok(WINDOW *,
bool);□
int  clear(void);□
int  clrtobot(void);□
int  clrtoeol(void);□
int  color_content(short, short *, short *, short *);□
int  COLOR_PAIR(int);□
int  color_set(short,void *);□
int  copywin(const WINDOW *, WINDOW *, int, int, int, int, int, int,□
        int);□
int  curs_set(int);□
int  def_prog_mode(void);□
int  def_shell_mode(void);□
int  delay_output(int);□
int  delch(void);□
```

```
int  deleteln(void);
void  delscreen(SCREEN *);
int  delwin(WINDOW *);
WINDOW *derwin(WINDOW *, int, int, int, int);
int  doupdate(void);
WINDOW *dupwin(WINDOW *);
int  echo(void);
int  echochar(const chtype);
int  echo_wchar(const
cchar_t *);
int  endwin(void);
char  erasechar(void);
int  erase(void);
int  erasewchar(wchar_t *);
void  filter(void);
int  flash(void);
int  flushinp(void);
chtype getbkgd(WINDOW *);
int  getbkgrnd(cchar_t *);
int  getcchar(const cchar_t *, wchar_t *, attr_t *, short *, void *);
int  getch(void);
int  getnstr(char *, int);
int  getn_wstr(wint_t *, int);
int  getstr(char *);
int  get_wch(wint_t *);
WINDOW *getwin(FILE *);
int  get_wstr(wint_t *);
int  halfdelay(int);
bool  has_colors(void);
bool  has_ic(void);
bool  has_il(void);
int
 hline(chtype, int);
int  hline_set(const cchar_t *, int);
void  idcok(WINDOW *, bool);
int  idlok(WINDOW *, bool);
void  immedok(WINDOW *, bool);
chtype inch(void);
int  inchnstr(chtype *, int);
int  inchstr(chtype *);
WINDOW *initscr(void);
int  init_color(short, short, short, short);
int  init_pair(short, short, short);
int  innstr(char *, int);
int  innwstr(wchar_t *, int);
int  insch(chtype);
int  insdelln(int);
int  insertln(void);
int  insnstr(const char *, int);
int  ins_nwstr(const wchar_t *, int);
int  insstr(const char *);
int  instr(char *);
int  ins_wch(const cchar_t *);
int
```

```
ins_wstr(const wchar_t *);
int  intrflush(WINDOW *, bool);
int  in_wch(cchar_t *);
int  in_wchnstr(cchar_t *, int);
int  in_wchstr(cchar_t *);
int  inwstr(wchar_t *);
bool  isendwin(void);
bool  is_linetouched(WINDOW *, int);
bool  is_wintouched(WINDOW *);
char  *keyname(int);
char  *key_name(wchar_t);
int  keypad(WINDOW *, bool);
char  killchar(void);
int  killwchar(wchar_t *);
int  leaveok(WINDOW *, bool);
char  *longname(void);
int  meta(WINDOW *, bool);
int  move(int, int);
int  mvaddch(int, int, const chtype);
int  mvaddchnstr(int, int, const chtype *,
int);
int  mvaddchstr(int, int, const chtype *);
int  mvaddnstr(int, int, const char *, int);
int  mvaddnwstr(int, int, const wchar_t *, int);
int  mvaddstr(int, int, const char *);
int  mvadd_wch(int, int, const cchar_t *);
int  mvadd_wchnstr(int, int, const cchar_t *, int);
int  mvadd_wchstr(int, int, const cchar_t *);
int  mvaddwstr(int, int, const wchar_t *);
int  mvchgat(int, int, int, attr_t, short, const void *);
int  mvcur(int, int, int, int);
int  mvdelch(int, int);
int  mvderwin(WINDOW *, int, int);
int  mvgetch(int, int);
int  mvgetnstr(int, int, char *, int);
int  mvgetn_wstr(int, int, wint_t *, int);
int  mvgetstr(int, int,
char *);
int  mvget_wch(int, int, wint_t *);
int  mvget_wstr(int, int, wint_t *);
int  mvhline(int, int, chtype, int);
int  mvhline_set(int, int, const cchar_t *, int);
chtype mvinch(int, int);
int  mvinchnstr(int, int, chtype *, int);
int  mvinchstr(int, int, chtype *);
int  mvinnstr(int, int, char *, int);
int  mvinnwstr(int, int, wchar_t *, int);
int  mvinsch(int, int, chtype);
int  mvinsnstr(int, int, const char *, int);
int  mvins_nwstr(int, int, const wchar_t *, int);
int  mvinsstr(int, int, const char *);
int  mvinstr(int, int, char *);
int  mvins_wch(int, int, const cchar_t *);
int  mvins_wstr(int, int, const
```

```
wchar_t *);
int  mvin_wch(int, int, cchar_t *);
int  mvin_wchnstr(int, int, cchar_t *, int);
int  mvin_wchstr(int, int, cchar_t *);
int  mvinwstr(int, int, wchar_t *);
int  mvprintw(int, int, char *, ...);
int  mvscanw(int, int, char *, ...);
int  mvvline(int, int, chtype, int);
int  mvvline_set(int, int, const cchar_t *, int);
int  mvwaddch(WINDOW *, int, int, const chtype);
int  mvwaddchnstr(WINDOW *, int, int, const chtype *, int);
int  mvwaddchstr(WINDOW *, int, int, const chtype *);
int  mvwaddnstr(WINDOW *, int, int, const char *, int);
int  mvwaddnwstr(WINDOW *, int, int, const wchar_t *, int);
int
mvwaddstr(WINDOW *, int, int, const char *);
int  mvwadd_wch(WINDOW *, int, int, const cchar_t *);
int  mvwadd_wchnstr(WINDOW *, int, int, const cchar_t *, int);
int  mvwadd_wchstr(WINDOW *, int, int, const cchar_t *);
int  mvwaddwstr(WINDOW *, int, int, const wchar_t *);
int  mvwchgat(WINDOW *, int, int, int, attr_t, short, const void *);
int  mvwdelch(WINDOW *, int, int);
int  mvwgetch(WINDOW *, int, int);
int  mvwgetnstr(WINDOW *, int, int, char *, int);
int  mvwgetn_wstr(WINDOW *, int, int, wint_t *, int);
int  mvwgetstr(WINDOW *, int, int, char *);
int  mvwget_wch(WINDOW *, int, int, wint_t *);
int
 mvwget_wstr(WINDOW *, int, int, wint_t *);
int  mvwhline(WINDOW *, int, int, chtype, int);
int  mvwhline_set(WINDOW *, int, int, const cchar_t *, int);
int  mvwin(WINDOW *, int, int);
chtype mvwinch(WINDOW *, int, int);
int  mvwinchnstr(WINDOW *, int, int, chtype *, int);
int  mvwinchstr(WINDOW *, int, int, chtype *);
int  mvwinnstr(WINDOW *, int, int, char *, int);
int  mvwinnwstr(WINDOW *, int, int, wchar_t *, int);
int  mvwinsch(WINDOW *, int, int, chtype);
int  mvwinsnstr(WINDOW *, int, int, const char *, int);
int  mvwins_nwstr(WINDOW *, int, int, const wchar_t *, int);
int  mvwinsstr(WINDOW *, int, int, const char *);
int
 mvwinstr(WINDOW *, int, int, char *);
int  mvwins_wch(WINDOW *, int, int, const cchar_t *);
int  mvwins_wstr(WINDOW *, int, int, const wchar_t *);
int  mvwin_wch(WINDOW *, int, int, cchar_t *);
int  mvwin_wchnstr(WINDOW *, int, int, cchar_t *, int);
int  mvwin_wchstr(WINDOW *, int, int, cchar_t *);
int  mvwinwstr(WINDOW *, int, int, wchar_t *);
int  mvwprintw(WINDOW *, int, int, char *, ...);
int  mvwscanw(WINDOW *, int, int, char *, ...);
int  mvwvline(WINDOW *, int, int, chtype, int);
int  mvwvline_set(WINDOW *, int, int, const cchar_t *, int);
int  napms(int);
```

```
WINDOW *newpad(int,
int);
SCREEN *newterm(char *, FILE *, FILE *);
WINDOW *newwin(int, int, int, int);
int  nl(void);
int  nocbreak(void);
int  nodelay(WINDOW *, bool);
int  noecho(void);
int  nonl(void);
void  noqiflush(void);
int  noraw(void);
int  notimeout(WINDOW *, bool);
int  overlay(const WINDOW *, WINDOW *);
int  overwrite(const WINDOW *, WINDOW *);
int  pair_content(short, short *, short *);
int  PAIR_NUMBER(int);
int  pechochar(WINDOW *, chtype);
int  pecho_wchar(WINDOW *, const cchar_t*);
int  pnoutrefresh(WINDOW *, int, int, int, int, int, int);
int  prefresh(WINDOW *, int, int, int, int, int, int);
int  printw(char *,
...);
int  putp(const char *);
int  putwin(WINDOW *, FILE *);
void  qiflush(void);
int  raw(void);
int  redrawwin(WINDOW *);
int  refresh(void);
int  reset_prog_mode(void);
int  reset_shell_mode(void);
int  resetty(void);
int  ripoffline(int, int (*)(WINDOW *, int));
int  savetty(void);
int  scanw(char *, ...);
int  scr_dump(const char *);
int  scr_init(const char *);
int  scrl(int);
int  scroll(WINDOW *);
int  scrollok(WINDOW *, bool);
int  scr_restore(const char *);
int  scr_set(const char *);
int  setcchar(cchar_t*, const wchar_t*, const attr_t, short,

 const void*);
int  setscrreg(int, int);
SCREEN *set_term(SCREEN *);
int  setupterm(char *, int, int *);
int  slk_attr_off(const attr_t, void *);
int  slk_attroff(const chtype);
int  slk_attr_on(const attr_t, void *);
int  slk_attron(const chtype);
int  slk_attr_set(const attr_t, short, void *);
int  slk_attrset(const chtype);
int  slk_clear(void);
```

```
int  slk_color(short);
int  slk_init(int);
char  *slk_label(int);
int  slk_noutrefresh(void);
int  slk_refresh(void);
int  slk_restore(void);
int  slk_set(int, const char *, int);
int  slk_touch(void);
int
 slk_wset(int, const wchar_t *, int);
int  standend(void);
int  standout(void);
int  start_color(void);
WINDOW *subpad(WINDOW *, int, int, int, int);
WINDOW *subwin(WINDOW *, int, int, int, int);
int  syncok(WINDOW *, bool);
chtype termattrs(void);
attr_t term_attrs(void);
char  *termname(void);
int  tigetflag(char *);
int  tigetnum(char *);
char  *tigetstr(char *);
void  timeout(int);
int  touchline(WINDOW *, int, int);
int  touchwin(WINDOW *);
char  *tparm(char *, long, long, long, long, long, long, long, long,
       long);
int  typeahead(int);
int  ungetch(int);
int  unget_wch(const wchar_t);
int
 untouchwin(WINDOW *);
void  use_env(bool);
int  vid_attr(attr_t, short, void *);
int  vidattr(chtype);
int  vid_puts(attr_t, short, void *, int (*)(int));
int  vidputs(chtype, int (*)(int));
int  vline(chtype, int);
int  vline_set(const cchar_t *, int);
int  vwprintw(WINDOW *, char *, va_list *);
int  vw_printw(WINDOW *, char *, va_list *);
int  vwscanw(WINDOW *, char *, va_list *);
int  vw_scanw(WINDOW *, char *, va_list *);
int  waddch(WINDOW *, const chtype);
int  waddchnstr(WINDOW *, const chtype *, int);
int  waddchstr(WINDOW *, const chtype *);
int
 waddnstr(WINDOW *, const char *, int);
int  waddnwstr(WINDOW *, const wchar_t *, int);
int  waddstr(WINDOW *, const char *);
int  wadd_wch(WINDOW *, const cchar_t *);
int  wadd_wchnstr(WINDOW *, const cchar_t *, int);
int  wadd_wchstr(WINDOW *, const cchar_t *);
int  waddwstr(WINDOW *, const wchar_t *);
```

```
int  wattroff(WINDOW *, int);
int  wattron(WINDOW *, int);
int  wattrset(WINDOW *, int);
int  wattr_get(WINDOW *, attr_t *, short *, void *);
int  wattr_off(WINDOW *, attr_t, void *);
int  wattr_on(WINDOW *, attr_t, void *);
int
 wattr_set(WINDOW *, attr_t, short, void *);
int  wbkgd(WINDOW *, chtype);
void  wbkgdset(WINDOW *, chtype);
int  wbkgrnd(WINDOW *, const cchar_t *);
void  wbkgrndset(WINDOW *, const cchar_t *);
int  wborder(WINDOW *, chtype, chtype, chtype, chtype, chtype, chtype,
        chtype, chtype);
int  wborder_set(WINDOW *, const cchar_t *, const cchar_t *,
         const cchar_t *, const cchar_t *, const cchar_t *,
         const cchar_t *, const cchar_t *, const cchar_t *);
int  wchgat(WINDOW *, int, attr_t, short, const void *);
int  wclear(WINDOW *);
int  wclrtobot(WINDOW *);
int
 wclrtoeol(WINDOW *);
void  wcursyncup(WINDOW *);
int  wcolor_set(WINDOW *, short, void *);
int  wdelch(WINDOW *);
int  wdeleteln(WINDOW *);
int  wechochar(WINDOW *, const chtype);
int  wecho_wchar(WINDOW *, const cchar_t *);
int  werase(WINDOW *);
int  wgetbkgrnd(WINDOW *, cchar_t *);
int  wgetch(WINDOW *);
int  wgetnstr(WINDOW *, char *, int);
int  wgetn_wstr(WINDOW *, wint_t *, int);
int  wgetstr(WINDOW *, char *);
int  wget_wch(WINDOW *, wint_t *);
int  wget_wstr(WINDOW *, wint_t *);
int  whline(WINDOW *, chtype, int);
int  whline_set(WINDOW *,
const cchar_t *, int);
chtype winch(WINDOW *);
int  winchnstr(WINDOW *, chtype *, int);
int  winchstr(WINDOW *, chtype *);
int  winnstr(WINDOW *, char *, int);
int  winnwstr(WINDOW *, wchar_t *, int);
int  winsch(WINDOW *, chtype);
int  winsdelln(WINDOW *, int);
int  winsertln(WINDOW *);
int  winsnstr(WINDOW *, const char *, int);
int  wins_nwstr(WINDOW *, const wchar_t *, int);
int  winsstr(WINDOW *, const char *);
int  winstr(WINDOW *, char *);
int  wins_wch(WINDOW *, const cchar_t *);
int  wins_wstr(WINDOW *, const wchar_t *);
int  win_wch(WINDOW *,
```

```
cchar_t *);
int  win_wchnstr(WINDOW *, cchar_t *, int);
int  win_wchstr(WINDOW *, cchar_t *);
int  winwstr(WINDOW *, wchar_t *);
int  wmove(WINDOW *, int, int);
int  wnoutrefresh(WINDOW *);
int  wprintw(WINDOW *, char *, ...);
int  wredrawln(WINDOW *, int, int);
int  wrefresh(WINDOW *);
int  wscanw(WINDOW *, char *, ...);
int  wscrl(WINDOW *, int);
int  wsetscrreg(WINDOW *, int, int);
int  wstandend(WINDOW *);
int  wstandout(WINDOW *);
void  wsyncup(WINDOW *);
void  wsyncdown(WINDOW *);
void  wtimeout(WINDOW *, int);
int  wtouchln(WINDOW *, int, int, int);
wchar_t *wunctrl(cchar_t *);
int
 wvline(WINDOW *, chtype, int);
int  wvline_set(WINDOW *, const cchar_t *, int);
```

SEE ALSO

terminfo(4), stdio(5), term(5), termios(5), unctrl(5), wchar(5).

Character User Interface Reference Manual

term(5)

NAME

`term` - terminal capabilities

SYNOPSIS

```
#include <term.h>
```

DESCRIPTION

The following data type is defined through `typedef`:

`TERMINAL`
: An opaque representation of the capabilities for a single terminal from the `terminfo` database.

The `<term.h>` header provides a declaration for the following object: `cur_term`. It represents the current terminal record from the `terminfo` database that the application has selected by calling `set_curterm()`.

The `<term.h>` header contains the variable names listed in the **Variable** column in `terminfo(4)`.

The following are declared as functions, and may also be defined as macros:

```
int  del_curterm(TERMINAL *);
int  putp(const char *);
int  restartterm(char *, int, int *);
TERMINAL *set_curterm(TERMINAL *);
int  setupterm(char *, int, int *);
int  tgetent(char *, const char *);
int  tgetflag(char *);
int  tgetnum(char *);
char *tgetstr(char *, char **):
char *tgoto(char *, int, int);
int  tigetflag(char *);
int  tigetnum(char *);
char *tigetstr(char *);
char *tparm(char *,long, long, long, long, long, long, long, long, long);
int  tputs(const char *, int, int (*)(int));
```

The `<term.h>` header defines the following data type through `typedef`:

`bool` As described in `curses(5)`.

SEE ALSO

`printf(3S)`, `putp(3X)`, `set_curterm(3X)`, `tgetent(3X)`, `tigetflag(3X)`, `terminfo(4)`, `curses(5)`.

Character User Interface Reference Manual

term_names(5)

NAME

`term_names` - conventional names for terminals

DESCRIPTION

Terminal names are maintained as part of the shell environment in the environment variable `TERM` [see `sh(1)`, `profile(4)`, and `environ(5)`]. These names are used by certain commands [for example, `tabs`, `tput`, and `vi`] and certain functions [for example, see `curses(3X)`].

Files under `/usr/share/lib/terminfo` are used to name terminals and describe their capabilities. These files are in the format described in `terminfo(4)`. Entries in `terminfo` source files consist of a number of comma-separated fields. To print a description of a terminal *term*, use the command `infocmp -I` *term* [see `infocmp(1M)`]. White space after each comma is ignored. The first line of each terminal description in the `terminfo` database gives the names by which `terminfo` knows the terminal, separated by bar (|) characters. The first name given is the most common abbreviation for the terminal [this is the one to use to set the environment variable `TERMINFO` in `$HOME/.profile`; see `profile(4)`], the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should contain no blanks and must be unique in the first 14 characters; the last name may contain blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, for example, for the AT&T 4425 terminal, `att4425`. This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Up to 8 characters, chosen from the set `a` through `z` and `0` through `9`, make up a basic terminal name. Names should generally be based on original vendors rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name. Terminal sub-models, operational modes that the hardware can be in, or user preferences should be indicated by appending a hyphen and an indicator of the mode. Thus, an AT&T 4425 terminal in 132 column mode is `att4425-w`. The following suffixes should be used where possible:

Suffix	Meaning	Example
`-w`	Wide mode (more than 80 columns)	`att4425-w`
`-am`	With auto. margins (usually default)	`vt100-am`
`-nam`	Without automatic margins	`vt100-nam`
-n	Number of lines on the screen	`aaa-60`
`-na`	No arrow keys (leave them in local)	`c100-na`
`-np`	Number of pages of memory	`c100-4p`
`-rv`	Reverse video	`att4415-rv`

To avoid conflicts with the naming conventions used in describing the different modes of a terminal (e.g., `-w`), it is recommended that a terminal's root name not contain hyphens. Further, it is good practice to make all terminal names used in the `terminfo(4)` database unique. Terminal entries that are present only for inclusion in other entries via the `use=` facilities should have a "`+`" in their name, as in `4415+nl`.

Here are some of the known terminal names: (For a complete list, enter the command `ls -C /usr/share/lib/terminfo/?`.)

`2621`, `hp2621`	Hewlett-Packard 2621 series
`2631`	Hewlett-Packard 2631 line printer

2631-c	Hewlett-Packard 2631 line printer, compressed mode
2631-e	Hewlett-Packard 2631 line printer, expanded mode
2640, hp2640	Hewlett-Packard 2640 series
2645, hp2645	Hewlett-Packard 2645 series
3270	IBM Model 3270
33, tty33	AT&T Teletype Model 33 KSR
35, tty35	AT&T Teletype Model 35 KSR
37, tty37	AT&T Teletype Model 37 KSR
4000a	Trendata 4000a
4014, tek4014	TEKTRONIX 4014
40, tty40	AT&T Teletype Dataspeed 40/2
43, tty43	AT&T Teletype Model 43 KSR
4410, 5410	AT&T 4410/5410 in 80-column mode, version 2
4410-nfk, 5410-nfk	AT&T 4410/5410 without function keys, version 1
4410-nsl, 5410-nsl	AT&T 4410/5410 without pln defined
4410-w, 5410-w	AT&T 4410/5410 in 132-column mode
4410v1, 5410v1	AT&T 4410/5410 in 80-column mode, version 1
4410v1-w, 5410v1-w	AT&T 4410/5410 in 132-column mode, version 1
4415, 5420	AT&T 4415/5420 in 80-column mode
4415-nl, 5420-nl	AT&T 4415/5420 without changing labels
4415-rv, 5420-rv	AT&T 4415/5420 80 columns in reverse video
4415-rv-nl, 5420-rv-nl	AT&T 4415/5420 reverse video without changing labels
4415-w, 5420-w	AT&T 4415/5420 in 132-column mode
4415-w-nl, 5420-w-nl	AT&T 4415/5420 in 132-column mode without changing labels
4415-w-rv, 5420-w-rv	AT&T 4415/5420 132 columns in reverse video
4418, 5418	AT&T 5418 in 80-column mode
4418-w, 5418-w	AT&T 5418 in 132-column mode
4420	AT&T Teletype Model 4420
4424	AT&T Teletype Model 4424
4424-2	AT&T Teletype Model 4424 in display function group ii
4425, 5425	AT&T 4425/5425
4425-fk, 5425-fk	AT&T 4425/5425 without function keys
4425-nl, 5425-nl	AT&T 4425/5425 without changing labels in 80-column mode
4425-w, 5425-w	AT&T 4425/5425 in 132-column mode
4425-w-fk, 5425-w-fk	AT&T 4425/5425 without function keys in 132-column mode
4425-nl-w, 5425-nl-w	AT&T 4425/5425 without changing labels in 132-column mode
4426	AT&T Teletype Model 4426S

`450`	DASI 450 (same as Diablo 1620)
`450-12`	DASI 450 in 12-pitch mode
`500`, `att500`	AT&T-IS 500 terminal
`510`, `510a`	AT&T 510/510a in 80-column mode
`513bct`, `att513`	AT&T 513 bct terminal
`5320`	AT&T 5320 hardcopy terminal
`5420_2`	AT&T 5420 model 2 in 80-column mode
`5420_2-w`	AT&T 5420 model 2 in 132-column mode
`5620`, `dmd`	AT&T 5620 terminal 88 columns
`5620-24`, `dmd-24`	AT&T Teletype Model DMD 5620 in a 24x80 layer
`5620-34`, `dmd-34`	AT&T Teletype Model DMD 5620 in a 34x80 layer
`610`, `610bct`	AT&T 610 bct terminal in 80-column mode
`610-w`, `610bct-w`	AT&T 610 bct terminal in 132-column mode
`630`, `630MTG`	AT&T 630 Multi-Tasking Graphics terminal
`7300`, `pc7300`, `unix_pc`	AT&T UNIX PC Model 7300
`735`, `ti`	Texas Instruments TI735 and TI725
`745`	Texas Instruments TI745
`97801`, `97807`	Siemens Nixdorf Informationssysteme
`dumb`	generic name for terminals that lack reverse line-feed and other special escape sequences
`hp`	Hewlett-Packard (same as 2645)
`lp`	generic name for a line printer
`pt505`	AT&T Personal Terminal 505 (22 lines)
`pt505-24`	AT&T Personal Terminal 505 (24-line mode)
`sync`	generic name for synchronous Teletype Model 4540-compatible terminals

Commands whose behavior depends on the type of terminal should accept arguments of the form `-T`*term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable `TERM`, which, in turn, should contain *term*.

FILES

`/usr/share/lib/terminfo/?/*`
compiled terminal description database

SEE ALSO

`sh`(1), `stty`(1), `tabs`(1), `tput`(1), `vi`(1), `infocmp`(1M), `curses`(3X), `profile`(4), `term`(4), `terminfo`(4), `environ`(5).

Character User Interface Reference Manual

unctrl(5)

NAME

`unctrl` - definitions for unctrl()

DESCRIPTION

The `<unctrl.h>` header defines the `chtype` type as defined in `curses(5)`.

The following is declared as a function, and may also be defined as a macro:

```
char *unctrl(chtype);
```

SEE ALSO

`unctrl(3X)`, `curses(5)`.