NAME

editline, **el_init**, **el_init_fd**, **el_end**, **el_reset**, **el_gets**, **el_wgets**, **el_getc**, **el_wgetc**, **el_push**, **el_wpush**, **el_parse**, **el_wparse**, **el_set**, **el_wset**, **el_get**, **el_wget**, **el_source**, **el_resize**, **el_cursor**, **el_line**, **el_wline**, **el_insertstr**, **el_winsertstr**, **el_deletestr**, **el_wdeletestr**, **history_init**, **history_winit**, **history_end**, **history_wend**, **history**, **history_w**, **tok_init**, **tok_winit**, **tok_end**, **tok_wend**, **tok_reset**, **tok_wreset**, **tok_line**, **tok_wline**, **tok_str**, **tok_wstr** - line editor, history and tokenization functions

LIBRARY

Command Line Editor Library (libedit, -ledit)

SYNOPSIS

#include <histedit.h>

*EditLine **
el_init(*const char *prog, FILE *fin, FILE *fout, FILE *ferr*);

*EditLine **
el_init_fd(*const char *prog, FILE *fin, FILE *fout, FILE *ferr, int fdin, int fdout, int fderr*);

void
el_end(*EditLine *e*);

void
el_reset(*EditLine *e*);

*const char **
el_gets(*EditLine *e, int *count*);

*const wchar_t **
el_wgets(*EditLine *e, int *count*);

int
el_getc(*EditLine *e, char *ch*);

int
el_wgetc(*EditLine *e, wchar_t *wc*);

void
el_push(*EditLine *e, const char *mbs*);

void
el_wpush(*EditLine *e, const wchar_t *wcs*);

int
el_parse(*EditLine *e, int argc, const char *argv[]*);

int
el_wparse(*EditLine *e, int argc, const wchar_t *argv[]*);

int
el_set(*EditLine *e, int op, ...*);

int
el_wset(*EditLine *e, int op, ...*);

int
el_get(*EditLine *e, int op, ...*);

int
el_wget(*EditLine *e, int op, ...*);

int
el_source(*EditLine *e, const char *file*);

void
el_resize(*EditLine *e*);

int
el_cursor(*EditLine *e, int count*);

*const LineInfo **
el_line(*EditLine *e*);

*const LineInfoW **
el_wline(*EditLine *e*);

int
el_insertstr(*EditLine *e, const char *str*);

int

el_winsertstr(*EditLine *e, const wchar_t *str*);

void
el_deletestr(*EditLine *e, int count*);

void
el_wdeletestr(*EditLine *e, int count*);

*History **
history_init(void);

*HistoryW **
history_winit(void);

void
history_end(*History *h*);

void
history_wend(*HistoryW *h*);

int
history(*History *h, HistEvent *ev, int op, ...*);

int
history_w(*HistoryW *h, HistEventW *ev, int op, ...*);

*Tokenizer **
tok_init(*const char *IFS*);

*TokenizerW **
tok_winit(*const wchar_t *IFS*);

void
tok_end(*Tokenizer *t*);

void
tok_wend(*TokenizerW *t*);

void
tok_reset(*Tokenizer *t*);

void
tok_wreset(*TokenizerW *t*);

int
tok_line(*Tokenizer *t, const LineInfo *li, int *argc, const char **argv[], int *cursorc, int *cursoro*);

int
tok_wline(*TokenizerW *t, const LineInfoW *li, int *argc, const wchar_t **argv[], int *cursorc, int *cursoro*);

int
tok_str(*Tokenizer *t, const char *str, int *argc, const char **argv[]*);

int
tok_wstr(*TokenizerW *t, const wchar_t *str, int *argc, const wchar_t **argv[]*);

DESCRIPTION

The **editline** library provides generic line editing, history and tokenization functions, similar to those found in **sh**(1).

These functions are available in the **libedit** library (which needs the **libtermcap** library). Programs should be linked with **-ledit** ltermcap .

The **editline** library respects the *LC_CTYPE* locale set by the application program and never uses **setlocale**(3) to change the locale.

LINE EDITING FUNCTIONS

The line editing functions use a common data structure, *EditLine*, which is created by **el_init**() or **el_init_fd**() and freed by **el_end**().

The wide-character functions behave the same way as their narrow counterparts.

The following functions are available:

el_init()
Initialize the line editor, and return a data structure to be used by all other line editing functions, or NULL on failure. *prog* is the name of the invoking program, used when reading the **editrc**(5) file to determine which settings to use. *fin*, *fout* and *ferr* are the input, output, and error streams (respectively) to use. In this documentation, references to ‘‘the tty’’ are actually to this input/output stream combination.

el_init_fd()
Like **el_init**() but allows specifying file descriptors for the **stdio**(3) corresponding streams, in case those were created with **funopen**(3).

el_end()
Clean up and finish with *e*, assumed to have been created with **el_init**() or **el_init_fd**().

el_reset()
Reset the tty and the parser. This should be called after an error which may have upset the tty's state.

el_gets()
Read a line from the tty. *count* is modified to contain the number of characters read. Returns the line read if successful, or NULL if no characters were read or if an error occurred. If an error occurred, *count* is set to -1 and errno contains the error code that caused it. The return value may not remain valid across calls to **el_gets**() and must be copied if the data is to be retained.

el_wgetc()
Read a wide character from the tty, respecting the current locale, or from the input queue described in **editline**(7) if that is not empty, and store it in *wc*. If an invalid or incomplete character is found, it is discarded, *errno* is set to Er EILSEQ , and the next character is read and stored in *wc*. Returns 1 if a valid character was read, 0 on end of file, or -1 on **read**(2) failure. In the latter case, *errno* is set to indicate the error.

el_getc()
Read a wide character as described for **el_wgetc**() and return 0 on end of file or -1 on failure. If the wide character can be represented as a single-byte character, convert it with **wctob**(3), store the result in *ch*, and return 1; otherwise, set *errno* to Er ERANGE and return -1. In the C or POSIX locale, this simply reads a byte, but for any other locale, including UTF-8, this is rarely useful.

el_wpush()
Push the wide character string *wcs* back onto the input queue described in **editline**(7). If the queue overflows, for example due to a recursive macro, or if an error occurs, for example because *wcs* is NULL or memory allocation fails, the function beeps at the user, but does not report the problem to the caller.

el_push()
Use the current locale to convert the multibyte string *mbs* to a wide character string, and pass the result to **el_wpush**().

el_parse()

Parses the *argv* array (which is *argc* elements in size) to execute builtin **editline** commands. If the command is prefixed with ‘‘prog :’’ then **el_parse**() will only execute the command if ‘‘prog’’ matches the *prog* argument supplied to **el_init**(). The return value is -1 if the command is unknown, 0 if there was no error or ‘‘prog’’ didn’t match, or 1 if the command returned an error. Refer to **editrc**(5) for more information.

el_set()

Set **editline** parameters. *op* determines which parameter to set, and each operation has its own parameter list. Returns 0 on success, -1 on failure.

The following values for *op* are supported, along with the required argument list:

EL_PROMPT , *char *(*f)(EditLine *)*

Define prompt printing function as *f*, which is to return a string that contains the prompt.

EL_PROMPT_ESC , *char *(*f)(EditLine *), Fa char c*

Same as EL_PROMPT , but the *c* argument indicates the start/stop literal prompt character.

If a start/stop literal character is found in the prompt, the character itself is not printed, but characters after it are printed directly to the terminal without affecting the state of the current line. A subsequent second start/stop literal character ends this behavior. This is typically used to embed literal escape sequences that change the color/style of the terminal in the prompt. Note that the literal escape character cannot be the last character in the prompt, as the escape sequence is attached to the next character in the prompt. 0 unsets it.

EL_REFRESH

Re-display the current line on the next terminal line.

EL_RPROMPT , *char *(*f)(EditLine *)*

Define right side prompt printing function as *f*, which is to return a string that contains the prompt.

EL_RPROMPT_ESC , *char *(*f)(EditLine *), Fa char c*

Define the right prompt printing function but with a literal escape character.

EL_TERMINAL , *const char *type*

Define terminal type of the tty to be *type*, or to *TERM* if *type* is NULL .

EL_EDITOR , *const char *mode*

Set editing mode to *mode*, which must be one of ‘‘emacs’’ or ‘‘vi’’.

EL_SIGNAL , *int flag*

If *flag* is non-zero, **editline** will install its own signal handler for the following signals when reading command input: SIGCONT , SIGHUP , SIGINT , SIGQUIT , SIGSTOP , SIGTERM , SIGTSTP , and SIGWINCH . Otherwise, the current signal handlers will be used.

EL_BIND , *const char *, Fa ..., Dv NULL*

Perform the **bind** builtin command. Refer to **editrc**(5) for more information.

EL_ECHOTC , *const char *, Fa ..., Dv NULL*

Perform the **echotc** builtin command. Refer to **editrc**(5) for more information.

EL_SETTC , *const char *, Fa ..., Dv NULL*

Perform the **settc** builtin command. Refer to **editrc**(5) for more information.

EL_SETTY , *const char *, Fa ..., Dv NULL*

Perform the **setty** builtin command. Refer to **editrc**(5) for more information.

EL_TELLTC , *const char *, Fa ..., Dv NULL*

Perform the **telltc** builtin command. Refer to **editrc**(5) for more information.

EL_ADDFN , *const char *name, Fa const char *help,*

Fa "unsigned char (*func)(EditLine *e, int ch)" Add a user defined function, **func**(), referred to as *name* which is invoked when a key which is bound to *name* is entered. *help* is a description of *name*. At invocation time, *ch* is the key which caused the invocation. The return value of **func**() should be one of:

CC_NORM

Add a normal character.

CC_NEWLINE

End of line was entered.

CC_EOF

EOF was entered.

CC_ARGHACK

Expecting further command input as arguments, do nothing visually.

CC_REFRESH
Refresh display.

CC_REFRESH_BEEP
Refresh display, and beep.

CC_CURSOR
Cursor moved, so update and perform CC_REFRESH .

CC_REDISPLAY
Redisplay entire input line. This is useful if a key binding outputs extra information.

CC_ERROR
An error occurred. Beep, and flush tty.

CC_FATAL
Fatal error, reset tty to known state.

EL_HIST , *History *(*func)(History *, int op, ...),*
Fa "const char *ptr" Defines which history function to use, which is usually **history**(). *ptr* should be the value returned by **history_init**().

EL_EDITMODE , *int flag*
If *flag* is non-zero, editing is enabled (the default). Note that this is only an indication, and does not affect the operation of **.** At this time, it is the caller’s responsibility to check this (using **el_get**()) to determine if editing should be enabled or not.

EL_UNBUFFERED , *int flag*
If *flag* is zero, unbuffered mode is disabled (the default). In unbuffered mode, **el_gets**() will return immediately after processing a single character.

EL_SAFEREAD , *int flag*
If the *flag* argument is non-zero, then **editline** attempts to recover from read errors, ignoring the first interrupted error, and trying to reset the input file descriptor to reset non-blocking I/O. This is disabled by default, and desirable only when **editline** is used in shell-like applications.

EL_GETCFN , *el_rfunc_t f*
Whenever reading a character, use the function -ragged -offset indent -compact

int

Fo f *EditLine *e wchar_t *wc* Fc which stores the character in *wc* and returns 1 on success, 0 on end of file, or -1 on I/O or encoding errors. Functions internally using it include **el_wgets**(), **el_wgetc**(), **el_gets**(), and **el_getc**(). Initially, a builtin function is installed, and replacing it is discouraged because writing such a function is very error prone. The builtin function can be restored at any time by passing the special value EL_BUILTIN_GETCFN instead of a function pointer.

EL_CLIENTDATA , *void *data*

Register *data* to be associated with this EditLine structure. It can be retrieved with the corresponding **el_get**(); call.

EL_SETFP , *int fd, Fa FILE *fp*

Set the current **editline** file pointer for ‘‘input’’ *fd* = 0 , ‘‘output’’ *fd* = 1 , or ‘‘error’’ *fd* = 2 from *fp*.

el_get()

Get **editline** parameters. *op* determines which parameter to retrieve into *result*. Returns 0 if successful, -1 otherwise.

The following values for *op* are supported, along with actual type of *result :*

EL_PROMPT , *char *(*f)(EditLine *), Fa char *c*

Set *f* to a pointer to the function that displays the prompt. If *c* is not NULL , set it to the start/stop literal prompt character.

EL_RPROMPT , *char *(*f)(EditLine *), Fa char *c*

Set *f* to a pointer to the function that displays the prompt. If *c* is not NULL , set it to the start/stop literal prompt character.

EL_EDITOR , *const char **n*

Set the name of the editor in *n*, which will be one of ‘‘emacs’’ or ‘‘vi’’.

EL_GETTC , *const char *name, Fa void *value*

If *name* is a valid **termcap**(5) capability set *value* to the current value of that capability.

EL_SIGNAL , *int *s*

Set *s* to non-zero if **editline** has installed private signal handlers (see **el_get**() above).

EL_EDITMODE , *int *c*

Set *c* to non-zero if editing is enabled.

EL_GETCFN , *el_rfunc_t *f*
Set *f* to a pointer to the function that reads characters, or to EL_BUILTIN_GETCFN if the builtin function is in use.

EL_CLIENTDATA , *void **data*
Set *data* to the previously registered client data set by an **el_set**() call.

EL_UNBUFFERED , *int *c*
Set *c* to non-zero if unbuffered mode is enabled.

EL_SAFEREAD , *int *c*
Set *c* to non-zero if safe read is set.

EL_GETFP , *int fd", Fa FILE **fp*
Set *fp* to the current **editline** file pointer for ‘‘input’’ *fd* = 0 , ‘‘output’’ *fd* = 1 , or ‘‘error’’ *fd* = 2 .

el_source()
Initialize **editline** by reading the contents of *file*. **el_parse**() is called for each line in *file*. If *file* is NULL , try *$EDITRC* and if that is not set *$HOME/.editrc*. Refer to **editrc**(5) for details on the format of *file*. **el_source**() returns 0 on success and -1 on error.

el_resize()
Must be called if the terminal size changes. If EL_SIGNAL has been set with **el_set**(), then this is done automatically. Otherwise, it’s the responsibility of the application to call **el_resize**() on the appropriate occasions.

el_cursor()
Move the cursor to the right (if positive) or to the left (if negative) *count* characters. Returns the resulting offset of the cursor from the beginning of the line.

el_line()
Return the editing information for the current line in a *LineInfo* structure, which is defined as follows:

```
typedef struct lineinfo {
   const char *buffer;    /* address of buffer */
   const char *cursor;    /* address of cursor */
   const char *lastchar;  /* address of last character */
} LineInfo;
```

buffer is not NUL terminated. This function may be called after **el_gets**() to obtain the *LineInfo* structure pertaining to line returned by that function, and from within user defined functions added with EL_ADDFN .

el_insertstr()
Insert *str* into the line at the cursor. Returns -1 if *str* is empty or won't fit, and 0 otherwise.

el_deletestr()
Delete *count* characters before the cursor.

HISTORY LIST FUNCTIONS

The history functions use a common data structure, *History*, which is created by **history_init**() and freed by **history_end**().

The following functions are available:

history_init()
Initialize the history list, and return a data structure to be used by all other history list functions, or NULL on failure.

history_end()
Clean up and finish with *h*, assumed to have been created with **history_init**().

history()
Perform operation *op* on the history list, with optional arguments as needed by the operation. *ev* is changed accordingly to operation. The following values for *op* are supported, along with the required argument list:

H_SETSIZE , *int size*
Set size of history to *size* elements.

H_GETSIZE
Get number of events currently in history.

H_END
Cleans up and finishes with *h*, assumed to be created with **history_init**().

H_CLEAR
Clear the history.

H_FUNC , *void *ptr, Fa history_gfun_t first,*
Fa "history_gfun_t next" , Fa "history_gfun_t last" , Fa "history_gfun_t prev" , Fa "history_gfun_t curr" , Fa "history_sfun_t set" , Fa "history_vfun_t clear" , Fa "history_efun_t enter" , Fa "history_efun_t add" Define functions to perform various history operations. *ptr* is the argument given to a function when it's invoked.

H_FIRST
Return the first element in the history.

H_LAST
Return the last element in the history.

H_PREV
Return the previous element in the history. It is newer than the current one.

H_NEXT
Return the next element in the history. It is older than the current one.

H_CURR
Return the current element in the history.

H_SET , *int position*
Set the cursor to point to the requested element.

H_ADD , *const char *str*
Append *str* to the current element of the history, or perform the H_ENTER operation with argument *str* if there is no current element.

H_APPEND , *const char *str*
Append *str* to the last new element of the history.

H_ENTER , *const char *str*
Add *str* as a new element to the history and, if necessary, removing the oldest entry to keep the list to the created size. If H_SETUNIQUE has been called with a non-zero argument, the element will not be entered into the history if its contents match the ones of the current history element. If the element is entered **history**() returns 1; if it is ignored as a duplicate returns 0. Finally **history**() returns -1 if an error occurred.

H_PREV_STR , *const char *str*
Return the closest previous event that starts with *str*.

H_NEXT_STR , *const char *str*
Return the closest next event that starts with *str*.

H_PREV_EVENT , *int e*
Return the previous event numbered *e*.

H_NEXT_EVENT , *int e*
Return the next event numbered *e*.

H_LOAD , *const char *file*
Load the history list stored in *file*.

H_SAVE , *const char *file*
Save the history list to *file*.

H_SAVE_FP , *FILE *fp*
Save the history list to the opened

FILE
pointer *fp*.

H_NSAVE_FP , *size_t n, Fa FILE *fp*
Save the last

n
history entries to the opened

FILE
pointer *fp*.

H_SETUNIQUE , *int unique*
Set flag that adjacent identical event strings should not be entered into the history.

H_GETUNIQUE
Retrieve the current setting if adjacent identical elements should be entered into the history.

H_DEL , *int e*
Delete the event numbered *e*. This function is only provided for **readline** compatibility. The caller is responsible for free'ing the string in the returned *HistEvent*.

history(); returns >= 0 if the operation *op* succeeds. Otherwise, -1 is returned and *ev* is updated to contain more details about the error.

TOKENIZATION FUNCTIONS

The tokenization functions use a common data structure, *Tokenizer*, which is created by **tok_init**() and freed by **tok_end**().

The following functions are available:

tok_init()

Initialize the tokenizer, and return a data structure to be used by all other tokenizer functions. *IFS* contains the Input Field Separators, which defaults to <space ,> <tab ,> and <newline> if NULL .

tok_end()

Clean up and finish with *t*, assumed to have been created with **tok_init**().

tok_reset()

Reset the tokenizer state. Use after a line has been successfully tokenized by **tok_line**() or **tok_str**() and before a new line is to be tokenized.

tok_line()

Tokenize *li*, If successful, modify: *argv* to contain the words, *argc* to contain the number of words, *cursorc* (if not NULL) to contain the index of the word containing the cursor, and *cursoro* (if not NULL) to contain the offset within *argv[cursorc]* of the cursor.

Returns 0 if successful, -1 for an internal error, 1 for an unmatched single quote, 2 for an unmatched double quote, and 3 for a backslash quoted <newline .> A positive exit code indicates that another line should be read and tokenization attempted again.

tok_str()

A simpler form of **tok_line**(;) *str* is a NUL terminated string to tokenize.

SEE ALSO

sh(1), **signal**(3), **termcap**(3), **editrc**(5), **termcap**(5), **editline**(7)

HISTORY

The **editline** library first appeared in Bx 4.4 . CC_REDISPLAY appeared in Nx 1.3 . CC_REFRESH_BEEP , EL_EDITMODE and the readline emulation appeared in Nx 1.4 . EL_RPROMPT appeared in Nx 1.5 .

AUTHORS

-nosplit The **editline** library was written by

Christos Zoulas .

Luke Mewburn wrote this manual and implemented CC_REDISPLAY , CC_REFRESH_BEEP , EL_EDITMODE , and EL_RPROMPT .

Jaromir Dolecek implemented the readline emulation.

Johny Mattsson implemented wide-character support.

BUGS

At this time, it is the responsibility of the caller to check the result of the EL_EDITMODE operation of **el_get**() (after an **el_source**() or **el_parse**()) to determine if **editline** should be used for further input. I.e., EL_EDITMODE is purely an indication of the result of the most recent **editrc**(5) **edit** command.