

**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 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 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 \*)*, *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 \*)*, *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 \**, ..., NULL

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

**EL\_ECHOTC**, *const char \**, ..., NULL

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

**EL\_SETTC**, *const char \**, ..., NULL

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

**EL\_SETTY**, *const char \**, ..., NULL

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

**EL\_TELLTC**, *const char \**, ..., NULL

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

**EL\_ADDFN**, *const char \*name, const char \*help, 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, ...), 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 **editline**. 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\_tf*

Whenever reading a character, use the function

*int f(EditLine \*e, wchar\_t \*wc)*

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*, *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 \*)*, *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 \*)*, *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*, *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, 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, history\_gfun\_t first, history\_gfun\_t next, history\_gfun\_t last, history\_gfun\_t prev, history\_gfun\_t curr, history\_sfun\_t set, history\_vfun\_t clear, history\_efun\_t enter, 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*, *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  $\geq 0$  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 4.4BSD. CC\_REDISPLAY appeared in NetBSD 1.3. CC\_REFRESH\_BEEP, EL\_EDITMODE and the readline emulation appeared in NetBSD 1.4. EL\_RPROMPT appeared in NetBSD 1.5.

## **AUTHORS**

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. Johnny 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.