NAME

editline, el_init, el_end, el_reset, el_gets, el_getc, el_push, el_parse, el_set, el_get, el_source, el_resize, el_line, el_insertstr, el_deletestr, history_init, history_end, history, tok_init, tok_end, tok_reset, tok_line, tok_str -- 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);

void el_end(EditLine *e);

void el_reset(EditLine *e);

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

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

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

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

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

int el_get(EditLine *e, int op, void *result);

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

void el_resize(EditLine *e);

const LineInfo * el_line(EditLine *e);

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

void el_deletestr(EditLine *e, int count);

History * history_init();

void history_end(History *h);

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

Tokenizer * tok_init(const char *IFS);

void tok_end(Tokenizer *t);

void tok_reset(Tokenizer *t);

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

int tok_str(Tokenizer *t, const char *str, int *argc, const char **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 libcurses library). Programs should be linked with -ledit -lcurses.

LINE EDITING FUNCTIONS

The line editing functions use a common data structure, EditLine, which is created by el_init() and freed by el_end(). The following functions are available:

el_init()
Initialise the line editor, and return a data structure to be used by all other line editing functions. prog is the name of 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_end
Clean up and finish with e, assumed to have been created with el_init().

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.

el_getc()
Read a character from the tty. ch is modified to contain the character read. Returns the number of characters read if successful, -1 otherwise.

el_push()
Pushes str back onto the input stream. This is used by the macro expansion mechanism. Refer to the description of bind -s in editrc(5) for more information.

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.

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

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_GETCFN, int (*f)(EditLine *, char *c)
Define the character reading function as f, which is to return the number of characters read and store them in c. This function is called internally by el_gets() and el_getc(). The builtin function can be set or restored with the special function name ``EL_BUILTIN_GETCFN".

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

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 *)
Return a pointer to the function that displays the prompt.

EL_RPROMPT, char *(*f)(EditLine *)
Return a pointer to the function that displays the rightside prompt.

EL_EDITOR, const char *
Return the name of the editor, which will be one of ``emacs" or ``vi".

EL_SIGNAL, int *
Return non-zero if editline has installed private signal handlers (see el_get() above).

EL_EDITMODE, int *
Return non-zero if editing is enabled.

EL_GETCFN, int (**f)(EditLine *, char *)
Return a pointer to the function that read characters, which is equal to ``EL_BUILTIN_GETCFN" in the case of the default builtin function.

EL_CLIENTDATA, void **data
Retrieve data previously registered with the corresponding el_set() call.

EL_UNBUFFERED, int
Sets or clears unbuffered mode. In this mode, el_gets() will return immediately after processing a single character.

EL_PREP_TERM, int
Sets or clears terminal editing mode.

el_source()
Initialise editline by reading the contents of file. el_parse() is called for each line in file. If file is NULL, try $PWD/.editrc then $HOME/.editrc. Refer to editrc(5) for details on the format of file.

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_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 NULL 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 num 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()
Initialise the history list, and return a data structure to be used by all other history list functions.

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_tcurr, 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.

H_NEXT
Return the next element in the history.

H_CURR
Return the current element in the history.

H_SET
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 was has been called with a non-zero arguments, 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_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 num
Delete the event numbered e. This function is only provided for readline(3) 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()
Initialise 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 NULL terminated string totokenize.

SEE ALSO

sh(1), signal(3), curses(3), editrc(5)

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.

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.