```
**********************************************************
   17692 Sat Jan 18 13:36:57 2020
new/usr/src/man/man3tecla/cpl_complete_word.3tecla
12212 typos in some section 3tecla man pages
**********************************************************
```

```
'\" te
.\" Copyright (c) 2000, 2001, 2002, 2003, 2004 by Martin C. Shepherd. All Rights
.\" Permission is hereby granted, free of charge, to any person obtaining a copy
.\" "Software"), to deal in the Software without restriction, including
.\" without limitation the rights to use, copy, modify, merge, publish,
.\" distribute, and/or sell copies of the Software, and to permit persons
.\" to whom the Software is furnished to do so, provided that the above
.\" copyright notice(s) and this permission notice appear in all copies of
.\" the Software and that both the above copyright notice(s) and this
.\" permission notice appear in supporting documentation.
.\"
.\" THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
.\" OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
.\" MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT
.\" OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
.\" HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL
.\" INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING
.\" FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
.\" NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
.\" WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
.\"
.\" Except as contained in this notice, the name of a copyright holder
.\" shall not be used in advertising or otherwise to promote the sale, use
.\" or other dealings in this Software without prior written authorization
.\" of the copyright holder.
.\" Portions Copyright (c) 2007, Sun Microsystems, Inc. All Rights Reserved.
.TH CPL_COMPLETE_WORD 3TECLA "January 18, 2020"
.TH CPL_COMPLETE_WORD 3TECLA "Jun 1, 2004"
.SH NAME
cpl_complete_word, cfc_file_start, cfc_literal_escapes, cfc_set_check_fn,
cpl_add_completion, cpl_file_completions, cpl_last_error, cpl_list_completions,
cpl_recall_matches, cpl_record_error, del_CplFileConf, cpl_check_exe,
del_WordCompletion, new_CplFileConf, new_WordCompletion \- look up possible
completions for a word
.SH SYNOPSIS
.LP
.nf
cc [ \fIflag\fR\&.\|.\|. ] \fIfile\fR\&.\|.\|. \fB-ltecla\fR [ \fIlibrary\fR\&.\
#include <stdio.h>
#include <libtecla.h>

\fBWordCompletion *\fR\fBnew_WordCompletion\fR(\fBvoid\fR);
.fi

.LP
.nf
\fBWordCompletion *\fR\fBdel_WordCompletion\fR(\fBWordCompletion *\fR\fIcpl\fR);
.fi

.LP
.nf
\fBCPL_MATCH_FN\fR(\fBcpl_file_completions\fR);
.fi

.LP
.nf
\fBCplFileConf *\fR\fBnew_CplFileConf\fR(\fBvoid\fR);
.fi

.LP
.nf
```



```
\fBvoid\fR \fBcfc_file_start\fR(\fB(CplFileConf *\fR\fIcfc\fR, \fBint\fR \fIstar
.fi

.LP
.nf
\fBvoid\fR \fBcfc_literal_escapes\fR(\fBCplFileConf *\fR\fIcfc\fR, \fBint\fR \fI
.fi

.LP
.nf
\fBvoid\fR \fBcfc_set_check_fn\fR(\fBCplFileConf *\fR\fIcfc\fR, \fBCplCheckFn *\
     \fBvoid *\fR\fIchk_data\fR);
.fi

.LP
.nf
\fBCPL_CHECK_FN\fR(\fBcpl_check_exe\fR);
.fi

.LP
.nf
\fBCplFileConf *\fR\fBdel_CplFileConf\fR(\fBCplFileConf *\fR\fIcfc\fR);
.fi

.LP
.nf
\fBCplMatches *\fR\fBcpl_complete_word\fR(\fBWordCompletion *\fR\fIcpl\fR, \fBco
     \fBint\fR \fIword_end\fR, \fBvoid *\fR\fIdata\fR, \fBCplMatchFn *\fR\fImatc
.fi

.LP
.nf
\fBCplMatches *\fR\fBcpl_recall_matches\fR(\fBWordCompletion *\fR\fIcpl\fR);
.fi

.LP
.nf
\fBint\fR \fBcpl_list_completions\fR(\fBCplMatches *\fR\fIresult\fR, \fBFILE *\f
.fi

.LP
.nf
\fBint\fR \fBcpl_add_completion\fR(\fBWordCompletion *\fR\fIcpl\fR, \fBconst cha
     \fBint\fR \fIword_start\fR, \fBint\fR \fIword_end\fR, \fBconst char *\fR\fI
     \fBconst char *\fR\fItype_suffix\fR, \fBconst char *\fR\fIcont_suffix\fR);
.fi

.LP
.nf
\fBvoid\fR \fBcpl_record_error\fR(\fBWordCompletion *\fR\fIcpl\fR, \fBconst char
.fi

.LP
.nf
\fBconst char *\fR\fBcpl_last_error\fR(\fBWordCompletion *\fR\fIcpl\fR);
.fi

.SH DESCRIPTION
.sp
.LP
The \fBcpl_complete_word()\fR function is part of the \fBlibtecla\fR(3LIB)
library. It is usually called behind the scenes by \fBgl_get_line\fR(3TECLA),
but can also be called separately.
.sp
.LP
Given an input line containing an incomplete word to be completed, it calls a
```

user-provided callback function (or the provided file-completion callback function) to look up all possible completion suffixes for that word. The callback function is expected to look backward in the line, starting from the specified cursor position, to find the start of the word to be completed, then to look up all possible completions of that word and record them, one at a time, by calling \fBcpl_add_completion()\fR.
.sp
.LP
The \fBnew_WordCompletion()\fR function creates the resources used by the \fBcpl_complete_word()\fR function. In particular, it maintains the memory that is used to return the results of calling \fBcpl_complete_word()\fR.
.sp
.LP
The \fBdel_WordCompletion()\fR function deletes the resources that were returned by a previous call to \fBnew_WordCompletion()\fR. It always returns \fINULL\fR (that is, a deleted object). It takes no action if the \fIcpl\fR argument is \fINULL\fR.
.sp
.LP
The callback functions that look up possible completions should be defined with the \fBCPL_MATCH_FN()\fR macro, which is defined in <\fBlibtecla.h\fR>. Functions of this type are called by \fBcpl_complete_word()\fR, and all of the arguments of the callback are those that were passed to said function. In particular, the \fIline\fR argument contains the input line containing the word to be completed, and \fIword_end\fR is the index of the character that follows the last character of the incomplete word within this string. The callback is expected to look backwards from \fIword_end\fR for the start of the incomplete word. What constitutes the start of a word clearly depends on the application, so it makes sense for the callback to take on this responsibility. For example, the builtin filename completion function looks backwards until it encounters an unescaped space or the start of the line. Having found the start of the word, the callback should then lookup all possible completions of this word, and record each completion with separate calls to \fBcpl_add_completion()\fR. If the callback needs access to an application-specific symbol table, it can pass it and any other data that it needs using the \fIdata\fR argument. This removes any need for global variables.
.sp
.LP
The callback function should return 0 if no errors occur. On failure it should return 1 and register a terse description of the error by calling \fBcpl_record_error()\fR.
.sp
.LP
The last error message recorded by calling \fBcpl_record_error()\fR can subsequently be queried by calling \fBcpl_last_error()\fR.
.sp
.LP
The \fBcpl_add_completion()\fR function is called zero or more times by the completion callback function to record each possible completion in the specified \fBWordCompletion\fR object. These completions are subsequently returned by \fBcpl_complete_word()\fR. The \fIcpl\fR, \fIline\fR, and \fIword_end\fR arguments should be those that were passed to the callback function. The \fIword_start\fR argument should be the index within the input line string of the start of the word that is being completed. This should equal \fIword_end\fR if a zero-length string is being completed. The \fIsuffix\fR argument is the string that would have to be appended to the incomplete word to complete it. If this needs any quoting (for example, the addition of **backslashes before special characters) to be valid within the displayed input** *backslashes before special charaters) to be valid within the displayed input* line, this should be included. A copy of the suffix string is allocated internally, so there is no need to maintain your copy of the string after \fBcpl_add_completion()\fR returns.
.sp
.LP
In the array of possible completions that the \fBcpl_complete_word()\fR function returns, the suffix recorded by \fBcpl_add_completion()\fR is listed

along with the concatenation of this suffix with the word that lies between *along with the concatentation of this suffix with the word that lies between* \fIword_start\fR and \fIword_end\fR in the input line.
.sp
.LP
The \fItype_suffix\fR argument specifies an optional string to be appended to the completion if it is displayed as part of a list of completions by **\fIcpl_list_completions\fR. The intention is that this indicates to the user the** *\fIcpl_list_completions\fR. The intention is that this indicate to the user the* type of each completion. For example, the file completion function places a directory separator after completions that are directories, to indicate their **nature to the user. Similarly, if the completion were a function, you could** *nature to the user. Similary, if the completion were a function, you could* indicate this to the user by setting \fItype_suffix\fR to "()". Note that the \fItype_suffix\fR string is not copied, so if the argument is not a literal string between speech marks, be sure that the string remains valid for at least as long as the results of \fBcpl_complete_word()\fR are needed.
.sp
.LP
The \fIcont_suffix\fR argument is a continuation suffix to append to the completed word in the input line if this is the only completion. This is something that is not part of the completion itself, but that gives the user an indication about how they might continue to extend the token. For example, the file-completion callback function adds a directory separator if the completed word is a directory. If the completed word were a function name, you could similarly aid the user by arranging for an open parenthesis to be appended.
.sp
.LP
The \fBcpl_complete_word()\fR function is normally called behind the scenes by *The \fBcpl_complete_word()\fR is normally called behind the scenes by* \fBgl_get_line\fR(3TECLA), but can also be called separately if you separately allocate a \fBWordCompletion\fR object. It performs word completion, as described at the beginning of this section. Its first argument is a resource object previously returned by \fBnew_WordCompletion()\fR. The \fIline\fR argument is the input line string, containing the word to be completed. The \fIword_end\fR argument contains the index of the character in the input line, that just follows the last character of the word to be completed. When called by \fBgl_get_line()\fR, this is the character over which the user pressed TAB. The \fImatch_fn\fR argument is the function pointer of the callback function which will lookup possible completions of the word, as described above, and the \fIdata\fR argument provides a way for the application to pass arbitrary data to the callback function.
.sp
.LP
If no errors occur, the \fBcpl_complete_word()\fR function returns a pointer to a \fBCplMatches\fR container, as defined below. This container is allocated as part of the \fIcpl\fR object that was passed to \fBcpl_complete_word()\fR, and will thus change on each call which uses the same \fIcpl\fR argument.
.sp
.in +2
.nf
typedef struct {
    char *completion;   /* A matching completion */
                             /* string */
    char *suffix;   /* The part of the */
                             /* completion string which */
                             /* would have to be */
                             /* appended to complete the */
                             /* original word. */
    const char *type_suffix; /* A suffix to be added when */
                             /* listing completions, to */
                             /* indicate the type of the */
                             /* completion. */
} CplMatch;
_____***unchanged_portion_omitted_***
.fi

```
.in -2

.sp
.LP
If an error occurs during completion, \fBcpl_complete_word()\fR returns
\fINULL\fR. A description of the error can be acquired by calling the
\fBcpl_last_error()\fR function.
.sp
.LP
The \fBcpl_last_error()\fR function returns a terse description of the error
which occurred on the last call to \fBcpl_complete_word()\fR or
which occurred on the last call to \fBcpl_com plete_word()\fR or
\fBcpl_add_completion()\fR.
.sp
.LP
As a convenience, the return value of the last call to
\fBcpl_complete_word()\fR can be recalled at a later time by calling
\fBcpl_recall_matches()\fR. If \fBcpl_complete_word()\fR returned \fINULL\fR,
so will \fBcpl_recall_matches()\fR.
.sp
.LP
When the \fBcpl_complete_word()\fR function returns multiple possible
completions, the \fBcpl_list_completions()\fR function can be called upon to
list them, suitably arranged across the available width of the terminal. It
arranges for the displayed columns of completions to all have the same width,
set by the longest completion. It also appends the \fItype_suffix\fR strings
that were recorded with each completion, thus indicating their types to the
user.
.SS "Builtin Filename completion Callback"
By default the \fBgl_get_line()\fR function passes the
.sp
.LP
By default the \fBgl_get_line()\fR function, passes the
\fBCPL_MATCH_FN\fR(\fBcps_file_completions\fR) completion callback function to
\fBcpl_complete_word()\fR. This function can also be used separately, either by
sending it to \fBcpl_complete_word()\fR, or by calling it directly from your
own completion callback function.
.sp
.in +2
.nf
#define CPL_MATCH_FN(fn) int (fn)(WordCompletion *cpl, \e
                                  void *data, const char *line, \e
                                  int word_end)

typedef CPL_MATCH_FN(CplMatchFn);

CPL_MATCH_FN(cpl_file_completions);
.fi
.in -2

.sp
.LP
Certain aspects of the behavior of this callback can be changed via its
\fIdata\fR argument. If you are happy with its default behavior you can pass
\fINULL\fR in this argument. Otherwise it should be a pointer to a
\fBCplFileConf\fR object, previously allocated by calling
\fBnew_CplFileConf()\fR.
.sp
.LP
\fBCplFileConf\fR objects encapsulate the configuration parameters of
\fBcpl_file_completions()\fR. These parameters, which start out with default
values, can be changed by calling the accessor functions described below.
.sp
.LP
By default, the \fBcpl_file_completions()\fR callback function searches
backwards for the start of the filename being completed, looking for the first
```



```
unescaped space or the start of the input line. If you wish to specify a
different location, call \fBcfc_file_start()\fR with the index at which the
filename starts in the input line. Passing \fIstart_index\fR=-1 reenables the
default behavior.
.sp
.LP
By default, when \fBcpl_file_completions()\fR looks at a filename in the input
line, each lone backslash in the input line is interpreted as being a special
character which removes any special significance of the character which follows
it, such as a space which should be taken as part of the filename rather than
delimiting the start of the filename. These backslashes are thus ignored while
looking for completions, and subsequently added before spaces, tabs and literal
backslashes in the list of completions. To have unescaped backslashes treated
back slashes in the list of completions. To have unescaped back slashes treated
as normal characters, call \fBcfc_literal_escapes()\fR with a non-zero value in
its \fIliteral\fR argument.
.sp
.LP
By default, \fBcpl_file_completions()\fR reports all files whose names start
with the prefix that is being completed. If you only want a selected subset of
these files to be reported in the list of completions, you can arrange this by
providing a callback function which takes the full pathname of a file, and
returns 0 if the file should be ignored, or 1 if the file should be included in
the list of completions. To register such a function for use by
\fBcpl_file_completions()\fR, call \fBcfc_set_check_fn()\fR, and pass it a
pointer to the function, together with a pointer to any data that you would
like passed to this callback whenever it is called. Your callback can make its
decisions based on any property of the file, such as the filename itself,
whether the file is readable, writable or executable, or even based on what the
file contains.
.sp
.in +2
.nf
#define CPL_CHECK_FN(fn) int (fn)(void *data, \e
                                  const char *pathname)

typedef CPL_CHECK_FN(CplCheckFn);

void cfc_set_check_fn(CplFileConf *cfc, CplCheckFn *chk_fn, \e
                                        void *chk_data);
.fi
.in -2

.sp
.LP
The \fBcpl_check_exe()\fR function is a provided callback of the above type,
for use with \fBcpl_file_completions()\fR. It returns non-zero if the filename
that it is given represents a normal file that the user has permission to
execute. You could use this to have \fBcpl_file_completions()\fR only list
that it is given represents a normal file that the user has execute permission
to. You could use this to have \fBcpl_file_completions()\fR only list
completions of executable files.
.sp
.LP
When you have finished with a \fBCplFileConf\fR variable, you can pass it to
the \fBdel_CplFileConf()\fR destructor function to reclaim its memory.
.SS "Thread Safety"
.sp
.LP
It is safe to use the facilities of this module in multiple threads, provided
that each thread uses a separately allocated \fBWordCompletion\fR object. In
other words, if two threads want to do word completion, they should each call
\fBnew_WordCompletion()\fR to allocate their own completion objects.
.SH ATTRIBUTES
.sp
.LP
```

```
388 See \fBattributes\fR(5) for descriptions of the following attributes:
389 .sp

391 .sp
392 .TS
393 box;
394 c | c
395 l | l .
396 ATTRIBUTE TYPE  ATTRIBUTE VALUE
397 _
398 Interface Stability     Evolving
399 _
400 MT-Level        MT-Safe
401 .TE

403 .SH SEE ALSO
413 .sp
414 .LP
404 \fBef_expand_file\fR(3TECLA), \fBgl_get_line\fR(3TECLA), \fBlibtecla\fR(3LIB),
405 \fBpca_lookup_file\fR(3TECLA), \fBattributes\fR(5)
```

```
**********************************************************
   85606 Sat Jan 18 13:36:57 2020
new/usr/src/man/man3tecla/gl_get_line.3tecla
12212 typos in some section 3tecla man pages
**********************************************************
```

```
'\" te
.\" Copyright (c) 2000, 2001, 2002, 2003, 2004 by Martin C. Shepherd.
.\" All Rights Reserved.
.\" Permission is hereby granted, free of charge, to any person obtaining a copy
.\" "Software"), to deal in the Software without restriction, including
.\" without limitation the rights to use, copy, modify, merge, publish,
.\" distribute, and/or sell copies of the Software, and to permit persons
.\" to whom the Software is furnished to do so, provided that the above
.\" copyright notice(s) and this permission notice appear in all copies of
.\" the Software and that both the above copyright notice(s) and this
.\" permission notice appear in supporting documentation.
.\"
.\" THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
.\" OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
.\" MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT
.\" OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
.\" HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL
.\" INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING
.\" FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
.\" NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
.\" WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
.\"
.\" Except as contained in this notice, the name of a copyright holder
.\" shall not be used in advertising or otherwise to promote the sale, use
.\" or other dealings in this Software without prior written authorization
.\" of the copyright holder.
.\" Portions Copyright (c) 2007, Sun Microsystems, Inc. All Rights Reserved.
.TH GL_GET_LINE 3TECLA "January 18, 2020"
.TH GL_GET_LINE 3TECLA "April 9, 2016"
.SH NAME
gl_get_line, new_GetLine, del_GetLine, gl_customize_completion,
gl_change_terminal, gl_configure_getline, gl_load_history, gl_save_history,
gl_group_history, gl_show_history, gl_watch_fd, gl_inactivity_timeout,
gl_terminal_size, gl_set_term_size, gl_resize_history, gl_limit_history,
gl_clear_history, gl_toggle_history, gl_lookup_history, gl_state_of_history,
gl_range_of_history, gl_size_of_history, gl_echo_mode, gl_replace_prompt,
gl_prompt_style, gl_ignore_signal, gl_trap_signal, gl_last_signal,
gl_completion_action, gl_register_action, gl_display_text, gl_return_status,
gl_error_message, gl_catch_blocked, gl_list_signals, gl_bind_keyseq,
gl_erase_terminal, gl_automatic_history, gl_append_history, gl_query_char,
gl_read_char \- allow the user to compose an input line
.SH SYNOPSIS
.LP
.nf
cc [ \fIflag\fR\&.\|.\|. ] \fIfile\fR\&.\|.\|. \fB-ltecla\fR [ \fIlibrary\fR\&.\
#include <stdio.h>
#include <libtecla.h>

\fBGetLine *\fR\fBnew_GetLine\fR(\fBsize_t\fR \fIlinelen\fR, \fBsize_t\fR \fIhis
.fi

.LP
.nf
\fBGetLine *\fR\fBdel_GetLine\fR(\fBGetLine *\fR\fIgl\fR);
.fi

.LP
.nf
\fBchar *\fR\fBgl_get_line\fR(\fBGetLine *\fR\fIgl\fR, \fBconst char *\fR\fIprom
     \fBconst char *\fR\fIstart_line\fR, \fBint\fR \fIstart_pos\fR);
.fi
```



```
.LP
.nf
\fBint\fR \fBgl_query_char\fR(\fBGetLine *\fR\fIgl\fR, \fBconst char *\fR\fIprom
.fi

.LP
.nf
\fBint\fR \fBgl_read_char\fR(\fBGetLine *\fR\fIgl\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_customize_completion\fR(\fBGetLine *\fR\fIgl\fR, \fBvoid *\fR\fI
     \fBCplMatchFn *\fR\fImatch_fn\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_change_terminal\fR(\fBGetLine *\fR\fIgl\fR, \fBFILE *\fR\fIinput
     \fBFILE *\fR\fIoutput_fp\fR, \fBconst char *\fR\fIterm\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_configure_getline\fR(\fBGetLine *\fR\fIgl\fR, \fBconst char *\fR
     \fBconst char *\fR\fIapp_file\fR,\ \fBconst char *\fR\fIuser_file\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_bind_keyseq\fR(\fBGetLine *\fR\fIgl\fR, \fBGlKeyOrigin\fR \fIori
     \fBconst char *\fR\fIkeyseq\fR, \fBconst char *\fR\fIaction\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_save_history\fR(\fBGetLine *\fR\fIgl\fR, \fBconst char *\fR\fIfi
     \fBconst char *\fR\fIcomment\fR, \fBint\fR \fImax_lines\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_load_history\fR(\fBGetLine *\fR\fIgl\fR, \fBconst char *\fR\fIfi
     \fBconst char *\fR\fIcomment\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_watch_fd\fR(\fBGetLine *\fR\fIgl\fR, \fBint\fR \fIfd\fR, \fBGlFd
     \fBGlFdEventFn *\fR\fIcallback\fR, \fBvoid *\fR\fIdata\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_inactivity_timeout\fR(\fBGetLine *\fR\fIgl\fR, \fBGlTimeoutFn *\
     \fBvoid *\fR\fIdata\fR, \fBunsigned long\fR \fIsec\fR, \fBunsigned long\fR
.fi

.LP
.nf
\fBint\fR \fBgl_group_history\fR(\fBGetLine *\fR\fIgl\fR, \fBunsigned\fR \fIstre
.fi

.LP
.nf
```

```
\fBint\fR \fBgl_show_history\fR(\fBGetLine *\fR\fIgl\fR, \fBFILE *\fR\fIfp\fR, \
     \fBint\fR \fIall_groups\fR, \fBint\fR \fImax_lines\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_resize_history\fR(\fBGetLine *\fR\fIgl\fR, \fBsize_t\fR \fIbufsi
.fi

.LP
.nf
\fBvoid\fR \fBgl_limit_history\fR(\fBGetLine *\fR\fIgl\fR, \fBint\fR \fImax_line
.fi

.LP
.nf
\fBvoid\fR \fBgl_clear_history\fR(\fBGetLine *\fR\fIgl\fR, \fBint\fR \fIall_grou
.fi

.LP
.nf
\fBvoid\fR \fBgl_toggle_history\fR(\fBGetLine *\fR\fIgl\fR, \fBint\fR \fIenable\
.fi

.LP
.nf
\fBGlTerminalSize\fR \fBgl_terminal_size\fR(\fBGetLine *\fR\fIgl\fR, \fBint\fR \
     \fBint\fR \fIdef_nline\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_set_term_size\fR(\fBGetLine *\fR\fIgl\fR, \fBint\fR \fIncolumn\f
.fi

.LP
.nf
\fBint\fR \fBgl_lookup_history\fR(\fBGetLine *\fR\fIgl\fR, \fBunsigned long\fR \
     \fBGlHistoryLine *\fR\fIhline\fR);
.fi

.LP
.nf
\fBvoid\fR \fBgl_state_of_history\fR(\fBGetLine *\fR\fIgl\fR, \fBGlHistoryState
.fi

.LP
.nf
\fBvoid\fR \fBgl_range_of_history\fR(\fBGetLine *\fR\fIgl\fR, \fBGlHistoryRange
.fi

.LP
.nf
\fBvoid\fR \fBgl_size_of_history\fR(\fBGetLine *\fR\fIgl\fR, \fBGlHistorySize *\
.fi

.LP
.nf
\fBvoid\fR \fBgl_echo_mode\fR(\fBGetLine *\fR\fIgl\fR, \fBint\fR \fIenable\fR);
.fi

.LP
.nf
\fBvoid\fR \fBgl_replace_prompt\fR(\fBGetLine *\fR\fIgl\fR, \fBconst char *\fR\f
.fi
```



```
.LP
.nf
\fBvoid\fR \fBgl_prompt_style\fR(\fBGetLine *\fR\fIgl\fR, \fBGlPromptStyle\fR \f
.fi

.LP
.nf
\fBint\fR \fBgl_ignore_signal\fR(\fBGetLine *\fR\fIgl\fR, \fBint\fR \fIsigno\fR)
.fi

.LP
.nf
\fBint\fR \fBgl_trap_signal\fR(\fBGetLine *\fR\fIgl\fR, \fBint\fR \fIsigno\fR, \
     \fBGlAfterSignal\fR \fIafter\fR, \fBint\fR \fIerrno_value\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_last_signal\fR(\fBGetLine *\fR\fIgl\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_completion_action\fR(\fBGetLine *\fR\fIgl\fR, \fBvoid *\fR\fIdat
     \fBCplMatchFn *\fR\fImatch_fn\fR, \fBint\fR \fIlist_only\fR, \fBconst char
     \fBconst char *\fR\fIkeyseq\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_register_action\fR(\fBGetLine *\fR\fIgl\fR, \fBvoid *\fR\fIdata\
     \fBconst char *\fR\fIname\fR, \fBconst char *\fR\fIkeyseq\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_display_text\fR(\fBGetLine *\fR\fIgl\fR, \fBint\fR \fIindentatio
     \fBconst char *\fR\fIprefix\fR, \fBconst char *\fR\fIsuffix\fR, \fBint\fR \
     \fBint\fR \fIdef_width\fR, \fBint\fR \fIstart\fR, \fBconst char *\fR\fIstri
.fi

.LP
.nf
\fBGlReturnStatus\fR \fBgl_return_status\fR(\fBGetLine *\fR\fIgl\fR);
.fi

.LP
.nf
\fBconst char *\fR\fBgl_error_message\fR(\fBGetLine *\fR\fIgl\fR, \fBchar *\fR\f
.fi

.LP
.nf
\fBvoid\fR \fBgl_catch_blocked\fR(\fBGetLine *\fR\fIgl\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_list_signals\fR(\fBGetLine *\fR\fIgl\fR, \fBsigset_t *\fR\fIset\
.fi

.LP
.nf
\fBint\fR \fBgl_append_history\fR(\fBGetLine *\fR\fIgl\fR, \fBconst char *\fR\fI
.fi
```

```
.LP
.nf
\fBint\fR \fBgl_automatic_history\fR(\fBGetLine *\fR\fIgl\fR, \fBint\fR \fIenabl
.fi

.LP
.nf
\fBint\fR \fBgl_erase_terminal\fR(\fBGetLine *\fR\fIgl\fR);
.fi

.SH DESCRIPTION
.LP
The \fBgl_get_line()\fR function is part of the \fBlibtecla\fR(3LIB) library.
If the user is typing at a terminal, each call prompts them for a line of
input, then provides interactive editing facilities, similar to those of the
UNIX \fBtcsh\fR shell. In addition to simple command-line editing, it supports
recall of previously entered command lines, TAB completion of file names, and
in-line wild-card expansion of filenames. Documentation of both the user-level
command-line editing features and all user configuration options can be found
on the \fBtecla\fR(5) manual page.
.SS "An Example"
.LP
The following shows a complete example of how to use the \fBgl_get_line()\fR
function to get input from the user:
.sp
.in +2
.nf
#include <stdio.h>
#include <locale.h>
#include <libtecla.h>

int main(int argc, char *argv[])
{
  char *line;    /* The line that the user typed */
  GetLine *gl;   /* The gl_get_line() resource object */

  setlocale(LC_CTYPE, ""); /* Adopt the user's choice */
                           /* of character set. */

  gl = new_GetLine(1024, 2048);
  if(!gl)
    return 1;
  while((line=gl_get_line(gl, "$ ", NULL, -1)) != NULL &&
        strcmp(line, "exit\en") != 0)
    printf("You typed: %s\en", line);

  gl = del_GetLine(gl);
  return 0;
}
.fi
.in -2

.sp
.LP
In the example, first the resources needed by the \fBgl_get_line()\fR function
are created by calling \fBnew_GetLine()\fR. This allocates the memory used in
subsequent calls to the \fBgl_get_line()\fR function, including the history
buffer for recording previously entered lines. Then one or more lines are read
from the user, until either an error occurs, or the user types exit. Then
finally the resources that were allocated by \fBnew_GetLine()\fR, are returned
to the system by calling \fBdel_GetLine()\fR. Note the use of the \fINULL\fR
return value of \fBdel_GetLine()\fR to make \fIgl\fR \fINULL\fR. This is a
safety precaution. If the program subsequently attempts to pass \fIgl\fR to
\fBgl_get_line()\fR, said function will complain, and return an error, instead
of attempting to use the deleted resource object.
.SS "The Functions Used In The Example"
```



```
.LP
The \fBnew_GetLine()\fR function creates the resources used by the
\fBgl_get_line()\fR function and returns an opaque pointer to the object that
contains them. The maximum length of an input line is specified by the
\fIlinelen\fR argument, and the number of bytes to allocate for storing history
lines is set by the \fIhistlen\fR argument. History lines are stored
back-to-back in a single buffer of this size. Note that this means that the
number of history lines that can be stored at any given time, depends on the
lengths of the individual lines. If you want to place an upper limit on the
number of lines that can be stored, see the description of the
\fBgl_limit_history()\fR function. If you do not want history at all, specify
\fIhistlen\fR as zero, and no history buffer will be allocated.
.sp
.LP
On error, a message is printed to \fBstderr\fR and \fINULL\fR is returned.
.sp
.LP
The \fBdel_GetLine()\fR function deletes the resources that were returned by a
previous call to \fBnew_GetLine()\fR. It always returns \fINULL\fR (for
example, a deleted object). It does nothing if the \fIgl\fR argument is
\fINULL\fR.
.sp
.LP
The \fBgl_get_line()\fR function can be called any number of times to read
input from the user. The gl argument must have been previously returned by a
call to \fBnew_GetLine()\fR. The \fIprompt\fR argument should be a normal
null-terminated string, specifying the prompt to present the user with. By
default prompts are displayed literally, but if enabled with the
\fBgl_prompt_style()\fR function, prompts can contain directives to do
underlining, switch to and from bold fonts, or turn highlighting on and off.
.sp
.LP
If you want to specify the initial contents of the line for the user to edit,
pass the desired string with the \fIstart_line\fR argument. You can then
specify which character of this line the cursor is initially positioned over by
using the \fIstart_pos\fR argument. This should be -1 if you want the cursor to
follow the last character of the start line. If you do not want to preload the
line in this manner, send \fIstart_line\fR as \fINULL\fR, and set
\fIstart_pos\fR to -1.
.sp
.LP
The \fBgl_get_line()\fR function returns a pointer to the line entered by the
user, or \fINULL\fR on error or at the end of the input. The returned pointer
is part of the specified \fIgl\fR resource object, and thus should not be freed
by the caller, or assumed to be unchanging from one call to the next. When
reading from a user at a terminal, there will always be a newline character at
the end of the returned line. When standard input is being taken from a pipe or
a file, there will similarly be a newline unless the input line was too long to
store in the internal buffer. In the latter case you should call
\fBgl_get_line()\fR again to read the rest of the line. Note that this behavior
makes \fBgl_get_line()\fR similar to \fBfgets\fR(3C). When \fBstdin\fR is not
connected to a terminal, \fBgl_get_line()\fR simply calls \fBfgets()\fR.
.SS "The Return Status Of \fBgl_get_line()\fR"
.LP
The \fBgl_get_line()\fR function has two possible return values: a pointer to
the completed input line, or \fINULL\fR. Additional information about what
caused \fBgl_get_line()\fR to return is available both by inspecting
\fBerrno\fR and by calling the \fBgl_return_status()\fR function.
.sp
.LP
The following are the possible enumerated values returned by
\fBgl_return_status()\fR:
.sp
.ne 2
.na
\fB\fBGLR_NEWLINE\fR\fR
```

```
.ad
.RS 15n
The last call to \fBgl_get_line()\fR successfully returned a completed input
line.
.RE

.sp
.ne 2
.na
\fB\fBGLR_BLOCKED\fR\fR
.ad
.RS 15n
The \fBgl_get_line()\fR function was in non-blocking server mode, and returned
early to avoid blocking the process while waiting for terminal I/O. The
\fBgl_pending_io()\fR function can be used to see what type of I/O
\fBgl_get_line()\fR was waiting for. See the \fBgl_io_mode\fR(3TECLA).
.RE

.sp
.ne 2
.na
\fB\fBGLR_SIGNAL\fR\fR
.ad
.RS 15n
A signal was caught by \fBgl_get_line()\fR that had an after-signal disposition
of \fBGLS_ABORT\fR. See \fBgl_trap_signal()\fR.
.RE

.sp
.ne 2
.na
\fB\fBGLR_TIMEOUT\fR\fR
.ad
.RS 15n
The inactivity timer expired while \fBgl_get_line()\fR was waiting for input,
and the timeout callback function returned \fBGLTO_ABORT\fR. See
\fBgl_inactivity_timeout()\fR for information about timeouts.
.RE

.sp
.ne 2
.na
\fB\fBGLR_FDABORT\fR\fR
.ad
.RS 15n
An application I/O callback returned \fBGLFD_ABORT\fR. See
An application I/O callback returned \fBGLFD_ABORT\fR. Ssee
\fBgl_watch_fd()\fR.
.RE

.sp
.ne 2
.na
\fB\fBGLR_EOF\fR\fR
.ad
.RS 15n
End of file reached. This can happen when input is coming from a file or a
pipe, instead of the terminal. It also occurs if the user invokes the
list-or-eof or del-char-or-list-or-eof actions at the start of a new line.
.RE

.sp
.ne 2
.na
\fB\fBGLR_ERROR\fR\fR
.ad
```



```
.RS 15n
An unexpected error caused \fBgl_get_line()\fR to abort (consult \fBerrno\fR
and/or \fBgl_error_message()\fR for details.
.RE

.sp
.LP
When \fBgl_return_status()\fR returns \fBGLR_ERROR\fR and the value of
\fBerrno\fR is not sufficient to explain what happened, you can use the
\fBgl_error_message()\fR function to request a description of the last error
that occurred.
.sp
.LP
The return value of \fBgl_error_message()\fR is a pointer to the message that
occurred. If the \fIbuff\fR argument is \fINULL\fR, this will be a pointer to a
buffer within \fIgl\fR whose value will probably change on the next call to any
function associated with \fBgl_get_line()\fR. Otherwise, if a non-null
\fIbuff\fR argument is provided, the error message, including a '\e0'
terminator, will be written within the first \fIn\fR elements of this buffer,
and the return value will be a pointer to the first element of this buffer. If
the message will not fit in the provided buffer, it will be truncated to fit.
.SS "Optional Prompt Formatting"
.LP
Whereas by default the prompt string that you specify is displayed literally
without any special interpretation of the characters within it, the
\fBgl_prompt_style()\fR function can be used to enable optional formatting
directives within the prompt.
.sp
.LP
The \fIstyle\fR argument, which specifies the formatting style, can take any of
the following values:
.sp
.ne 2
.na
\fB\fBGL_FORMAT_PROMPT\fR\fR
.ad
.RS 21n
In this style, the formatting directives described below, when included in
prompt strings, are interpreted as follows:
.sp
.ne 2
.na
\fB\fB%B\fR\fR
.ad
.RS 6n
Display subsequent characters with a bold font.
.RE

.sp
.ne 2
.na
\fB\fB%b\fR\fR
.ad
.RS 6n
Stop displaying characters with the bold font.
.RE

.sp
.ne 2
.na
\fB\fB%F\fR\fR
.ad
.RS 6n
Make subsequent characters flash.
.RE
```

```
.sp
.ne 2
.na
\fB\fB%f\fR\fR
.ad
.RS 6n
Turn off flashing characters.
.RE

.sp
.ne 2
.na
\fB\fB%U\fR\fR
.ad
.RS 6n
Underline subsequent characters.
.RE

.sp
.ne 2
.na
\fB\fB%u\fR\fR
.ad
.RS 6n
Stop underlining characters.
.RE

.sp
.ne 2
.na
\fB\fB%P\fR\fR
.ad
.RS 6n
Switch to a pale (half brightness) font.
.RE

.sp
.ne 2
.na
\fB\fB%p\fR\fR
.ad
.RS 6n
Stop using the pale font.
.RE

.sp
.ne 2
.na
\fB\fB%S\fR\fR
.ad
.RS 6n
Highlight subsequent characters (also known as standout mode).
.RE

.sp
.ne 2
.na
\fB\fB%s\fR\fR
.ad
.RS 6n
Stop highlighting characters.
.RE

.sp
.ne 2
.na
```



```
\fB\fB%V\fR\fR
.ad
.RS 6n
Turn on reverse video.
.RE

.sp
.ne 2
.na
\fB\fB%v\fR\fR
.ad
.RS 6n
Turn off reverse video.
.RE

.sp
.ne 2
.na
\fB\fB%%\fR\fR
.ad
.RS 6n
Display a single % character.
.RE

For example, in this mode, a prompt string like "%UOK%u$" would display the
prompt "OK$", but with the OK part underlined.
.sp
Note that although a pair of characters that starts with a % character, but
does not match any of the above directives is displayed literally, if a new
directive is subsequently introduced which does match, the displayed prompt
will change, so it is better to always use %% to display a literal %.
.sp
Also note that not all terminals support all of these text attributes, and that
some substitute a different attribute for missing ones.
.RE

.sp
.ne 2
.na
\fB\fBGL_LITERAL_PROMPT\fR\fR
.ad
.RS 21n
In this style, the prompt string is printed literally. This is the default
style.
.RE

.SS "Alternate Configuration Sources"
.LP
By default users have the option of configuring the behavior of
\fBgl_get_line()\fR with a configuration file called \fB\&.teclarc\fR in their
home directories. The fact that all applications share this same configuration
file is both an advantage and a disadvantage. In most cases it is an advantage,
since it encourages uniformity, and frees the user from having to configure
each application separately. In some applications, however, this single means
of configuration is a problem. This is particularly true of embedded software,
where there's no filesystem to read a configuration file from, and also in
applications where a radically different choice of keybindings is needed to
emulate a legacy keyboard interface. To cater for such cases, the
\fBgl_configure_getline()\fR function allows the application to control where
configuration information is read from.
.sp
.LP
The \fBgl_configure_getline()\fR function allows the configuration commands
that would normally be read from a user's \fB~/.teclarc\fR file, to be read
from any or none of, a string, an application specific configuration file,
and/or a user-specific configuration file. If this function is called before
```

the first call to \fBgl_get_line()\fR, the default behavior of reading
\fB~/.teclarc\fR on the first call to \fBgl_get_line()\fR is disabled, so all
configurations must be achieved using the configuration sources specified with
this function.
.sp
.LP
If \fIapp_string\fR != \fINULL\fR, then it is interpreted as a string
containing one or more configuration commands, separated from each other in the
string by embedded newline characters. If \fIapp_file\fR != \fINULL\fR then it
is interpreted as the full pathname of an application-specific configuration
file. If user_file != \fINULL\fR then it is interpreted as the full path name
of a user-specific configuration file, such as \fB~/.teclarc\fR. For example,
in the call
.sp
.in +2
.nf
gl_configure_getline(gl, "edit-mode vi \en nobeep",
                     "/usr/share/myapp/teclarc", "~/.teclarc");
.fi
.in -2

.sp
.LP
The \fIapp_string\fR argument causes the calling application to start in
\fBvi\fR(1) edit-mode, instead of the default \fBemacs\fR mode, and turns off
the use of the terminal bell by the library. It then attempts to read
system-wide configuration commands from an optional file called
\fB/usr/share/myapp/teclarc\fR, then finally reads user-specific configuration
commands from an optional \fB\&.teclarc\fR file in the user's home directory.
Note that the arguments are listed in ascending order of priority, with the
contents of \fIapp_string\fR being potentially overridden by commands in
contents of \fIapp_string\fR being potentially over riden by commands in
\fIapp_file\fR, and commands in \fIapp_file\fR potentially being overridden by
commands in \fIuser_file\fR.
.sp
.LP
You can call this function as many times as needed, the results being
cumulative, but note that copies of any file names specified with the
\fIapp_file\fR and \fIuser_file\fR arguments are recorded internally for
subsequent use by the read-init-files key-binding function, so if you plan to
call this function multiple times, be sure that the last call specifies the
filenames that you want re-read when the user requests that the configuration
files be re-read.
.sp
.LP
Individual key sequences can also be bound and unbound using the
\fBgl_bind_keyseq()\fR function. The \fIorigin\fR argument specifies the
priority of the binding, according to whom it is being established for, and
must be one of the following two values.
.sp
.ne 2
.na
\fB\fBGL_USER_KEY\fR\fR
.ad
.RS 15n
The user requested this key-binding.
.RE

.sp
.ne 2
.na
\fB\fBGL_APP_KEY\fR\fR
.ad
.RS 15n
This is a default binding set by the application.
.RE

.sp
.LP
When both user and application bindings for a given key sequence have been
specified, the user binding takes precedence. The application's binding is
subsequently reinstated if the user's binding is later unbound with either
another call to this function, or a call to \fBgl_configure_getline()\fR.
.sp
.LP
The \fIkeyseq\fR argument specifies the key sequence to be bound or unbound,
and is expressed in the same way as in a \fB~/.teclarc\fR configuration file.
The \fIaction\fR argument must either be a string containing the name of the
action to bind the key sequence to, or it must be \fINULL\fR or \fB""\fR to
unbind the key sequence.
.SS "Customized Word Completion"
.LP
If in your application you would like to have TAB completion complete other
things in addition to or instead of filenames, you can arrange this by
registering an alternate completion callback function with a call to the
\fBgl_customize_completion()\fR function.
.sp
.LP
The \fIdata\fR argument provides a way for your application to pass arbitrary,
application-specific information to the callback function. This is passed to
the callback every time that it is called. It might for example point to the
symbol table from which possible completions are to be sought. The
\fImatch_fn\fR argument specifies the callback function to be called. The
\fICplMatchFn\fR function type is defined in <\fBlibtecla.h\fR>, as is a
\fBCPL_MATCH_FN()\fR macro that you can use to declare and prototype callback
functions. The declaration and responsibilities of callback functions are
described in depth on the \fBcpl_complete_word\fR(3TECLA) manual page.
.sp
.LP
The callback function is responsible for looking backwards in the input line
from the point at which the user pressed TAB, to find the start of the word
being completed. It then must lookup possible completions of this word, and
record them one by one in the \fBWordCompletion\fR object that is passed to it
as an argument, by calling the \fBcpl_add_completion()\fR function. If the
callback function wants to provide filename completion in addition to its own
specific completions, it has the option of itself calling the builtin filename
completion callback. This is also documented in the
completion callback. This also is documented on the
\fBcpl_complete_word\fR(3TECLA) manual page.
.sp
.LP
If you would like \fBgl_get_line()\fR to return the current input line when a
successful completion has been made, you can arrange this when you call
successful completion is been made, you can arrange this when you call
\fBcpl_add_completion()\fR by making the last character of the continuation
suffix a newline character. The input line will be updated to display the
completion, together with any continuation suffix up to the newline character,
completion, together with any contiuation suffix up to the newline character,
and \fBgl_get_line()\fR will return this input line.
.sp
.LP
If your callback function needs to write something to the terminal, it must
call \fBgl_normal_io()\fR before doing so. This will start a new line after the
input line that is currently being edited, reinstate normal terminal I/O, and
notify \fBgl_get_line()\fR that the input line will need to be redrawn when the
callback returns.
.SS "Adding Completion Actions"
.LP
In the previous section the ability to customize the behavior of the only
default completion action, complete-word, was described. In this section the
ability to install additional action functions, so that different types of word
completion can be bound to different key sequences, is described. This is

achieved by using the \fBgl_completion_action()\fR function.
.sp
.LP
The \fIdata\fR and \fImatch_fn\fR arguments are as described on the
\fBcpl_complete_word\fR(3TECLA) manual page, and specify the callback function
that should be invoked to identify possible completions. The \fIlist_only\fR
argument determines whether the action that is being defined should attempt to
complete the word as far as possible in the input line before displaying any
possible ambiguous completions, or whether it should simply display the list of
possible completions without touching the input line. The former option is
selected by specifying a value of 0, and the latter by specifying a value of 1.
The \fIname\fR argument specifies the name by which configuration files and
future invocations of this function should refer to the action. This must
either be the name of an existing completion action to be changed, or be a new
unused name for a new action. Finally, the \fIkeyseq\fR argument specifies the
default key sequence to bind the action to. If this is \fINULL\fR, no new key
sequence will be bound to the action.
.sp
.LP
Beware that in order for the user to be able to change the key sequence that is
bound to actions that are installed in this manner, you should call
bound to actions that are installed in this manner, you shouldcall
\fBgl_completion_action()\fR to install a given action for the first time
between calling \fBnew_GetLine()\fR and the first call to \fBgl_get_line()\fR.
Otherwise, when the user's configuration file is read on the first call to
\fBgl_get_line()\fR, the name of the your additional action will not be known,
and any reference to it in the configuration file will generate an error.
.sp
.LP
As discussed for \fBgl_customize_completion()\fR, if your callback function
needs to write anything to the terminal, it must call \fBgl_normal_io()\fR
before doing so.
.SS "Defining Custom Actions"
.LP
Although the built-in key-binding actions are sufficient for the needs of most
applications, occasionally a specialized application may need to define one or
more custom actions, bound to application-specific key sequences. For example,
a sales application would benefit from having a key sequence that displayed the
part name that corresponded to a part number preceding the cursor. Such a
feature is clearly beyond the scope of the built-in action functions. So for
such special cases, the \fBgl_register_action()\fR function is provided.
.sp
.LP
The \fBgl_register_action()\fR function lets the application register an
external function, \fIfn\fR, that will thereafter be called whenever either the
specified key sequence, \fIkeyseq\fR, is entered by the user, or the user
enters any other key sequence that the user subsequently binds to the specified
action name, \fIname\fR, in their configuration file. The \fIdata\fR argument
can be a pointer to anything that the application wants to have passed to the
action function, \fIfn\fR, whenever that function is invoked.
.sp
.LP
The action function, \fIfn\fR, should be declared using the
\fBGL_ACTION_FN()\fR macro, which is defined in <\fBlibtecla.h\fR>.
.sp
.in +2
.nf
#define GL_ACTION_FN(fn) GlAfterAction (fn)(GetLine *gl, \e
                         void *data, int count, size_t curpos, \e
                         const char *line)
.fi
.in -2

.sp
.LP
The \fIgl\fR and \fIdata\fR arguments are those that were previously passed to

\fBgl_register_action()\fR when the action function was registered. The
\fIcount\fR argument is a numeric argument which the user has the option of
entering using the digit-argument action, before invoking the action. If the
user does not enter a number, then the \fIcount\fR argument is set to 1.
Nominally this argument is interpreted as a repeat count, meaning that the
action should be repeated that many times. In practice however, for some
actions a repeat count makes little sense. In such cases, actions can either
simply ignore the \fIcount\fR argument, or use its value for a different
purpose.
.sp
.LP
A copy of the current input line is passed in the read-only \fIline\fR
argument. The current cursor position within this string is given by the index
contained in the \fIcurpos\fR argument. Note that direct manipulation of the
input line and the cursor position is not permitted because the rules dictated
by various modes (such as \fBvi\fR mode versus \fBemacs\fR mode, no-echo mode,
and insert mode versus overstrike mode) make it too complex for an application
writer to write a conforming editing action, as well as constrain future
changes to the internals of \fBgl_get_line()\fR. A potential solution to this
dilemma would be to allow the action function to edit the line using the
existing editing actions. This is currently under consideration.
.sp
.LP
If the action function wishes to write text to the terminal without this
getting mixed up with the displayed text of the input line, or read from the
terminal without having to handle raw terminal I/O, then before doing either of
these operations, it must temporarily suspend line editing by calling the
\fBgl_normal_io()\fR function. This function flushes any pending output to the
terminal, moves the cursor to the start of the line that follows the last
terminal line of the input line, then restores the terminal to a state that is
suitable for use with the C \fBstdio\fR facilities. The latter includes such
things as restoring the normal mapping of \en to \er\en, and, when in server
mode, restoring the normal blocking form of terminal I/O. Having called this
function, the action function can read from and write to the terminal without
the fear of creating a mess. It is not necessary for the action function to
restore the original editing environment before it returns. This is done
automatically by \fBgl_get_line()\fR after the action function returns. The
following is a simple example of an action function which writes the sentence
"Hello world" on a new terminal line after the line being edited. When this
function returns, the input line is redrawn on the line that follows the "Hello
world" line, and line editing resumes.
.sp
.in +2
.nf
static GL_ACTION_FN(say_hello_fn)
{
    if(gl_normal_io(gl))   /* Temporarily suspend editing */
        return GLA_ABORT;
    printf("Hello world\en");
    return GLA_CONTINUE;
}
.fi
.in -2

.sp
.LP
Action functions must return one of the following values, to tell
\fBgl_get_line()\fR how to proceed.
.sp
.ne 2
.na
\fB\fBGLA_ABORT\fR\fR
.ad
.RS 16n
Cause \fBgl_get_line()\fR to return \fINULL\fR.
.RE

```
.sp
.ne 2
.na
\fB\fBGLA_RETURN\fR\fR
.ad
.RS 16n
Cause \fBgl_get_line()\fR to return the completed input line
.RE

.sp
.ne 2
.na
\fB\fBGLA_CONTINUE\fR\fR
.ad
.RS 16n
Resume command-line editing.
.RE

.sp
.LP
Note that the \fIname\fR argument of \fBgl_register_action()\fR specifies the
name by which a user can refer to the action in their configuration file. This
allows them to re-bind the action to an alternate key-sequence. In order for
this to work, it is necessary to call \fBgl_register_action()\fR between
calling \fBnew_GetLine()\fR and the first call to \fBgl_get_line()\fR.
.SS "History Files"
.LP
To save the contents of the history buffer before quitting your application and
subsequently restore them when you next start the application, the
\fBgl_save_history()\fR and \fBgl_load_history()\fR functions are provided.
.sp
.LP
The \fIfilename\fR argument specifies the name to give the history file when
saving, or the name of an existing history file, when loading. This may contain
home directory and environment variable expressions, such as
\fB~/.myapp_history\fR or \fB$HOME/.myapp_history\fR.
.sp
.LP
Along with each history line, additional information about it, such as its
nesting level and when it was entered by the user, is recorded as a comment
preceding the line in the history file. Writing this as a comment allows the
history file to double as a command file, just in case you wish to replay a
whole session using it. Since comment prefixes differ in different languages,
the comment argument is provided for specifying the comment prefix. For
example, if your application were a UNIX shell, such as the Bourne shell, you
would specify "#" here. Whatever you choose for the comment character, you must
specify the same prefix to \fBgl_load_history()\fR that you used when you
called \fBgl_save_history()\fR to write the history file.
.sp
.LP
The \fImax_lines\fR argument must be either -1 to specify that all lines in the
history list be saved, or a positive number specifying a ceiling on how many of
the most recent lines should be saved.
.sp
.LP
Both functions return non-zero on error, after writing an error message to
Both fuctions return non-zero on error, after writing an error message to
\fBstderr\fR. Note that \fBgl_load_history()\fR does not consider the
non-existence of a file to be an error.
.SS "Multiple History Lists"
.LP
If your application uses a single \fBGetLine\fR object for entering many
different types of input lines, you might want \fBgl_get_line()\fR to
distinguish the different types of lines in the history list, and only recall
lines that match the current type of line. To support this requirement,
```



```
\fBgl_get_line()\fR marks lines being recorded in the history list with an
integer identifier chosen by the application. Initially this identifier is set
to 0 by \fBnew_GetLine()\fR, but it can be changed subsequently by calling
\fBgl_group_history()\fR.
.sp
.LP
The integer identifier ID can be any number chosen by the application, but note
that \fBgl_save_history()\fR and \fBgl_load_history()\fR preserve the
association between identifiers and historical input lines between program
invocations, so you should choose fixed identifiers for the different types of
input line used by your application.
.sp
.LP
Whenever \fBgl_get_line()\fR appends a new input line to the history list, the
current history identifier is recorded with it, and when it is asked to recall
a historical input line, it only recalls lines that are marked with the current
identifier.
.SS "Displaying History"
.LP
The history list can be displayed by calling \fBgl_show_history()\fR. This
function displays the current contents of the history list to the \fBstdio\fR
output stream \fIfp\fR. If the \fImax_lines\fR argument is greater than or
equal to zero, then no more than this number of  the most recent lines will be
displayed. If the \fIall_groups\fR argument is non-zero, lines from all history
groups are displayed. Otherwise only those of the currently selected history
group are displayed. The format string argument, \fIfmt\fR, determines how the
line is displayed. This can contain arbitrary characters which are written
verbatim, interleaved with any of the following format directives:
.sp
.ne 2
.na
\fB\fB%D\fR\fR
.ad
.RS 6n
The date on which the line was originally entered, formatted like 2001-11-20.
.RE

.sp
.ne 2
.na
\fB\fB%T\fR\fR
.ad
.RS 6n
The time of day when the line was entered, formatted like 23:59:59.
.RE

.sp
.ne 2
.na
\fB\fB%N\fR\fR
.ad
.RS 6n
The sequential entry number of the line in the history buffer.
.RE

.sp
.ne 2
.na
\fB\fB%G\fR\fR
.ad
.RS 6n
The number of the history group which the line belongs to.
.RE

.sp
.ne 2
```

```
.na
\fB\fB%%\fR\fR
.ad
.RS 6n
A literal % character.
.RE

.sp
.ne 2
.na
\fB\fB%H\fR\fR
.ad
.RS 6n
The history line itself.
.RE

.sp
.LP
Thus a format string like "%D %T %H0" would output something like:
.sp
.in +2
.nf
2001-11-20 10:23:34  Hello world
.fi
.in -2

.sp
.LP
Note the inclusion of an explicit newline character in the format string.
.SS "Looking Up History"
.LP
The \fBgl_lookup_history()\fR function allows the calling application to look
up lines in the history list.
.sp
.LP
The \fIid\fR argument indicates which line to look up, where the first line
that was entered in the history list after \fBnew_GetLine()\fR was called is
denoted by 0, and subsequently entered lines are denoted with successively
higher numbers. Note that the range of lines currently preserved in the history
list can be queried by calling the \fBgl_range_of_history()\fR function. If the
requested line is in the history list, the details of the line are recorded in
the variable pointed to by the \fIhline\fR argument, and 1 is returned.
Otherwise 0 is returned, and the variable pointed to by \fIhline\fR is left
unchanged.
.sp
.LP
Beware that the string returned in \fIhline\fR->\fIline\fR is part of the
history buffer, so it must not be modified by the caller, and will be recycled
on the next call to any function that takes \fIgl\fR as its argument. Therefore
you should make a private copy of this string if you need to keep it.
.SS "Manual History Archival"
.LP
By default, whenever a line is entered by the user, it is automatically
appended to the history list, just before \fBgl_get_line()\fR returns the line
to the caller. This is convenient for the majority of applications, but there
are also applications that need finer-grained control over what gets added to
the history list. In such cases, the automatic addition of entered lines to the
history list can be turned off by calling the \fBgl_automatic_history()\fR
function.
.sp
.LP
If this function is called with its \fIenable\fR argument set to 0,
\fBgl_get_line()\fR will not automatically archive subsequently entered lines.
Automatic archiving can be reenabled at a later time by calling this function
again, with its \fIenable\fR argument set to 1. While automatic history
archiving is disabled, the calling application can use the
```



```
\fBgl_append_history()\fR to append lines to the history list as needed.
.sp
.LP
The \fIline\fR argument specifies the line to be added to the history list.
This must be a normal '\e0 ' terminated string. If this string contains any
newline characters, the line that gets archived in the history list will be
terminated by the first of these. Otherwise it will be terminated by the '\e0 '
terminator. If the line is longer than the maximum input line length that was
specified when \fBnew_GetLine()\fR was called, it will be truncated to the
actual \fBgl_get_line()\fR line length when the line is recalled.
.sp
.LP
If successful, \fBgl_append_history()\fR returns 0. Otherwise it returns
non-zero and sets \fBerrno\fR to one of the following values.
.sp
.ne 2
.na
\fB\fBEINVAL\fR\fR
.ad
.RS 10n
One of the arguments passed to \fBgl_append_history()\fR was \fINULL\fR.
.RE

.sp
.ne 2
.na
\fB\fBENOMEM\fR\fR
.ad
.RS 10n
The specified line was longer than the allocated size of the history buffer (as
specified when \fBnew_GetLine()\fR was called), so it could not be archived.
.RE

.sp
.LP
A textual description of the error can optionally be obtained by calling
\fBgl_error_message()\fR. Note that after such an error, the history list
remains in a valid state to receive new history lines, so there is little harm
in simply ignoring the return status of \fBgl_append_history()\fR.
.SS "Miscellaneous History Configuration"
.LP
If you wish to change the size of the history buffer that was originally
specified in the call to \fBnew_GetLine()\fR, you can do so with the
\fBgl_resize_history()\fR function.
.sp
.LP
The \fIhistlen\fR argument specifies the new size in bytes, and if you specify
this as 0, the buffer will be deleted.
.sp
.LP
As mentioned in the discussion of \fBnew_GetLine()\fR, the number of lines that
can be stored in the history buffer, depends on the lengths of the individual
lines. For example, a 1000 byte buffer could equally store 10 lines of average
length 100 bytes, or 20 lines of average length 50 bytes. Although the buffer
is never expanded when new lines are added, a list of pointers into the buffer
does get expanded when needed to accommodate the number of lines currently
stored in the buffer. To place an upper limit on the number of lines in the
buffer, and thus a ceiling on the amount of memory used in this list, you can
call the \fBgl_limit_history()\fR function.
.sp
.LP
The \fImax_lines\fR should either be a positive number >= 0, specifying an
upper limit on the number of lines in the buffer, or be -1 to cancel any
previously specified limit. When a limit is in effect, only the \fImax_lines\fR
most recently appended lines are kept in the buffer. Older lines are discarded.
.sp
```

```
.LP
To discard lines from the history buffer, use the \fBgl_clear_history()\fR
function.
.sp
.LP
The \fIall_groups\fR argument tells the function whether to delete just the
lines associated with the current history group (see \fBgl_group_history()\fR)
or all historical lines in the buffer.
.sp
.LP
The \fBgl_toggle_history()\fR function allows you to toggle history on and off
without losing the current contents of the history list.
.sp
.LP
Setting the \fIenable\fR argument to 0 turns off the history mechanism, and
setting it to 1 turns it back on. When history is turned off, no new lines will
be added to the history list, and history lookup key-bindings will act as
though there is nothing in the history buffer.
.SS "Querying History Information"
.LP
The configured state of the history list can be queried with the
\fBgl_history_state()\fR function. On return, the status information is
recorded in the variable pointed to by the \fIstate\fR argument.
.sp
.LP
The \fBgl_range_of_history()\fR function returns the number and range of lines
in the history list. The return values are recorded in the variable pointed to
by the range argument. If the \fIlines\fR member of this structure is greater
than zero, then the oldest and newest members report the range of lines in the
list, and \fInewest\fR=\fIoldest\fR+\fIlines\fR-1. Otherwise they are both
zero.
.sp
.LP
The \fBgl_size_of_history()\fR function returns the total size of the history
buffer and the amount of the buffer that is currently occupied.
.sp
.LP
On return, the size information is recorded in the variable pointed to by the
\fIsize\fR argument.
.SS "Changing Terminals"
.LP
The \fBnew_GetLine()\fR constructor function assumes that input is to be read
from \fBstdin\fR and output written to \fBstdout\fR. The following function
allows you to switch to different input and output streams.
.sp
.LP
The \fIgl\fR argument is the object that was returned by \fBnew_GetLine()\fR.
The \fIinput_fp\fR argument specifies the stream to read from, and
\fIoutput_fp\fR specifies the stream to be written to. Only if both of these
refer to a terminal, will interactive terminal input be enabled. Otherwise
\fBgl_get_line()\fR will simply call \fBfgets()\fR to read command input. If
both streams refer to a terminal, then they must refer to the same terminal,
and the type of this terminal must be specified with the \fIterm\fR argument.
The value of the \fIterm\fR argument is looked up in the terminal information
database (\fBterminfo\fR or \fBtermcap\fR), in order to determine which special
control sequences are needed to control various aspects of the terminal.
\fBnew_GetLine()\fR for example, passes the return value of
\fBgetenv\fR("TERM") in this argument. Note that if one or both of
\fIinput_fp\fR and \fIoutput_fp\fR do not refer to a terminal, then it is legal
to pass \fINULL\fR instead of a terminal type.
.sp
.LP
Note that if you want to pass file descriptors to \fBgl_change_terminal()\fR,
you can do this by creating \fBstdio\fR stream wrappers using the POSIX
\fBfdopen\fR(3C) function.
.SS "External Event Handling"
```



```
.LP
By default, \fBgl_get_line()\fR does not return until either a complete input
line has been entered by the user, or an error occurs. In programs that need to
watch for I/O from other sources than the terminal, there are two options.
.RS +4
.TP
.ie t \(bu
.el o
Use the functions described in the \fBgl_io_mode\fR(3TECLA) manual page to
switch \fBgl_get_line()\fR into non-blocking server mode. In this mode,
\fBgl_get_line()\fR becomes a non-blocking, incremental line-editing function
that can safely be called from an external event loop. Although this is a very
versatile method, it involves taking on some responsibilities that are normally
performed behind the scenes by \fBgl_get_line()\fR.
.RE
.RS +4
.TP
.ie t \(bu
.el o
While \fBgl_get_line()\fR is waiting for keyboard input from the user, you can
ask it to also watch for activity on arbitrary file descriptors, such as
network sockets or pipes, and have it call functions of your choosing when
activity is seen. This works on any system that has the select system call,
which is most, if not all flavors of UNIX.
.RE
.sp
.LP
Registering a file descriptor to be watched by \fBgl_get_line()\fR involves
calling the \fBgl_watch_fd()\fR function. If this returns non-zero, then it
means that either your arguments are invalid, or that this facility is not
supported on the host system.
.sp
.LP
The \fIfd\fR argument is the file descriptor to be watched. The event argument
specifies what type of activity is of interest, chosen from the following
enumerated values:
.sp
.ne 2
.na
\fB\fBGLFD_READ\fR\fR
.ad
.RS 15n
Watch for the arrival of data to be read.
.RE

.sp
.ne 2
.na
\fB\fBGLFD_WRITE\fR\fR
.ad
.RS 15n
Watch for the ability to write to the file descriptor without blocking.
.RE

.sp
.ne 2
.na
\fB\fBGLFD_URGENT\fR\fR
.ad
.RS 15n
Watch for the arrival of urgent out-of-band data on the file descriptor.
.RE

.sp
.LP
The \fIcallback\fR argument is the function to call when the selected activity
```

is seen. It should be defined with the following macro, which is defined in
libtecla.h.
.sp
.in +2
.nf
#define GL_FD_EVENT_FN(fn) GlFdStatus (fn)(GetLine *gl, \
                                   void *data, int fd, GlFdEvent event)
.fi
.in -2

.sp
.LP
The data argument of the \fBgl_watch_fd()\fR function is passed to the callback
function for its own use, and can point to anything you like, including
\fINULL\fR. The file descriptor and the event argument are also passed to the
callback function, and this potentially allows the same callback function to be
registered to more than one type of event and/or more than one file descriptor.
The return value of the callback function should be one of the following
values.
.sp
.ne 2
.na
\fB\fBGLFD_ABORT\fR\fR
.ad
.RS 17n
Tell \fBgl_get_line()\fR to abort. When this happens, \fBgl_get_line()\fR
returns \fINULL\fR, and a following call to \fBgl_return_status()\fR will
return \fBGLR_FDABORT\fR. Note that if the application needs \fBerrno\fR always
to have a meaningful value when \fBgl_get_line()\fR returns \fINULL\fR, the
callback function should set \fBerrno\fR appropriately.
.RE

.sp
.ne 2
.na
\fB\fBGLFD_REFRESH\fR\fR
.ad
.RS 17n
Redraw the input line then continue waiting for input. Return this if your
callback wrote to the terminal.
.RE

.sp
.ne 2
.na
\fB\fBGLFD_CONTINUE\fR\fR
.ad
.RS 17n
Continue to wait for input, without redrawing the line.
.RE

.sp
.LP
Note that before calling the callback, \fBgl_get_line()\fR blocks most signals
and leaves its own signal handlers installed, so if you need to catch a
particular signal you will need to both temporarily install your own signal
handler, and unblock the signal. Be sure to re-block the signal (if it was
originally blocked) and reinstate the original signal handler, if any, before
returning.
.sp
.LP
Your callback should not try to read from the terminal, which is left in raw
mode as far as input is concerned. You can write to the terminal as usual,
since features like conversion of newline to carriage-return/linefeed are
re-enabled while the callback is running. If your callback function does write
to the terminal, be sure to output a newline first, and when your callback

returns, tell \fBgl_get_line()\fR that the input line needs to be redrawn, by
returning the \fBGLFD_REFRESH\fR status code.
.sp
.LP
To remove a callback function that you previously registered for a given file
descriptor and event, simply call \fBgl_watch_fd()\fR with the same \fIfd\fR
and \fIevent\fR arguments, but with a \fIcallback\fR argument of 0. The
\fIdata\fR argument is ignored in this case.
.SS "Setting An Inactivity Timeout"
.LP
The \fBgl_inactivity_timeout()\fR function can be used to set or cancel an
inactivity timeout. Inactivity in this case refers both to keyboard input, and
to I/O on any file descriptors registered by prior and subsequent calls to
\fBgl_watch_fd()\fR.
.sp
.LP
The timeout is specified in the form of an integral number of seconds and an
integral number of nanoseconds, specified by the \fIsec\fR and \fInsec\fR
arguments, respectively. Subsequently, whenever no activity is seen for this
time period, the function specified by the \fIcallback\fR argument is called.
The \fIdata\fR argument of \fBgl_inactivity_timeout()\fR is passed to this
callback function whenever it is invoked, and can thus be used to pass
arbitrary application-specific information to the callback. The following macro
is provided in <\fBlibtecla.h\fR> for applications to use to declare and
prototype timeout callback functions.
.sp
.in +2
.nf
#define GL_TIMEOUT_FN(fn) GlAfterTimeout (fn)(GetLine *gl, void *data)
.fi
.in -2

.sp
.LP
On returning, the application’s callback is expected to return one of the
following enumerators to tell \fBgl_get_line()\fR how to proceed after the
timeout has been handled by the callback.
.sp
.ne 2
.na
\fB\fBGLTO_ABORT\fR\fR
.ad
.RS 17n
Tell \fBgl_get_line()\fR to abort. When this happens, \fBgl_get_line()\fR will
return \fINULL\fR, and a following call to \fBgl_return_status()\fR will return
\fBGLR_TIMEOUT\fR. Note that if the application needs \fBerrno\fR always to
have a meaningful value when \fBgl_get_line()\fR returns \fINULL\fR, the
callback function should set \fBerrno\fR appropriately.
.RE

.sp
.ne 2
.na
\fB\fBGLTO_REFRESH\fR\fR
.ad
.RS 17n
Redraw the input line, then continue waiting for input. You should return this
value if your callback wrote to the terminal.
.RE

.sp
.ne 2
.na
\fB\fBGLTO_CONTINUE\fR\fR
.ad
.RS 17n

In normal blocking-I/O mode, continue to wait for input, without redrawing the user's input line. In non-blocking server I/O mode (see \fBgl_io_mode\fR(3TECLA)), \fBgl_get_line()\fR acts as though I/O blocked. This means that \fBgl_get_line()\fR will immediately return \fINULL\fR, and a following call to \fBgl_return_status()\fR will return \fBGLR_BLOCKED\fR.
.RE

.sp
.LP
Note that before calling the callback, \fBgl_get_line()\fR blocks most signals and leaves its own signal handlers installed, so if you need to catch a particular signal you will need to both temporarily install your own signal handler and unblock the signal. Be sure to re-block the signal (if it was originally blocked) and reinstate the original signal handler, if any, before returning.
.sp
.LP
Your callback should not try to read from the terminal, which is left in raw mode as far as input is concerned. You can however write to the terminal as usual, since features like conversion of newline to carriage-return/linefeed are re-enabled while the callback is running. If your callback function does write to the terminal, be sure to output a newline first, and when your callback returns, tell \fBgl_get_line()\fR that the input line needs to be redrawn, by returning the \fBGLTO_REFRESH\fR status code.
.sp
.LP
Finally, note that although the timeout arguments include a nanosecond component, few computer clocks presently have resolutions that are finer than a few milliseconds, so asking for less than a few milliseconds is equivalent to requesting zero seconds on many systems. If this would be a problem, you should base your timeout selection on the actual resolution of the host clock (for example, by calling \fBsysconf\fR(\fB_SC_CLK_TCK\fR)).
.sp
.LP
To turn off timeouts, simply call \fBgl_inactivity_timeout()\fR with a \fIcallback\fR argument of 0. The \fIdata\fR argument is ignored in this case.
.SS "Signal Handling Defaults"
.LP
By default, the \fBgl_get_line()\fR function intercepts a number of signals. This is particularly important for signals that would by default terminate the process, since the terminal needs to be restored to a usable state before this happens. This section describes the signals that are trapped by default and how \fBgl_get_line()\fR responds to them. Changing these defaults is the topic of the following section.
.sp
.LP
When the following subset of signals are caught, \fBgl_get_line()\fR first restores the terminal settings and signal handling to how they were before \fBgl_get_line()\fR was called, resends the signal to allow the calling application's signal handlers to handle it, then, if the process still exists, returns \fINULL\fR and sets \fBerrno\fR as specified below.
.sp
.ne 2
.na
\fB\fBSIGINT\fR\fR
.ad
.RS 11n
This signal is generated both by the keyboard interrupt key (usually \fB^C\fR), and the keyboard break key. The \fBerrno\fR value is \fBEINTR\fR.
.RE

.sp
.ne 2
.na
\fB\fBSIGHUP\fR\fR
.ad

.RS 11n
This signal is generated when the controlling terminal exits. The \fBerrno\fR value is \fBENOTTY\fR.
.RE

.sp
.ne 2
.na
\fB\fBSIGPIPE\fR\fR
.ad
.RS 11n
This signal is generated when a program attempts to write to a pipe whose remote end is not being read by any process. This can happen for example if you have called \fBgl_change_terminal()\fR to redirect output to a pipe hidden under a pseudo terminal. The \fBerrno\fR value is \fBEPIPE\fR.
.RE

.sp
.ne 2
.na
\fB\fBSIGQUIT\fR\fR
.ad
.RS 11n
This signal is generated by the keyboard quit key (usually \fB^\e\fR). The \fBerrno\fR value is \fBEINTR\fR.
.RE

.sp
.ne 2
.na
\fB\fBSIGABRT\fR\fR
.ad
.RS 11n
This signal is generated by the standard C, abort function. By default it both terminates the process and generates a core dump. The \fBerrno\fR value is \fBEINTR\fR.
.RE

.sp
.ne 2
.na
\fB\fBSIGTERM\fR\fR
.ad
.RS 11n
This is the default signal that the UNIX kill command sends to processes. The \fBerrno\fR value is \fBEINTR\fR.
.RE

.sp
.LP
Note that in the case of all of the above signals, POSIX mandates that by default the process is terminated, with the addition of a core dump in the case of the \fBSIGQUIT\fR signal. In other words, if the calling application does not override the default handler by supplying its own signal handler, receipt of the corresponding signal will terminate the application before \fBgl_get_line()\fR returns.
.sp
.LP
If \fBgl_get_line()\fR aborts with \fBerrno\fR set to \fBEINTR\fR, you can find out what signal caused it to abort, by calling the \fBgl_last_signal()\fR function. This returns the numeric code (for example, \fBSIGINT\fR) of the last signal that was received during the most recent call to \fBgl_get_line()\fR, or -1 if no signals were received.
.sp
.LP
On systems that support it, when a \fBSIGWINCH\fR (window change) signal is

```
1551 received, \fBgl_get_line()\fR queries the terminal to find out its new size,
1552 redraws the current input line to accommodate the new size, then returns to
1553 waiting for keyboard input from the user. Unlike other signals, this signal is
1554 not resent to the application.
1555 .sp
1556 .LP
1557 Finally, the following signals cause \fBgl_get_line()\fR to first restore the
1558 terminal and signal environment to that which prevailed before
1559 \fBgl_get_line()\fR was called, then resend the signal to the application. If
1560 the process still exists after the signal has been delivered, then
1561 \fBgl_get_line()\fR then re-establishes its own signal handlers, switches the
1562 terminal back to raw mode, redisplays the input line, and goes back to awaiting
1563 terminal input from the user.
1564 .sp
1565 .ne 2
1566 .na
1567 \fB\fBSIGCONT\fR\fR
1568 .ad
1569 .RS 13n
1570 This signal is generated when a suspended process is resumed.
1571 .RE

1573 .sp
1574 .ne 2
1575 .na
1576 \fB\fBSIGPOLL\fR\fR
1577 .ad
1578 .RS 13n
1579 On SVR4 systems, this signal notifies the process of an asynchronous I/O event.
1580 Note that under 4.3+BSD, \fBSIGIO\fR and \fBSIGPOLL\fR are the same. On other
1581 systems, \fBSIGIO\fR is ignored by default, so \fBgl_get_line()\fR does not
1582 trap it by default.
1583 .RE

1585 .sp
1586 .ne 2
1587 .na
1588 \fB\fBSIGPWR\fR\fR
1589 .ad
1590 .RS 13n
1591 This signal is generated when a power failure occurs (presumably when the
1592 system is on a UPS).
1593 .RE

1595 .sp
1596 .ne 2
1597 .na
1598 \fB\fBSIGALRM\fR\fR
1599 .ad
1600 .RS 13n
1601 This signal is generated when a timer expires.
1602 .RE

1604 .sp
1605 .ne 2
1606 .na
1607 \fB\fBSIGUSR1\fR\fR
1608 .ad
1609 .RS 13n
1610 An application specific signal.
1611 .RE

1613 .sp
1614 .ne 2
1615 .na
1616 \fB\fBSIGUSR2\fR\fR
```



```
1617 .ad
1618 .RS 13n
1619 Another application specific signal.
1620 .RE

1622 .sp
1623 .ne 2
1624 .na
1625 \fB\fBSIGVTALRM\fR\fR
1626 .ad
1627 .RS 13n
1628 This signal is generated when a virtual timer expires. See \fBsetitimer\fR(2).
1629 .RE

1631 .sp
1632 .ne 2
1633 .na
1634 \fB\fBSIGXCPU\fR\fR
1635 .ad
1636 .RS 13n
1637 This signal is generated when a process exceeds its soft CPU time limit.
1638 .RE

1640 .sp
1641 .ne 2
1642 .na
1643 \fB\fBSIGXFSZ\fR\fR
1644 .ad
1645 .RS 13n
1646 This signal is generated when a process exceeds its soft file-size limit.
1647 .RE

1649 .sp
1650 .ne 2
1651 .na
1652 \fB\fBSIGTSTP\fR\fR
1653 .ad
1654 .RS 13n
1655 This signal is generated by the terminal suspend key, which is usually
1656 \fB^Z\fR, or the delayed terminal suspend key, which is usually \fB^Y\fR.
1657 .RE

1659 .sp
1660 .ne 2
1661 .na
1662 \fB\fBSIGTTIN\fR\fR
1663 .ad
1664 .RS 13n
1665 This signal is generated if the program attempts to read from the terminal
1666 while the program is running in the background.
1667 .RE

1669 .sp
1670 .ne 2
1671 .na
1672 \fB\fBSIGTTOU\fR\fR
1673 .ad
1674 .RS 13n
1675 This signal is generated if the program attempts to write to the terminal while
1676 the program is running in the background.
1677 .RE

1679 .sp
1680 .LP
1681 Obviously not all of the above signals are supported on all systems, so code to
1682 support them is conditionally compiled into the tecla library.
```

```
.sp
.LP
Note that if \fBSIGKILL\fR or \fBSIGPOLL\fR, which by definition cannot be
caught, or any of the hardware generated exception signals, such as
\fBSIGSEGV\fR, \fBSIGBUS\fR, and \fBSIGFPE\fR, are received and unhandled while
\fBgl_get_line()\fR has the terminal in raw mode, the program will be
terminated without the terminal having been restored to a usable state. In
practice, job-control shells usually reset the terminal settings when a process
relinquishes the controlling terminal, so this is only a problem with older
shells.
.SS "Customized Signal Handling"
.LP
The previous section listed the signals that \fBgl_get_line()\fR traps by
default, and described how it responds to them. This section describes how to
both add and remove signals from the list of trapped signals, and how to
specify how \fBgl_get_line()\fR should respond to a given signal.
.sp
.LP
If you do not need \fBgl_get_line()\fR to do anything in response to a signal
that it normally traps, you can tell to \fBgl_get_line()\fR to ignore that
signal by calling \fBgl_ignore_signal()\fR.
.sp
.LP
The \fIsigno\fR argument is the number of the signal (for example,
\fBSIGINT\fR) that you want to have ignored. If the specified signal is not
currently one of those being trapped, this function does nothing.
.sp
.LP
The \fBgl_trap_signal()\fR function allows you to either add a new signal to
the list that \fBgl_get_line()\fR traps or modify how it responds to a signal
that it already traps.
.sp
.LP
The \fIsigno\fR argument is the number of the signal that you want to have
trapped. The \fIflags\fR argument is a set of flags that determine the
environment in which the application's signal handler is invoked. The
\fIafter\fR argument tells \fBgl_get_line()\fR what to do after the
application's signal handler returns. The \fIerrno_value\fR tells
\fBgl_get_line()\fR what to set \fBerrno\fR to if told to abort.
.sp
.LP
The \fIflags\fR argument is a bitwise OR of zero or more of the following
enumerators:
.sp
.ne 2
.na
\fB\fBGLS_RESTORE_SIG\fR\fR
.ad
.RS 20n
Restore the caller's signal environment while handling the signal.
.RE

.sp
.ne 2
.na
\fB\fBGLS_RESTORE_TTY\fR\fR
.ad
.RS 20n
Restore the caller's terminal settings while handling the signal.
.RE

.sp
.ne 2
.na
\fB\fBGLS_RESTORE_LINE\fR\fR
.ad
```



```
.RS 20n
Move the cursor to the start of the line following the input line before
invoking the application's signal handler.
.RE

.sp
.ne 2
.na
\fB\fBGLS_REDRAW_LINE\fR\fR
.ad
.RS 20n
Redraw the input line when the application's signal handler returns.
.RE

.sp
.ne 2
.na
\fB\fBGLS_UNBLOCK_SIG\fR\fR
.ad
.RS 20n
Normally, if the calling program has a signal blocked (see
\fBsigprocmask\fR(2)), \fBgl_get_line()\fR does not trap that signal. This flag
tells \fBgl_get_line()\fR to trap the signal and unblock it for the duration of
the call to \fBgl_get_line()\fR.
.RE

.sp
.ne 2
.na
\fB\fBGLS_DONT_FORWARD\fR\fR
.ad
.RS 20n
If this flag is included, the signal will not be forwarded to the signal
handler of the calling program.
.RE

.sp
.LP
Two commonly useful flag combinations are also enumerated as follows:
.sp
.ne 2
.na
\fB\fBGLS_RESTORE_ENV\fR\fR
.ad
.RS 21n
\fBGLS_RESTORE_SIG\fR | \fBGLS_RESTORE_TTY\fR |\fBGLS_REDRAW_LINE\fR
.RE

.sp
.ne 2
.na
\fB\fBGLS_SUSPEND_INPUT\fR\fR
.ad
.RS 21n
\fBGLS_RESTORE_ENV\fR | \fBGLS_RESTORE_LINE\fR
.RE

.sp
.LP
If your signal handler, or the default system signal handler for this signal,
if you have not overridden it, never either writes to the terminal, nor
suspends or terminates the calling program, then you can safely set the
\fIflags\fR argument to 0.
.RS +4
.TP
.ie t \(bu
```

```
1814 .el o
1815 The cursor does not get left in the middle of the input line.
1816 .RE
1817 .RS +4
1818 .TP
1819 .ie t \(bu
1820 .el o
1821 So that the user can type in input and have it echoed.
1822 .RE
1823 .RS +4
1824 .TP
1825 .ie t \(bu
1826 .el o
1827 So that you do not need to end each output line with \er\en, instead of just
1828 \en.
1829 .RE
1830 .sp
1831 .LP
1832 The \fBGL_RESTORE_ENV\fR combination is the same as \fBGL_SUSPEND_INPUT\fR,
1833 except that it does not move the cursor. If your signal handler does not read
1834 or write anything to the terminal, the user will not see any visible indication
1835 that a signal was caught. This can be useful if you have a signal handler that
1836 only occasionally writes to the terminal, where using \fBGL_SUSPEND_LINE\fR
1837 would cause the input line to be unnecessarily duplicated when nothing had been
1838 written to the terminal. Such a signal handler, when it does write to the
1839 terminal, should be sure to start a new line at the start of its first write,
1840 by writing a new line before returning. If the signal arrives while the user is
1841 entering a line that only occupies a signal terminal line, or if the cursor is
1842 on the last terminal line of a longer input line, this will have the same
1843 effect as \fBGL_SUSPEND_INPUT\fR. Otherwise it will start writing on a line
1844 that already contains part of the displayed input line. This does not do any
1845 harm, but it looks a bit ugly, which is why the \fBGL_SUSPEND_INPUT\fR
1846 combination is better if you know that you are always going to be writing to
1868 combination is better if you know that you are always going to be writting to
1847 the terminal.
1848 .sp
1849 .LP
1850 The \fIafter\fR argument, which determines what \fBgl_get_line()\fR does after
1851 the application's signal handler returns (if  it returns), can take any one of
1852 the following values:
1853 .sp
1854 .ne 2
1855 .na
1856 \fB\fBGLS_RETURN\fR\fR
1857 .ad
1858 .RS 16n
1859 Return the completed input line, just as though the user had pressed the return
1860 key.
1861 .RE

1863 .sp
1864 .ne 2
1865 .na
1866 \fB\fBGLS_ABORT\fR\fR
1867 .ad
1868 .RS 16n
1869 Cause \fBgl_get_line()\fR to abort. When this happens, \fBgl_get_line()\fR
1870 returns \fINULL\fR, and a following call to \fBgl_return_status()\fR will
1871 return \fBGLR_SIGNAL\fR. Note that if the application needs \fBerrno\fR always
1872 to have a meaningful value when \fBgl_get_line()\fR returns \fINULL\fR, the
1873 callback function should set \fBerrno\fR appropriately.
1874 .RE

1876 .sp
1877 .ne 2
1878 .na
```



```
1879 \fB\fBGLS_CONTINUE\fR\fR
1880 .ad
1881 .RS 16n
1882 Resume command line editing.
1883 .RE

1885 .sp
1886 .LP
1887 The \fIerrno_value\fR argument is intended to be combined with the
1888 \fBGLS_ABORT\fR option, telling \fBgl_get_line()\fR what to set the standard
1889 \fBerrno\fR variable to before returning \fINULL\fR to the calling program. It
1890 can also, however, be used with the \fBGL_RETURN\fR option, in case you want to
1891 have a way to distinguish between an input line that was entered using the
1892 return key, and one that was entered by the receipt of a signal.
1893 .SS "Reliable Signal Handling"
1916 .LP
1894 Signal handling is surprisingly hard to do reliably without race conditions. In
1895 \fBgl_get_line()\fR a lot of care has been taken to allow applications to
1896 perform reliable signal handling around \fBgl_get_line()\fR. This section
1897 explains how to make use of this.
1898 .sp
1899 .LP
1900 As an example of the problems that can arise if the application is not written
1901 correctly, imagine that one's application has a \fBSIGINT\fR signal handler
1902 that sets a global flag. Now suppose that the application tests this flag just
1903 before invoking \fBgl_get_line()\fR. If a \fBSIGINT\fR signal happens to be
1904 received in the small window of time between the statement that tests the value
1905 of this flag, and the statement that calls \fBgl_get_line()\fR, then
1906 \fBgl_get_line()\fR will not see the signal, and will not be interrupted. As a
1907 result, the application will not be able to respond to the signal until the
1908 user gets around to finishing entering the input line and \fBgl_get_line()\fR
1909 returns. Depending on the application, this might or might not be a disaster,
1910 but at the very least it would puzzle the user.
1911 .sp
1912 .LP
1913 The way to avoid such problems is to do the following.
1914 .RS +4
1915 .TP
1916 1.
1917 If needed, use the \fBgl_trap_signal()\fR function to configure
1918 \fBgl_get_line()\fR to abort when important signals are caught.
1919 .RE
1920 .RS +4
1921 .TP
1922 2.
1923 Configure \fBgl_get_line()\fR such that if any of the signals that it
1924 catches are blocked when \fBgl_get_line()\fR is called, they will be unblocked
1925 automatically during times when \fBgl_get_line()\fR is waiting for I/O. This
1926 can be done either on a per signal basis, by calling the \fBgl_trap_signal()\fR
1927 function, and specifying the \fBGLS_UNBLOCK\fR attribute of the signal, or
1928 globally by calling the \fBgl_catch_blocked()\fR function. This function simply
1929 adds the \fBGLS_UNBLOCK\fR attribute to all of the signals that it is currently
1930 configured to trap.
1931 .RE
1932 .RS +4
1933 .TP
1934 3.
1935 Just before calling \fBgl_get_line()\fR, block delivery of all of the
1936 signals that \fBgl_get_line()\fR is configured to trap. This can be done using
1937 the POSIX sigprocmask function in conjunction with the \fBgl_list_signals()\fR
1938 function. This function returns the set of signals that it is currently
1939 configured to catch in the set argument, which is in the form required by
1940 \fBsigprocmask\fR(2).
1941 .RE
1942 .RS +4
1943 .TP
```

4.
In the example, one would now test the global flag that the signal handler sets, knowing that there is now no danger of this flag being set again until \fBgl_get_line()\fR unblocks its signals while performing I/O.
.RE
.RS +4
.TP
5.
Eventually \fBgl_get_line()\fR returns, either because a signal was caught, an error occurred, or the user finished entering their input line.
.RE
.RS +4
.TP
6.
Now one would check the global signal flag again, and if it is set, respond to it, and zero the flag.
.RE
.RS +4
.TP
7.
Use \fBsigprocmask()\fR to unblock the signals that were blocked in step 3.
.RE
.sp
.LP
The same technique can be used around certain POSIX signal-aware functions, such as \fBsigsetjmp\fR(3C) and \fBsigsuspend\fR(2), and in particular, the former of these two functions can be used in conjunction with \fBsiglongjmp\fR(3C) to implement race-condition free signal handling around other long-running system calls. The \fBgl_get_line()\fR function manages to reliably trap signals around calls to functions like \fBread\fR(2) and \fBselect\fR(3C) without race conditions.
.sp
.LP
The \fBgl_get_line()\fR function first uses the POSIX \fBsigprocmask()\fR function to block the delivery of all of the signals that it is currently configured to catch. This is redundant if the application has already blocked them, but it does no harm. It undoes this step just before returning.
.sp
.LP
Whenever \fBgl_get_line()\fR needs to call read or select to wait for input from the user, it first calls the POSIX \fBsigsetjmp()\fR function, being sure to specify a non-zero value for its \fIsavemask\fR argument.
.sp
.LP
If \fBsigsetjmp()\fR returns zero, \fBgl_get_line()\fR then does the following.
.RS +4
.TP
1.
It uses the POSIX \fBsigaction\fR(2) function to register a temporary signal handler to all of the signals that it is configured to catch. This signal handler does two things.
.RS +4
.TP
a.
It records the number of the signal that was received in a file-scope variable.
.RE
.RS +4
.TP
b.
It then calls the POSIX \fBsiglongjmp()\fR function using the buffer that was passed to \fBsigsetjmp()\fR for its first argument and a non-zero value for its second argument.
.RE
When this signal handler is registered, the \fIsa_mask\fR member of the \fBstruct sigaction\fR \fIact\fR argument of the call to \fBsigaction()\fR is

configured to contain all of the signals that \fBgl_get_line()\fR is catching. This ensures that only one signal will be caught at once by our signal handler, which in turn ensures that multiple instances of our signal handler do not tread on each other's toes.
.RE
.RS +4
.TP
2.
Now that the signal handler has been set up, \fBgl_get_line()\fR unblocks all of the signals that it is configured to catch.
.RE
.RS +4
.TP
3.
It then calls the \fBread()\fR or \fBselect()\fR function to wait for keyboard input.
.RE
.RS +4
.TP
4.
If this function returns (that is, no signal is received), \fBgl_get_line()\fR blocks delivery of the signals of interest again.
.RE
.RS +4
.TP
5.
It then reinstates the signal handlers that were displaced by the one that was just installed.
.RE
.sp
.LP
Alternatively, if \fBsigsetjmp()\fR returns non-zero, this means that one of the signals being trapped was caught while the above steps were executing. When this happens, \fBgl_get_line()\fR does the following.
.sp
.LP
First, note that when a call to \fBsiglongjmp()\fR causes \fBsigsetjmp()\fR to return, provided that the \fIsavemask\fR argument of \fBsigsetjmp()\fR was non-zero, the signal process mask is restored to how it was when \fBsigsetjmp()\fR was called. This is the important difference between \fBsigsetjmp()\fR and the older problematic \fBsetjmp\fR(3C), and is the essential ingredient that makes it possible to avoid signal handling race conditions. Because of this we are guaranteed that all of the signals that we blocked before calling \fBsigsetjmp()\fR are blocked again as soon as any signal is caught. The following statements, which are then executed, are thus guaranteed to be executed without any further signals being caught.
.RS +4
.TP
1.
If so instructed by the \fBgl_get_line()\fR configuration attributes of the signal that was caught, \fBgl_get_line()\fR restores the terminal attributes to the state that they had when \fBgl_get_line()\fR was called. This is particularly important for signals that suspend or terminate the process, since otherwise the terminal would be left in an unusable state.
.RE
.RS +4
.TP
2.
It then reinstates the application's signal handlers.
.RE
.RS +4
.TP
3.
Then it uses the C standard-library \fBraise\fR(3C) function to re-send the application the signal that was caught.
.RE

```
.RS +4
.TP
4.
Next it unblocks delivery of the signal that we just sent. This results in
the signal that was just sent by \fBraise()\fR being caught by the
application's original signal handler, which can now handle it as it sees fit.
.RE
.RS +4
.TP
5.
If the signal handler returns (that is, it does not terminate the process),
\fBgl_get_line()\fR blocks delivery of the above signal again.
.RE
.RS +4
.TP
6.
It then undoes any actions performed in the first of the above steps and
redisplays the line, if the signal configuration calls for this.
.RE
.RS +4
.TP
7.
\fBgl_get_line()\fR then either resumes trying to read a character, or
aborts, depending on the configuration of the signal that was caught.
.RE
.sp
.LP
What the above steps do in essence is to take asynchronously delivered signals
and handle them synchronously, one at a time, at a point in the code where
\fBgl_get_line()\fR has complete control over its environment.
.SS "The Terminal Size"
.LP
On most systems the combination of the \fBTIOCGWINSZ\fR ioctl and the
\fBSIGWINCH\fR signal is used to maintain an accurate idea of the terminal
size. The terminal size is newly queried every time that \fBgl_get_line()\fR is
called and whenever a \fBSIGWINCH\fR signal is received.
.sp
.LP
On the few systems where this mechanism is not available, at startup
\fBnew_GetLine()\fR first looks for the \fBLINES\fR and \fBCOLUMNS\fR
environment variables. If these are not found, or they contain unusable values,
then if a terminal information database like \fBterminfo\fR or \fBtermcap\fR is
available, the default size of the terminal is looked up in this database. If
this too fails to provide the terminal size, a default size of 80 columns by 24
lines is used.
.sp
.LP
Even on systems that do support ioctl(\fBTIOCGWINSZ\fR), if the terminal is on
the other end of a serial line, the terminal driver generally has no way of
detecting when a resize occurs or of querying what the current size is. In such
cases no \fBSIGWINCH\fR is sent to the process, and the dimensions returned by
ioctl(\fBTIOCGWINSZ\fR) are not correct. The only way to handle such instances
is to provide a way for the user to enter a command that tells the remote
system what the new size is. This command would then call the
\fBgl_set_term_size()\fR function to tell \fBgl_get_line()\fR about the change
in size.
.sp
.LP
The \fIncolumn\fR and \fInline\fR arguments are used to specify the new
dimensions of the terminal, and must not be less than 1. On systems that do
support ioctl(\fBTIOCGWINSZ\fR), this function first calls
ioctl(\fBTIOCSWINSZ\fR) to tell the terminal driver about the change in size.
In non-blocking server-I/O mode, if a line is currently being input, the input
line is then redrawn to accommodate the changed size. Finally the new values are
recorded in \fIgl\fR for future use by \fBgl_get_line()\fR.
.sp
```



```
.LP
The \fBgl_terminal_size()\fR function allows you to query the current size of
the terminal, and install an alternate fallback size for cases where the size
is not available. Beware that the terminal size will not be available if
reading from a pipe or a file, so the default values can be important even on
systems that do support ways of finding out the terminal size.
.sp
.LP
This function first updates \fBgl_get_line()\fR's fallback terminal dimensions,
then records its findings in the return value.
.sp
.LP
The \fIdef_ncolumn\fR and \fIdef_nline\fR arguments specify the default number
of terminal columns and lines to use if the terminal size cannot be determined
by ioctl(\fBTIOCGWINSZ\fR) or environment variables.
.SS "Hiding What You Type"
.LP
When entering sensitive information, such as passwords, it is best not to have
the text that you are entering echoed on the terminal. Furthermore, such text
should not be recorded in the history list, since somebody finding your
terminal unattended could then recall it, or somebody snooping through your
directories could see it in your history file. With this in mind, the
\fBgl_echo_mode()\fR function allows you to toggle on and off the display and
archival of any text that is subsequently entered in calls to
\fBgl_get_line()\fR.
.sp
.LP
The \fIenable\fR argument specifies whether entered text should be visible or
not. If it is 0, then subsequently entered lines will not be visible on the
terminal, and will not be recorded in the history list. If it is 1, then
subsequent input lines will be displayed as they are entered, and provided that
history has not been turned off with a call to \fBgl_toggle_history()\fR, then
they will also be archived in the history list. Finally, if the enable argument
is -1, then the echoing mode is left unchanged, which allows you to
non-destructively query the current setting through the return value. In all
cases, the return value of the function is 0 if echoing was disabled before the
function was called, and 1 if it was enabled.
.sp
.LP
When echoing is turned off, note that although tab completion will invisibly
complete your prefix as far as possible, ambiguous completions will not be
displayed.
.SS "Single Character Queries"
.LP
Using \fBgl_get_line()\fR to query the user for a single character reply, is
inconvenient for the user, since they must hit the enter or return key before
the character that they typed is returned to the program. Thus the
\fBgl_query_char()\fR function has been provided for single character queries
like this.
.sp
.LP
This function displays the specified prompt at the start of a new line, and
waits for the user to type a character. When the user types a character,
\fBgl_query_char()\fR displays it to the right of the prompt, starts a newline,
then returns the character to the calling program. The return value of the
function is the character that was typed. If the read had to be aborted for
some reason, EOF is returned instead. In the latter case, the application can
call the previously documented \fBgl_return_status()\fR, to find out what went
wrong. This could, for example, have been the reception of a signal, or the
optional inactivity timer going off.
.sp
.LP
If the user simply hits enter, the value of the \fIdefchar\fR argument is
substituted. This means that when the user hits either newline or return, the
character specified in \fIdefchar\fR, is displayed after the prompt, as though
the user had typed it, as well as being returned to the calling application. If
```

such a replacement is not important, simply pass '\en' as the value of
\fIdefchar\fR.
.sp
.LP
If the entered character is an unprintable character, it is displayed
symbolically. For example, control-A is displayed as \fB^A\fR, and characters
beyond 127 are displayed in octal, preceded by a backslash.
.sp
.LP
As with \fBgl_get_line()\fR, echoing of the entered character can be disabled
using the \fBgl_echo_mode()\fR function.
.sp
.LP
If the calling process is suspended while waiting for the user to type their
response, the cursor is moved to the line following the prompt line, then when
the process resumes, the prompt is redisplayed, and \fBgl_query_char()\fR
resumes waiting for the user to type a character.
.sp
.LP
Note that in non-blocking server mode, if an incomplete input line is in the
process of being read when \fBgl_query_char()\fR is called, the partial input
line is discarded, and erased from the terminal, before the new prompt is
displayed. The next call to \fBgl_get_line()\fR will thus start editing a new
line.
.SS "Reading Raw Characters"
.LP
Whereas the \fBgl_query_char()\fR function visibly prompts the user for a
character, and displays what they typed, the \fBgl_read_char()\fR function
reads a signal character from the user, without writing anything to the
terminal, or perturbing any incompletely entered input line. This means that it
can be called not only from between calls to \fBgl_get_line()\fR, but also from
callback functions that the application has registered to be called by
\fBgl_get_line()\fR.
.sp
.LP
On success, the return value of \fBgl_read_char()\fR is the character that was
read. On failure, EOF is returned, and the \fBgl_return_status()\fR function
can be called to find out what went wrong. Possibilities include the optional
inactivity timer going off, the receipt of a signal that is configured to abort
\fBgl_get_line()\fR, or terminal I/O blocking, when in non-blocking server-I/O
mode.
.sp
.LP
Beware that certain keyboard keys, such as function keys, and cursor keys,
usually generate at least three characters each, so a single call to
\fBgl_read_char()\fR will not be enough to identify such keystrokes.
.SS "Clearing The Terminal"
.LP
The calling program can clear the terminal by calling
\fBgl_erase_terminal()\fR. In non-blocking server-I/O mode, this function also
arranges for the current input line to be redrawn from scratch when
\fBgl_get_line()\fR is next called.
.SS "Displaying Text Dynamically"
.LP
Between calls to \fBgl_get_line()\fR, the \fBgl_display_text()\fR function
provides a convenient way to display paragraphs of text, left-justified and
split over one or more terminal lines according to the constraints of the
current width of the terminal. Examples of the use of this function may be
found in the demo programs, where it is used to display introductions. In those
examples the advanced use of optional prefixes, suffixes and filled lines to
draw a box around the text is also illustrated.
.sp
.LP
If \fIgl\fR is not currently connected to a terminal, for example if the output
of a program that uses \fBgl_get_line()\fR is being piped to another program or
redirected to a file, then the value of the \fIdef_width\fR parameter is used

as the terminal width.
.sp
.LP
The \fIindentation\fR argument specifies the number of characters to use to
indent each line of output. The \fIfill_char\fR argument specifies the character
that will be used to perform this indentation.
.sp
.LP
The \fIprefix\fR argument can be either \fINULL\fR or a string to place at the
beginning of each new line (after any indentation). Similarly, the \fIsuffix\fR
argument can be either \fINULL\fR or a string to place at the end of each line.
The suffix is placed flush against the right edge of the terminal, and any
space between its first character and the last word on that line is filled with
the character specified by the \fIfill_char\fR argument. Normally the
fill-character is a space.
.sp
.LP
The \fIstart\fR argument tells \fBgl_display_text()\fR how many characters have
already been written to the current terminal line, and thus tells it the
starting column index of the cursor. Since the return value of
\fBgl_display_text()\fR is the ending column index of the cursor, by passing
the return value of one call to the start argument of the next call, a
paragraph that is broken between more than one string can be composed by
calling \fBgl_display_text()\fR for each successive portion of the paragraph.
Note that literal newline characters are necessary at the end of each paragraph
to force a new line to be started.
.sp
.LP
On error, \fBgl_display_text()\fR returns -1.
.SS "Callback Function Facilities"
.LP
Unless otherwise stated, callback functions such as tab completion callbacks
and event callbacks should not call any functions in this module. The following
functions, however, are designed specifically to be used by callback functions.
.sp
.LP
Calling the \fBgl_replace_prompt()\fR function from a callback tells
\fBgl_get_line()\fR to display a different prompt when the callback returns.
Except in non-blocking server mode, it has no effect if used between calls to
\fBgl_get_line()\fR. In non-blocking server mode, when used between two calls
to \fBgl_get_line()\fR that are operating on the same input line, the current
input line will be re-drawn with the new prompt on the following call to
\fBgl_get_line()\fR.
.SS "International Character Sets"
.LP
Since \fBlibtecla\fR(3LIB) version 1.4.0, \fBgl_get_line()\fR has been 8-bit
clean. This means that all 8-bit characters that are printable in the user's
current locale are now displayed verbatim and included in the returned input
line. Assuming that the calling program correctly contains a call like the
following,
.sp
.in +2
.nf
setlocale(LC_CTYPE, "")
.fi
.in -2

.sp
.LP
then the current locale is determined by the first of the environment variables
\fBLC_CTYPE\fR, \fBLC_ALL\fR, and \fBLANG\fR that is found to contain a valid
locale name. If none of these variables are defined, or the program neglects to
call \fBsetlocale\fR(3C), then the default C locale is used, which is US 7-bit
ASCII. On most UNIX-like platforms, you can get a list of valid locales by
typing the command:
.sp

```
.in +2
.nf
locale -a
.fi
.in -2
.sp

.sp
.LP
at the shell prompt. Further documentation on how the user can make use of this
to enter international characters can be found in the \fBtecla\fR(5) man page.
.SS "Thread Safety"
.LP
Unfortunately neither \fBterminfo\fR nor \fBtermcap\fR were designed to be
reentrant, so you cannot safely use the functions of the getline module in
multiple threads (you can use the separate file-expansion and word-completion
modules in multiple threads, see the corresponding man pages for details).
However due to the use of POSIX reentrant functions for looking up home
directories, it is safe to use this module from a single thread of a
multi-threaded program, provided that your other threads do not use any
\fBtermcap\fR or \fBterminfo\fR functions.
.SH ATTRIBUTES
.LP
See \fBattributes\fR(5) for descriptions of the following attributes:
.sp

.sp
.TS
box;
c | c
l | l .
ATTRIBUTE TYPE  ATTRIBUTE VALUE
_
Interface Stability     Committed
_
MT-Level        MT-Safe
.TE

.SH SEE ALSO
.LP
\fBcpl_complete_word\fR(3TECLA), \fBef_expand_file\fR(3TECLA),
\fBgl_io_mode\fR(3TECLA), \fBlibtecla\fR(3LIB), \fBpca_lookup_file\fR(3TECLA),
\fBattributes\fR(5), \fBtecla\fR(5)
```

```
**********************************************************
   22180 Sat Jan 18 13:36:58 2020
new/usr/src/man/man3tecla/gl_io_mode.3tecla
12212 typos in some section 3tecla man pages
**********************************************************
'\" te
.\" Copyright (c) 2000, 2001, 2002, 2003, 2004 by Martin C. Shepherd.
.\" All Rights Reserved.
.\" Permission is hereby granted, free of charge, to any person obtaining a copy
.\" "Software"), to deal in the Software without restriction, including
.\" without limitation the rights to use, copy, modify, merge, publish,
.\" distribute, and/or sell copies of the Software, and to permit persons
.\" to whom the Software is furnished to do so, provided that the above
.\" copyright notice(s) and this permission notice appear in all copies of
.\" the Software and that both the above copyright notice(s) and this
.\" permission notice appear in supporting documentation.
.\"
.\" THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
.\" OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
.\" MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT
.\" OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
.\" HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL
.\" INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING
.\" FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
.\" NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
.\" WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
.\"
.\" Except as contained in this notice, the name of a copyright holder
.\" shall not be used in advertising or otherwise to promote the sale, use
.\" or other dealings in this Software without prior written authorization
.\" of the copyright holder.
.\" Portions Copyright (c) 2007, Sun Microsystems, Inc. All Rights Reserved.
.TH GL_IO_MODE 3TECLA "January 18, 2020"
.TH GL_IO_MODE 3TECLA "Jun 1, 2004"
.SH NAME
gl_io_mode, gl_raw_io, gl_normal_io, gl_tty_signals, gl_abandon_line,
gl_handle_signal, gl_pending_io \- use \fBgl_get_line()\fR from an external
event loop
.SH SYNOPSIS
.LP
.nf
cc [ \fIflag\fR\&.\|.\|. ] \fIfile\fR\&.\|.\|. \fB-ltecla\fR [ \fIlibrary\fR\&.\
#include <libtecla.h>

\fBint\fR \fBgl_io_mode\fR(\fBGetLine *\fR\fIgl\fR, \fBGlIOMode\fR \fImode\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_raw_io\fR(\fBGetLine *\fR\fIgl\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_normal_io\fR(\fBGetLine *\fR\fIgl\fR);
.fi

.LP
.nf
\fBint\fR \fBgl_tty_signals\fR(\fBvoid (*\fR\fIterm_handler\fR)(int), \fBvoid (*
     \fBvoid (*\fR\fIcont_handler\fR)(int), \fBvoid (*\fR\fIsize_handler\fR)(int
.fi

.LP
.nf
\fBvoid\fR \fBgl_abandon_line\fR(\fBGetLine *\fR\fIgl\fR);
```



```
.fi

.LP
.nf
\fBvoid\fR \fBgl_handle_signal\fR(\fBint\fR \fIsigno\fR, \fBGetLine *\fR\fIgl\fR
.fi

.LP
.nf
\fBGlPendingIO\fR \fBgl_pending_io\fR(\fBGetLine *\fR\fIgl\fR);
.fi

.SH DESCRIPTION
.sp
.LP
The \fBgl_get_line\fR(3TECLA) function supports two different I/O modes. These
are selected by calling the \fBgl_io_mode()\fR function. The \fImode\fR
argument of \fBgl_io_mode()\fR specifies the new I/O mode and must be one of
the following.
.sp
.ne 2
.na
\fB\fBGL_NORMAL_MODE\fR\fR
.ad
.RS 18n
Select the normal blocking-I/O mode. In this mode \fBgl_get_line()\fR does not
return until either an error occurs of the user finishes entering a new line.
.RE

.sp
.ne 2
.na
\fB\fBGL_SERVER_MODE\fR\fR
.ad
.RS 18n
Select non-blocking server I/O mode. In this mode, since non-blocking terminal
I/O is used, the entry of each new input line typically requires many calls to
\fBgl_get_line()\fR from an external I/O-driven event loop.
.RE

.sp
.LP
Newly created GetLine objects start in normal I/O mode, so to switch to
non-blocking server mode requires an initial call to \fBgl_io_mode()\fR.
.SS "Server I/O Mode"
.sp
.LP
In non-blocking server I/O mode, the application is required to have an event
loop that calls \fBgl_get_line()\fR whenever the terminal file descriptor can
perform the type of I/O that \fBgl_get_line()\fR is waiting for. To determine
perform the type I/O that \fBgl_get_line()\fR is waiting for. To determine
which type of I/O \fBgl_get_line()\fR is waiting for, the application calls the
\fBgl_pending_io()\fR function.  The return value is one of the following two
enumerated values.
.sp
.ne 2
.na
\fB\fBGLP_READ\fR\fR
.ad
.RS 13n
\fBgl_get_line()\fR is waiting to write a character to the terminal.
.RE

.sp
.ne 2
.na
```

\fB\fBGLP_WRITE\fR\fR
.ad
.RS 13n
\fBgl_get_line()\fR is waiting to read a character from the keyboard.
\fBgl_get_line()\fR is waiting to read a character from the keyboad.
.RE

.sp
.LP
If the application is using either the \fBselect\fR(3C) or \fBpoll\fR(2)
function to watch for I/O on a group of file descriptors, then it should call
the \fBgl_pending_io()\fR function before each call to these functions to
determine which direction of I/O it should tell them to watch for, and
configure their arguments accordingly. In the case of the \fBselect()\fR
function, this means using the \fBFD_SET()\fR macro to add the terminal file
descriptor either to the set of file descriptors to be watched for readability
or the set to be watched for writability.
.sp
.LP
As in normal I/O mode, the return value of \fBgl_get_line()\fR is either a
pointer to a completed input line or \fINULL\fR. However, whereas in normal I/O
mode a \fINULL\fR return value always means that an error occurred, in
non-blocking server mode, \fINULL\fR is also returned when \fBgl_get_line()\fR
cannot read or write to the terminal without blocking. Thus in non-blocking
server mode, in order to determine when a \fINULL\fR return value signifies
that an error occurred or not, it is necessary to call the
\fBgl_return_status()\fR function. If this function returns the enumerated
value \fBGLR_BLOCKED\fR, \fBgl_get_line()\fR is waiting for I/O and no error
has occurred.
.sp
.LP
When \fBgl_get_line()\fR returns \fINULL\fR and \fBgl_return_status()\fR
indicates that this is due to blocked terminal I/O, the application should call
\fBgl_get_line()\fR again when the type of I/O reported by
\fBgl_pending_io()\fR becomes possible. The \fIprompt\fR, \fIstart_line\fR and
\fIstart_pos\fR arguments of \fBgl_get_line()\fR will be ignored on these
calls. If you need to change the prompt of the line that is currently being
edited, you can call the \fBgl_replace_prompt\fR(3TECLA) function between calls
to \fBgl_get_line()\fR.
.SS "Giving Up The Terminal"
.sp
.LP
A complication that is unique to non-blocking server mode is that it requires
that the terminal be left in raw mode between calls to \fBgl_get_line()\fR. If
this were not the case, the external event loop would not be able to detect
individual key-presses, and the basic line editing implemented by the terminal
driver would clash with the editing provided by \fBgl_get_line()\fR. When the
terminal needs to be used for purposes other than entering a new input line
with \fBgl_get_line()\fR, it needs to be restored to a usable state. In
particular, whenever the process is suspended or terminated, the terminal must
be returned to a normal state. If this is not done, then depending on the
characteristics of the shell that was used to invoke the program, the user
could end up with a hung terminal. To this end, the \fBgl_normal_io()\fR
function is provided for switching the terminal back to the state that it was
in when raw mode was last established.
.sp
.LP
The \fBgl_normal_io()\fR function first flushes any pending output to the
terminal, then moves the cursor to the start of the terminal line which follows
the end of the incompletely entered input line. At this point it is safe to
suspend or terminate the process, and it is safe for the application to read
and write to the terminal. To resume entry of the input line, the application
should call the \fBgl_raw_io()\fR function.
.sp
.LP
The \fBgl_normal_io()\fR function starts a new line, redisplays the partially

completed input line (if any), restores the cursor position within this line to
where it was when \fBgl_normal_io()\fR was called, then switches back to raw,
non-blocking terminal mode ready to continue entry of the input line when
\fBgl_get_line()\fR is next called.
.sp
.LP
Note that in non-blocking server mode, if \fBgl_get_line()\fR is called after a
call to \fBgl_normal_io()\fR, without an intervening call to \fBgl_raw_io()\fR,
\fBgl_get_line()\fR will call \fBgl_raw_mode()\fR itself, and the terminal will
remain in this mode when \fBgl_get_line()\fR returns.
.SS "Signal Handling"
.sp
.LP
In the previous section it was pointed out that in non-blocking server mode,
the terminal must be restored to a sane state whenever a signal is received
that either suspends or terminates the process. In normal I/O mode, this is
done for you by \fBgl_get_line()\fR, but in non-blocking server mode, since the
terminal is left in raw mode between calls to \fBgl_get_line()\fR, this signal
handling has to be done by the application. Since there are many signals that
can suspend or terminate a process, as well as other signals that are important
to \fBgl_get_line()\fR, such as the \fBSIGWINCH\fR signal, which tells it when
the terminal size has changed, the \fBgl_tty_signals()\fR function is provided
for installing signal handlers for all pertinent signals.
.sp
.LP
The \fBgl_tty_signals()\fR function uses \fBgl_get_line()\fR's internal list of
signals to assign specified signal handlers to groups of signals. The arguments
of this function are as follows.
.sp
.ne 2
.na
\fB\fIterm_handler\fR\fR
.ad
.RS 16n
This is the signal handler that is used to trap signals that by default
terminate any process that receives them (for example, \fBSIGINT\fR or
\fBSIGTERM\fR).
.RE

.sp
.ne 2
.na
\fB\fIsusp_handler\fR\fR
.ad
.RS 16n
This is the signal handler that is used to trap signals that by default suspend
any process that receives them, (for example, \fBSIGTSTP\fR or \fBSIGTTOU\fR).
.RE

.sp
.ne 2
.na
\fB\fIcont_handler\fR\fR
.ad
.RS 16n
This is the signal handler that is used to trap signals that are usually sent
when a process resumes after being suspended (usually \fBSIGCONT\fR). Beware
that there is nothing to stop a user from sending one of these signals at other
times.
.RE

.sp
.ne 2
.na
\fB\fIsize_handler\fR\fR
.ad

```
.RS 16n
This signal handler is used to trap signals that are sent to processes when
their controlling terminals are resized by the user (for example,
\fBSIGWINCH\fR).
.RE

.sp
.LP
These arguments can all be the same, if so desired, and \fBSIG_IGN\fR (ignore
this signal) or \fBSIG_DFL\fR (use the system-provided default signal handler)
can be specified instead of a function where pertinent. In particular, it is
rarely useful to trap \fBSIGCONT\fR, so the \fIcont_handler\fR argument will
usually be \fBSIG_DFL\fR or \fBSIG_IGN\fR.
.sp
.LP
The \fBgl_tty_signals()\fR function uses the POSIX \fBsigaction\fR(2) function
to install these signal handlers, and it is careful to use the \fIsa_mask\fR
member of each \fBsigaction\fR structure to ensure that only one of these
signals is ever delivered at a time. This guards against different instances of
these signal handlers from simultaneously trying to write to common global
data, such as a shared \fBsigsetjmp\fR(3C) buffer or a signal-received flag.
The signal handlers installed by this function should call the
\fBgl_handle_signal()\fR.
.sp
.LP
The \fIsigno\fR argument tells this function which signal it is being asked to
respond to, and the \fIgl\fR argument should be a pointer to the first element
of an array of \fIngl\fR \fBGetLine\fR objects. If your application has only
one of these objects, pass its pointer as the \fIgl\fR argument and specify
\fIngl\fR as 1.
.sp
.LP
Depending on the signal that is being handled, this function does different
things.
.SS "Process termination signals"
.sp
.LP
If the signal that was caught is one of those that by default terminates any
process that receives it, then \fBgl_handle_signal()\fR does the following
steps.
.RS +4
.TP
1.
First it blocks the delivery of all signals that can be blocked (ie.
\fBSIGKILL\fR and \fBSIGSTOP\fR cannot be blocked).
.RE
.RS +4
.TP
2.
Next it calls \fBgl_normal_io()\fR for each of the ngl GetLine objects. Note
that this does nothing to any of the GetLine objects that are not currently in
raw mode.
.RE
.RS +4
.TP
3.
Next it sets the signal handler of the signal to its default,
process-termination disposition.
.RE
.RS +4
.TP
4.
Next it re-sends the process the signal that was caught.
.RE
.RS +4
.TP
```



```
5.
Finally it unblocks delivery of this signal, which results in the process
being terminated.
.RE
.SS "Process suspension signals"
.sp
.LP
If the default disposition of the signal is to suspend the process, the same
steps are executed as for process termination signals, except that when the
process is later resumed, \fBgl_handle_signal()\fR continues, and does the
following steps.
.RS +4
.TP
1.
It re-blocks delivery of the signal.
.RE
.RS +4
.TP
2.
It reinstates the signal handler of the signal to the one that was displaced
when its default disposition was substituted.
.RE
.RS +4
.TP
3.
For any of the GetLine objects that were in raw mode when
\fBgl_handle_signal()\fR was called, \fBgl_handle_signal()\fR then calls
\fBgl_raw_io()\fR, to resume entry of the input lines on those terminals.
.RE
.RS +4
.TP
4.
Finally, it restores the signal process mask to how it was when
\fBgl_handle_signal()\fR was called.
.RE
.sp
.LP
Note that the process is suspended or terminated using the original signal that
was caught, rather than using the uncatchable \fBSIGSTOP\fR and \fBSIGKILL\fR
signals. This is important, because when a process is suspended or terminated,
the parent of the process may wish to use the status value returned by the wait
system call to figure out which signal was responsible. In particular, most
shells use this information to print a corresponding message to the terminal.
Users would be rightly confused if when their process received a \fBSIGPIPE\fR
signal, the program responded by sending itself a \fBSIGKILL\fR signal, and the
shell then printed out the provocative statement, "Killed!".
.SS "Interrupting The Event Loop"
.sp
.LP
If a signal is caught and handled when the application's event loop is waiting
in \fBselect()\fR or \fBpoll()\fR, these functions will be aborted with
\fBerrno\fR set to \fBEINTR\fR. When this happens the event loop should call
\fBgl_pending_io()\fR before calling \fBselect()\fR or \fBpoll()\fR again. It
should then arrange for \fBselect()\fR or \fBpoll()\fR to wait for the type of
I/O that \fBgl_pending_io()\fR reports. This is necessary because any signal
handler that calls \fBgl_handle_signal()\fR will frequently change the type of
I/O that \fBgl_get_line()\fR is waiting for.
.sp
.LP
If a signal arrives between the statements that configure the arguments of
\fBselect()\fR or \fBpoll()\fR and the calls to these functions, the signal
will not be seen by these functions, which will then not be aborted. If these
functions are waiting for keyboard input from the user when the signal is
received, and the signal handler arranges to redraw the input line to
accommodate a terminal resize or the resumption of the process. This redisplay
will be delayed until the user presses the next key. Apart from puzzling the
```

user, this clearly is not a serious problem. However there is a way, albeit complicated, to completely avoid this race condition. The following steps illustrate this.

1. Block all of the signals that \fBgl_get_line()\fR catches, by passing the signal set returned by \fBgl_list_signals()\fR to \fBsigprocmask\fR(2).
2. Call \fBgl_pending_io()\fR and set up the arguments of \fBselect()\fR or \fBpoll()\fR accordingly.
3. Call \fBsigsetjmp\fR(3C) with a non-zero \fIsavemask\fR argument.
4. Initially this \fBsigsetjmp()\fR statement will return zero, indicating that control is not resuming there after a matching call to \fBsiglongjmp\fR(3C).
5. Replace all of the handlers of the signals that \fBgl_get_line()\fR is configured to catch, with a signal handler that first records the number of the signal that was caught, in a file-scope variable, then calls \fBsiglongjmp()\fR with a non-zero \fIval\fR argument, to return execution to the above \fBsigsetjmp()\fR statement. Registering these signal handlers can conveniently be done using the \fBgl_tty_signals()\fR function.
6. Set the file-scope variable that the above signal handler uses to record any signal that is caught to -1, so that we can check whether a signal was caught by seeing if it contains a valid signal number.
7. Now unblock the signals that were blocked in step 1. Any signal that was received by the process in between step 1 and now will now be delivered, and trigger our signal handler, as will any signal that is received until we block these signals again.
8. Now call \fBselect()\fR or \fBpoll()\fR.
9. When select returns, again block the signals that were unblocked in step 7.

.sp
If a signal is arrived any time during the above steps, our signal handler will be triggered and cause control to return to the \fBsigsetjmp()\fR statement, where this time, \fBsigsetjmp()\fR will return non-zero, indicating that a signal was caught. When this happens we simply skip the above block of statements, and continue with the following statements, which are executed

regardless of whether or not a signal is caught. Note that when \fBsigsetjmp()\fR returns, regardless of why it returned, the process signal mask is returned to how it was when \fBsigsetjmp()\fR was called. Thus the following statements are always executed with all of our signals blocked.

10. Reinstate the signal handlers that were displaced in step 5.
11. **Check whether a signal was caught, by checking the file-scope variable that**
Check wether a signal was caught, by checking the file-scope variable that
the signal handler records signal numbers in.
12. If a signal was caught, send this signal to the application again and unblock only this signal so that it invokes the signal handler which was just reinstated in step 10.
13. Unblock all of the signals that were blocked in step 7.

.SS "Signals Caught By \fBgl_get_line()\fR"
.sp
.LP
Since the application is expected to handle signals in non-blocking server mode, \fBgl_get_line()\fR does not attempt to duplicate this when it is being called. If one of the signals that it is configured to catch is sent to the application while \fBgl_get_line()\fR is being called, \fBgl_get_line()\fR reinstates the caller's signal handlers, then immediately before returning, re-sends the signal to the process to let the application's signal handler handle it. If the process is not terminated by this signal, \fBgl_get_line()\fR returns \fINULL\fR, and a following call to \fBgl_return_status()\fR returns the enumerated value \fBGLR_SIGNAL\fR.

.SS "Aborting Line Input"
.sp
.LP
Often, rather than letting it terminate the process, applications respond to the \fBSIGINT\fR user-interrupt signal by aborting the current input line. This can be accomplished in non-blocking server-I/O mode by not calling \fBgl_handle_signal()\fR when this signal is caught, but by calling instead the \fBgl_abandon_line()\fR function. This function arranges that when \fBgl_get_line()\fR is next called, it first flushes any pending output to the
terminal, discards the current input line, outputs a new prompt on the next
terminal, discardes the current input line, outputs a new prompt on the next
line, and finally starts accepting input of a new input line from the user.

.SS "Signal Safe Functions"
.sp
.LP
Provided that certain rules are followed, the \fBgl_normal_io()\fR, \fBgl_raw_io()\fR, \fBgl_handle_signal()\fR, and \fBgl_abandon_line()\fR functions can be written to be safely callable from signal handlers. Other functions in this library should not be called from signal handlers. For this to be true, all signal handlers that call these functions must be registered in such a way that only one instance of any one of them can be running at one time. The way to do this is to use the POSIX \fBsigaction()\fR function to register all signal handlers, and when doing this, use the \fIsa_mask\fR member of the corresponding \fBsigaction\fR structure to indicate that all of the signals whose handlers invoke the above functions should be blocked when the

```
current signal is being handled. This prevents two signal handlers from
operating on a \fBGetLine\fR object at the same time.
.sp
.LP
To prevent signal handlers from accessing a \fBGetLine\fR object while
\fBgl_get_line()\fR or any of its associated public functions are operating on
it, all public functions associated with \fBgl_get_line()\fR, including
\fBgl_get_line()\fR itself, temporarily block the delivery of signals when they
are accessing \fBGetLine\fR objects. Beware that the only signals that they
block are the signals that \fBgl_get_line()\fR is currently configured to
catch, so be sure that if you call any of the above functions from signal
handlers, that the signals that these handlers are assigned to are configured
to be caught by \fBgl_get_line()\fR. See \fBgl_trap_signal\fR(3TECLA).
.SS "Using Timeouts To Poll"
.sp
.LP
If instead of using \fBselect()\fR or \fBpoll()\fR to wait for I/O your
application needs only to get out of \fBgl_get_line()\fR periodically to
briefly do something else before returning to accept input from the user, use
the \fBgl_inactivity_timeout\fR(3TECLA) function in non-blocking server mode to
specify that a callback function that returns \fBGLTO_CONTINUE\fR should be
called whenever \fBgl_get_line()\fR has been waiting for I/O for more than a
specified amount of time. When this callback is triggered, \fBgl_get_line()\fR
will return \fINULL\fR and a following call to \fBgl_return_status()\fR will
return \fBGLR_BLOCKED\fR.
.sp
.LP
The \fBgl_get_line()\fR function will not return until the user has not typed a
key for the specified interval, so if the interval is long and the user keeps
typing, \fBgl_get_line()\fR might not return for a while. There is no guarantee
that it will return in the time specified.
.SH ATTRIBUTES
.sp
.LP
See \fBattributes\fR(5) for descriptions of the following attributes:
.sp

.sp
.TS
box;
c | c
l | l .
ATTRIBUTE TYPE  ATTRIBUTE VALUE
_
Interface Stability     Evolving
_
MT-Level        MT-Safe
.TE

.SH SEE ALSO
.sp
.LP
\fBcpl_complete_word\fR(3TECLA), \fBef_expand_file\fR(3TECLA),
\fBgl_get_line\fR(3TECLA), \fBlibtecla\fR(3LIB), \fBpca_lookup_file\fR(3TECLA),
\fBattributes\fR(5), \fBtecla\fR(5)
```

```
**********************************************************
   13780 Sat Jan 18 13:36:58 2020
new/usr/src/man/man3tecla/pca_lookup_file.3tecla
12212 typos in some section 3tecla man pages
**********************************************************
'\" te
.\" Copyright (c) 2000, 2001, 2002, 2003, 2004 by Martin C. Shepherd. All Rights
.\" Permission is hereby granted, free of charge, to any person obtaining a copy
.\" "Software"), to deal in the Software without restriction, including
.\" without limitation the rights to use, copy, modify, merge, publish,
.\" distribute, and/or sell copies of the Software, and to permit persons
.\" to whom the Software is furnished to do so, provided that the above
.\" copyright notice(s) and this permission notice appear in all copies of
.\" the Software and that both the above copyright notice(s) and this
.\" permission notice appear in supporting documentation.
.\"
.\" THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
.\" OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
.\" MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT
.\" OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
.\" HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL
.\" INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING
.\" FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
.\" NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
.\" WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
.\"
.\" Except as contained in this notice, the name of a copyright holder
.\" shall not be used in advertising or otherwise to promote the sale, use
.\" or other dealings in this Software without prior written authorization
.\" of the copyright holder.
.\" Portions Copyright (c) 2007, Sun Microsystems, Inc. All Rights Reserved.
.TH PCA_LOOKUP_FILE 3TECLA "January 18, 2020"
.TH PCA_LOOKUP_FILE 3TECLA "Aug 13, 2007"
.SH NAME
pca_lookup_file, del_PathCache, del_PcaPathConf, new_PathCache,
new_PcaPathConf, pca_last_error, pca_path_completions, pca_scan_path,
pca_set_check_fn, ppc_file_start, ppc_literal_escapes \- lookup a file in a
list of directories
.SH SYNOPSIS
.LP
.nf
cc [ \fIflag\fR\&.\|.\|. ] \fIfile\fR\&.\|.\|. \fB-ltecla\fR [ \fIlibrary\fR\&.\
#include <libtecla.h>

\fBchar *\fR\fBpca_lookup_file\fR(\fBPathCache *\fR\fIpc\fR, \fBconst char *\fR\
     \fBint\fR \fIname_len\fR, \fBint\fR \fIliteral\fR);
.fi

.LP
.nf
\fBPathCache *\fR\fBdel_PathCache\fR(\fBPathCache *\fR\fIpc\fR);
.fi

.LP
.nf
\fBPcaPathConf *\fR\fBdel_PcaPathConf\fR(\fBPcaPathConf *\fR\fIppc\fR);
.fi

.LP
.nf
\fBPathCache *\fR\fBnew_PathCache\fR(\fBvoid\fR);
.fi

.LP
.nf
\fBPcaPathConf *\fR\fBnew_PcaPathConf\fR(\fBPathCache *\fR\fIpc\fR);
```



```
.fi

.LP
.nf
\fBconst char *\fR\fBpca_last_error\fR(\fBPathCache *\fR\fIpc\fR);
.fi

.LP
.nf
\fBCPL_MATCH_FN\fR(\fBpca_path_completions\fR);
.fi

.LP
.nf
\fBint\fR \fBpca_scan_path\fR(\fBPathCache *\fR\fIpc\fR, \fBconst char *\fR\fIpa
.fi

.LP
.nf
\fBvoid\fR \fBpca_set_check_fn\fR(\fBPathCache *\fR\fIpc\fR, \fBCplCheckFn *\fR\
     \fBvoid *\fR\fIdata\fR);
.fi

.LP
.nf
\fBvoid\fR \fBppc_file_start\fR(\fBPcaPathConf *\fR\fIppc\fR, \fBint\fR \fIstart
.fi

.LP
.nf
\fBvoid\fR \fBppc_literal_escapes\fR(\fBPcaPathConf *\fR\fIppc\fR, \fBint\fR \fI
.fi

.SH DESCRIPTION
.sp
.LP
The \fBPathCache\fR object is part of the \fBlibtecla\fR(3LIB) library.
\fBPathCache\fR objects allow an application to search for files in any colon
separated list of directories, such as the UNIX execution \fBPATH\fR
environment variable. Files in absolute directories are cached in a
\fBPathCache\fR object, whereas relative directories are scanned as needed.
Using a \fBPathCache\fR object, you can look up the full pathname of a simple
filename, or you can obtain a list of the possible completions of a given
filename prefix. By default all files in the list of directories are targets
for lookup and completion, but a versatile mechanism is provided for only
selecting specific types of files. The obvious application of this facility is
to provide Tab-completion and lookup of executable commands in the UNIX
\fBPATH\fR, so an optional callback which rejects all but executable files is
\fBPATH\fR, so an optional callback which rejects all but executable files, is
provided.
.SS "An Example"
.sp
.LP
Under UNIX, the following example program looks up and displays the full
pathnames of each of the command names on the command line.
.sp
.in +2
.nf
#include <stdio.h>
#include <stdlib.h>
#include <libtecla.h>

int main(int argc, char *argv[])
{
        int i;
        /*
```

```
        * Create a cache for executable files.
        */
        PathCache *pc = new_PathCache();
        if(!pc)
          exit(1);
        /*
        * Scan the user's PATH for executables.
        */
        if(pca_scan_path(pc, getenv("PATH"))) {
          fprintf(stderr, "%s\en", pca_last_error(pc));
          exit(1);
        }
        /*
        * Arrange to only report executable files.
        */
        pca_set_check_fn(pc, cpl_check_exe, NULL);
        /*
        * Lookup and display the full pathname of each of the
        * commands listed on the command line.
        */
        for(i=1; i<argc; i++) {
          char *cmd = pca_lookup_file(pc, argv[i], -1, 0);
          printf("The full pathname of '%s' is %s\e\en", argv[i],
                 cmd ? cmd : "unknown");
        }
        pc = del_PathCache(pc);  /* Clean up */
        return 0;
}
.fi
.in -2

.sp
.LP
The following is an example of what this does on a laptop under LINUX:
.sp
.in +2
.nf
$ ./example less more blob
The full pathname of 'less' is /usr/bin/less
The full pathname of 'more' is /bin/more
The full pathname of 'blob' is unknown
$
.fi
.in -2

.SS "Function Descriptions"
.sp
.LP
To use the facilities of this module, you must first allocate a \fBPathCache\fR
object by calling the \fBnew_PathCache()\fR constructor function. This function
creates the resources needed to cache and lookup files in a list of
directories. It returns \fINULL\fR on error.
.SS "Populating The Cache"
.sp
.LP
Once you have created a cache, it needs to be populated with files. To do this,
call the \fBpca_scan_path()\fR function. Whenever this function is called, it
discards the current contents of the cache, then scans the list of directories
specified in its path argument for files. The path argument must be a string
containing a colon-separated list of directories, such as
"\fB/usr/bin\fR:\fB/home/mcs/bin\fR:". This can include directories specified
by absolute pathnames such as "\fB/usr/bin\fR", as well as sub-directories
specified by relative pathnames such as "." or "\fBbin\fR". Files in the
absolute directories are immediately cached in the specified \fBPathCache\fR
object, whereas subdirectories, whose identities obviously change whenever the
current working directory is changed, are marked to be scanned on the fly
```



```
whenever a file is looked up.
.sp
.LP
On success this function return 0. On error it returns 1, and a description of
the error can be obtained by calling \fBpca_last_error\fR(\fIpc\fR).
.SS "Looking Up Files"
.sp
.LP
Once the cache has been populated with files, you can look up the full pathname
of a file, simply by specifying its filename to \fBpca_lookup_file()\fR.
.sp
.LP
To make it possible to pass this function a filename which is actually part of
a longer string, the \fIname_len\fR argument can be used to specify the length
of the filename at the start of the \fIname\fR[] argument. If you pass -1 for
this length, the length of the string will be determined with \fIstrlen\fR. If
the \fIname\fR[] string might contain backslashes that escape the special
meanings of spaces and tabs within the filename, give the \fIliteral\fR
argument the value 0. Otherwise, if backslashes should be treated as normal
characters, pass 1 for the value of the \fIliteral\fR argument.
.SS "Filename Completion"
.sp
.LP
Looking up the potential completions of a filename-prefix in the filename cache
is achieved by passing the provided \fBpca_path_completions()\fR callback
function to the \fBcpl_complete_word\fR(3TECLA) function.
.sp
.LP
This callback requires that its data argument be a pointer to a PcaPathConf
object. Configuration objects of this type are allocated by calling
\fBnew_PcaPathConf()\fR.
.sp
.LP
This function returns an object initialized with default configuration
parameters, which determine how the \fBcpl_path_completions()\fR callback
function behaves. The functions which allow you to individually change these
parameters are discussed below.
.sp
.LP
By default, the \fBpca_path_completions()\fR callback function searches
backwards for the start of the filename being completed, looking for the first
un-escaped space or the start of the input line. If you wish to specify a
different location, call \fBppc_file_start()\fR with the index at which the
filename starts in the input line. Passing \fIstart_index\fR=-1 re-enables the
default behavior.
.sp
.LP
By default, when \fBpca_path_completions()\fR looks at a filename in the input
line, each lone backslash in the input line is interpreted as being a special
character which removes any special significance of the character which follows
it, such as a space which should be taken as part of the filename rather than
delimiting the start of the filename. These backslashes are thus ignored while
looking for completions, and subsequently added before spaces, tabs and literal
backslashes in the list of completions. To have unescaped backslashes treated
as normal characters, call \fBppc_literal_escapes()\fR with a non-zero value in
its literal argument.
.sp
.LP
When you have finished with a \fBPcaPathConf\fR variable, you can pass it to
the \fBdel_PcaPathConf()\fR destructor function to reclaim its memory.
.SS "Being Selective"
.sp
.LP
If you are only interested in certain types or files, such as, for example,
executable files, or files whose names end in a particular suffix, you can
arrange for the file completion and lookup functions to be selective in the
```

```
filenames that they return. This is done by registering a callback function
with your \fBPathCache\fR object. Thereafter, whenever a filename is found
which either matches a filename being looked up or matches a prefix which is
being completed, your callback function will be called with the full pathname
of the file, plus any application-specific data that you provide. If the
callback returns 1 the filename will be reported as a match. If it returns 0,
it will be ignored. Suitable callback functions and their prototypes should be
declared with the following macro. The \fBCplCheckFn\fR typedef is also
provided in case you wish to declare pointers to such functions.
provided in case you wish to declare pointers to such functions
.sp
.in +2
.nf
#define CPL_CHECK_FN(fn) int (fn)(void *data, const char *pathname)
typedef CPL_CHECK_FN(CplCheckFn);
.fi
.in -2

.sp
.LP
Registering one of these functions involves calling the
\fBpca_set_check_fn()\fR function. In addition to the callback function passed
with the \fIcheck_fn\fR argument, you can pass a pointer to anything with the
\fIdata\fR argument. This pointer will be passed on to your callback function
by its own \fIdata\fR argument whenever it is called, providing a way to pass
application-specific data to your callback. Note that these callbacks are
passed the full pathname of each matching file, so the decision about whether a
file is of interest can be based on any property of the file, not just its
filename. As an example, the provided \fBcpl_check_exe()\fR callback function
looks at the executable permissions of the file and the permissions of its
parent directories, and only returns 1 if the user has execute permission to
the file. This callback function can thus be used to lookup or complete command
names found in the directories listed in the user's \fBPATH\fR environment
variable. The example program above provides a demonstration of this.
.sp
.LP
Beware that if somebody tries to complete an empty string, your callback will
get called once for every file in the cache, which could number in the
thousands. If your callback does anything time consuming, this could result in
an unacceptable delay for the user, so callbacks should be kept short.
.sp
.LP
To improve performance, whenever one of these callbacks is called, the choice
that it makes is cached, and the next time the corresponding file is looked up,
instead of calling the callback again, the cached record of whether it was
accepted or rejected is used. Thus if somebody tries to complete an empty
string, and hits tab a second time when nothing appears to happen, there will
only be one long delay, since the second pass will operate entirely from the
cached dispositions of the files. These cached dispositions are discarded
cached dispositions of the files. These cached dipositions are discarded
whenever \fBpca_scan_path()\fR is called, and whenever \fBpca_set_check_fn()\fR
is called with changed callback function or \fIdata\fR arguments.
.SS "Error Handling"
.sp
.LP
If \fBpca_scan_path()\fR reports that an error occurred by returning 1, you can
obtain a terse description of the error by calling
\fBpca_last_error\fR(\fIpc\fR). This returns an internal string containing an
error message.
.SS "Cleaning Up"
.sp
.LP
Once you have finished using a \fBPathCache\fR object, you can reclaim its
resources by passing it to the \fBdel_PathCache()\fR destructor function. This
takes a pointer to one of these objects, and always returns \fINULL\fR.
.SS "Thread Safety"
```



```
.sp
.LP
It is safe to use the facilities of this module in multiple threads, provided
that each thread uses a separately allocated \fBPathCache\fR object. In other
words, if two threads want to do path searching, they should each call
\fBnew_PathCache()\fR to allocate their own caches.
.SH ATTRIBUTES
.sp
.LP
See \fBattributes\fR(5) for descriptions of the following attributes:
.sp

.sp
.TS
box;
c | c
l | l .
ATTRIBUTE TYPE  ATTRIBUTE VALUE
_
Interface Stability     Evolving
_
MT-Level        MT-Safe
.TE

.SH SEE ALSO
.sp
.LP
\fBcpl_complete_word\fR(3TECLA), \fBef_expand_file\fR(3TECLA),
\fBgl_get_line\fR(3TECLA), \fBlibtecla\fR(3LIB), \fBattributes\fR(5)
```