GNU History Library

Brian Fox
Free Software Foundation
Version 1.1
April 1991

This document describes the GNU History library, a programming tool that provides a consistent user interface for recalling lines of previously typed input.

Published by the Free Software Foundation
675 Massachusetts Avenue,
Cambridge, MA 02139 USA

1 Using History Interactively

This chapter describes how to use the GNU History Library interactively, from a user's standpoint. It should be considered a user's guide. For information on using the GNU History Library in your own programs, see Chapter 2 [Programming with GNU History], page 3.

1.1 History Interaction

The History library provides a history expansion feature that is similar to the history expansion in Csh. The following text describes the sytax that you use to manipulate the history information.

History expansion takes place in two parts. The first is to determine which line from the previous history should be used during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the previous history is called the *event*, and the portions of that line that are acted upon are called *words*. The line is broken into words in the same fashion that the Bash shell does, so that several English (or Unix) words surrounded by quotes are considered as one word.

1.1.1 Event Designators

An event designator is a reference to a command line entry in the history list.

`!` Start a history subsitittution, except when followed by a space, tab, or the end of the line... = or (.

`!!` Refer to the previous command. This is a synonym for `!-1`.

`!n` Refer to command line *n*.

`!-n` Refer to the command line *n* lines back.

`!string` Refer to the most recent command starting with *string*.

`!?string`[`?`]
Refer to the most recent command containing *string*.

1.1.2 Word Designators

A : separates the event specification from the word designator. It can be omitted if the word designator begins with a ^, $, * or %. Words are numbered from the beginning of the line, with the first word being denoted by a 0 (zero).

`0 (zero)` The zero'th word. For many applications, this is the command word.

`n` The *n*'th word.

`^` The first argument. that is, word 1.

`$` The last argument.

`%` The word matched by the most recent `?string?` search.

`x-y` A range of words; `-`*y* Abbreviates `0-`*y*.

`*` All of the words, excepting the zero'th. This is a synonym for `1-$`. It is not an error to use `*` if there is just one word in the event. The empty string is returned in that case.

1.1.3 Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a `:`.

`#` The entire command line typed so far. This means the current command, not the previous command, so it really isn't a word designator, and doesn't belong in this section.

`h` Remove a trailing pathname component, leaving only the head.

`r` Remove a trailing suffix of the form '`.`'*suffix*, leaving the basename.

`e` Remove all but the suffix.

`t` Remove all leading pathname components, leaving the tail.

`p` Print the new command but do not execute it.

2 Programming with GNU History

This chapter describes how to interface the GNU History Library with programs that you write. It should be considered a technical guide. For information on the interactive use of GNU History, see Chapter 1 [Using History Interactively], page 1.

2.1 Introduction to History

Many programs read input from the user a line at a time. The GNU history library is able to keep track of those lines, associate arbitrary data with each line, and utilize information from previous lines in making up new ones.

The programmer using the History library has available to him functions for remembering lines on a history stack, associating arbitrary data with a line, removing lines from the stack, searching through the stack for a line containing an arbitrary text string, and referencing any line on the stack directly. In addition, a history *expansion* function is available which provides for a consistent user interface across many different programs.

The end-user using programs written with the History library has the benifit of a consistent user interface, with a set of well-known commands for manipulating the text of previous lines and using that text in new commands. The basic history manipulation commands are similar to the history substitution used by `Csh`.

If the programmer desires, he can use the Readline library, which includes some history manipulation by default, and has the added advantage of Emacs style command line editing.

2.2 History Storage

```
typedef struct _hist_entry {
  char *line;
  char *data;
} HIST_ENTRY;
```

2.3 History Functions

This section describes the calling sequence for the various functions present in GNU History.

void using_history () Function

Begin a session in which the history functions might be used. This just initializes the interactive variables.

void add_history (*char *string*) Function

Place *string* at the end of the history list. The associated data field (if any) is set to `NULL`.

int where_history () Function

Returns the number which says what history element we are now looking at.

int history_set_pos (*int pos*) Function

Set the position in the history list to *pos*.

int history_search_pos (*char *string, int direction, int pos*) Function

Search for *string* in the history list, starting at *pos*, an absolute index into the list. *direction*, if negative, says to search backwards from *pos*, else forwards. Returns the absolute index of the history element where *string* was found, or -1 otherwise.

HIST_ENTRY *remove_history (); Function

Remove history element *which* from the history. The removed element is returned to you so you can free the line, data, and containing structure.

void stifle_history (*int max*) Function

Stifle the history list, remembering only *max* number of entries.

int unstifle_history (); Function

Stop stifling the history. This returns the previous amount the history was stifled by. The value is positive if the history was stifled, negative if it wasn't.

int read_history (*char *filename*) Function

Add the contents of *filename* to the history list, a line at a time. If *filename* is `NULL`, then read from '`~/.history`'. Returns 0 if successful, or errno if not.

int read_history_range (*char *filename, int from, int to*) Function

Read a range of lines from *filename*, adding them to the history list. Start reading at the *from*'th line and end at the *to*'th. If *from* is zero, start at the beginning. If *to* is less than *from*, then read until the end of the file. If *filename* is `NULL`, then read from '`~/.history`'. Returns 0 if successful, or `errno` if not.

int write_history (*char *filename*) Function

Append the current history to *filename*. If *filename* is `NULL`, then append the history list to '`~/.history`'. Values returned are as in `read_history ()`.

int append_history (*int nelements, char *filename*) Function

Append *nelement* entries to *filename*. The entries appended are from the end of the list minus *nelements* up to the end of the list.

HIST_ENTRY *replace_history_entry () Function

Make the history entry at *which* have *line* and *data*. This returns the old entry so you can dispose of the data. In the case of an invalid *which*, a `NULL` pointer is returned.

HIST_ENTRY *current_history () Function

Return the history entry at the current position, as determined by `history_offset`. If there is no entry there, return a `NULL` pointer.

HIST_ENTRY *previous_history () Function

Back up *history_offset* to the previous history entry, and return a pointer to that entry. If there is no previous entry, return a `NULL` pointer.

HIST_ENTRY *next_history () Function

Move `history_offset` forward to the next history entry, and return the a pointer to that entry. If there is no next entry, return a `NULL` pointer.

HIST_ENTRY **history_list () Function

Return a `NULL` terminated array of `HIST_ENTRY` which is the current input history. Element 0 of this list is the beginning of time. If there is no history, return `NULL`.

int history_search (*char *string, int direction*) Function

Search the history for *string*, starting at `history_offset`. If *direction* `<` 0, then the search is through previous entries, else through subsequent. If *string* is found, then `current_history ()` is the history entry, and the value of this function is the offset in the line of that history entry that the *string* was found in. Otherwise, nothing is changed, and a -1 is returned.

int history_expand (*char *string, char **output*) Function

Expand *string*, placing the result into *output*, a pointer to a string. Returns:

`0` If no expansions took place (or, if the only change in the text was the de-slashifying of the history expansion character),

`1` if expansions did take place, or

`-1` if there was an error in expansion.

If an error ocurred in expansion, then *output* contains a descriptive error message.

char *history_arg_extract (*int first, int last, char *string*) Function

Extract a string segment consisting of the *first* through *last* arguments present in *string*. Arguments are broken up as in the GNU Bash shell.

int history_total_bytes (); Function

Return the number of bytes that the primary history entries are using. This just adds up the lengths of `the_history->lines`.

2.4 History Variables

This section describes the variables in GNU History that are externally visible.

int history_base Variable

For convenience only. You set this when interpreting history commands. It is the logical offset of the first history element.

2.5 History Programming Example

The following snippet of code demonstrates simple use of the GNU History Library.

```
main ()
{
  char line[1024], *t;
  int done = 0;

  line[0] = 0;

  while (!done)
    {
      fprintf (stdout, "history%% ");
      t = gets (line);

      if (!t)
        strcpy (line, "quit");

      if (line[0])
        {
          char *expansion;
          int result;

          using_history ();

          result = history_expand (line, &expansion);
          strcpy (line, expansion);
          free (expansion);
          if (result)
            fprintf (stderr, "%s\n", line);

          if (result < 0)
            continue;

          add_history (line);
        }

      if (strcmp (line, "quit") == 0) done = 1;
      if (strcmp (line, "save") == 0) write_history (0);
      if (strcmp (line, "read") == 0) read_history (0);
      if (strcmp (line, "list") == 0)
        {
          register HIST_ENTRY **the_list = history_list ();
          register int i;

          if (the_list)
            for (i = 0; the_list[i]; i++)
              fprintf (stdout, "%d: %s\n",
```

```
                  i + history_base, the_list[i]->line);
          }
        if (strncmp (line, "delete", strlen ("delete")) == 0)
          {
            int which;
            if ((sscanf (line + strlen ("delete"), "%d", &which)) == 1)
              {
                HIST_ENTRY *entry = remove_history (which);
                if (!entry)
                  fprintf (stderr, "No such entry %d\n", which);
                else
                  {
                    free (entry->line);
                    free (entry);
                  }
              }
            else
              {
                fprintf (stderr, "non-numeric arg given to `delete'\n");
              }
          }
      }
  }
```

Appendix A Concept Index

Appendix B Function and Variable Index

(Index is nonexistent)

Table of Contents