New Tricks for an Old Terminal Driver

Eric Fischer
The University of Chicago

New Tricks for an Old Terminal Driver

Eric Fischer
The University of Chicago
`enf@pobox.com`

Abstract

Users expect more out of command lines than they did fifteen years ago, but the terminal interface has not evolved to keep up with their expectations. With a few modifications, though, the terminal driver can provide every program with support for the arrow keys and several common Emacs commands.

Introduction

After some significant improvements in 4BSD and System III in the early 1980s, the Unix terminal interface stopped evolving. Since then, even substantial rewrites like 4.4BSD and completely new implementations like Linux have given terminals the same old set of features without adding anything new.

Meanwhile, users have continued to demand better and better interfaces. With stagnation in the operating system, the responsibility for improvement has fallen to individual programs. The result is that some programs (the ones that do all the work themselves) have very good interfaces, while others (the ones that depend upon operating system services) have very bad ones.

It would be very difficult to make the kernel include all the editing features from Emacs or *vi*. These editors are large and complicated programs, and a reasonably complete imitation of either of them must also be large and complicated. Attempting to make the same kernel code work on different versions of Unix makes the complexity even worse.

But a complete clone of Emacs is more than most programs really need. As Kernighan and Plauger put it, ‘‘most users of a tool are willing to meet you halfway; if you do ninety percent of the job, they will be ecstatic.’’ The standard kernel provides about fifty percent of what user testing shows that people want out of a command line editor. A few select enhancements will raise this figure to more than ninety nine percent.

Most importantly, it turns out, people want to be able to delete things: the previous character, the next character, the previous word, the whole line, and to the end of the line. They want to be able to move up and down a history list. They also want to be able to move left and right within the line they are editing, a character or a word at a time, and to the start or end of the line. Finally, they want to be able to complete partially-typed filenames and to clear the screen.

Other, more elaborate features are occasionally useful, but most users will never notice that anything is missing if they can do the things listed above. Selecting this small set of features to implement also means that portability need not be a major concern, since the ideas can easily be applied to a different kernel implementation even if the code itself cannot be.

Implementation

As you type characters, the terminal driver places them into a structure called (on BSD systems) the raw queue. This structure was not designed for elaborate editing, so there is normally no way to add characters to or remove characters from it other than at the end.

One way to work around this limitation without introducing an entirely different representation is to use two queues for the current input line. The raw queue still holds the characters are to the left of the cursor, and a new queue, the editing queue, holds the ones to the right of it, if there are any. As the cursor is moved left, characters are removed from the raw queue and placed into the editing queue. When moving the other direction, the opposite happens.

So, for example, if someone had typed the line

```
sort file | uniq -c | sort -rn
```

and moved the cursor back over the `i` as shown, the raw queue would contain

```
sort file | un
```

and the contents of the editing queue would be

```
nr- tros | c- qi
```

The characters in the editing queue are in reverse order because it is really being used as a stack, not a queue. When the characters are moved back, one at a time, into the raw queue, they will again be in the correct order.

Since the characters after the cursor are kept in their own queue, inserting and deleting characters before the cursor can be done in exactly the same way as with the standard driver. As a result, most of the code never needs to know that the editing queue exists at all. Most of the parts that do know about it only use the two functions that move the cursor left and right.

The functions that move the cursor left and right update the structures in memory and the image on the terminal at the same time. The only control character that the updating routine uses is Backspace, so it should work with nearly any terminal.

User interface

Once this infrastructure is in place, the next task is to make a user interface for it. Other programs have already established a de facto standard for what this interface should look like: people expect to be able to use the arrow keys and some of the control characters from Emacs. In practice, even diehard *vi* fans generally seem to be willing to put up with Emacs-style command line editing.

This is lucky, because a minimal version of Emacs is much easier to write than a minimal version of *vi*. A *vi* editing mode would be a good thing to add eventually, but I have given up on it for now because *vi*'s compound command structure and the awkward access to data in BSD queues made it too hard to do a good job.

Most basic Emacs commands, on the other hand, are very straightforward to implement. Control-B and Control-F, which move the cursor left and right, respectively, simply call the primitive function that performs this task. Control-A and Control-E, which move to the start and end of the line, do the same, but keep calling the function until it fails, which means that the end of the line has been reached.

The deletion commands are only slightly more complicated. Control-K, which deletes up to the end of the line, first checks to see how many characters there are after the cursor, moves forward that many characters, and then deletes backward the same number of times. Control-D, which deletes a single character, is cursed by being the same character that Unix uses for end-of-file. If there are any characters after the cursor, it deletes one; otherwise, it falls through to the normal input processing which interprets it as end-of-file.

These Emacs commands, incidentally, are only interpreted when the `L_EMACS` bit is set in the terminal control flags and the terminal is in canonical mode. This allows anyone who only wants to use the standard terminal facilities to disable the Emacs commands with `stty -emacs`. There is room for an `L_VI` flag for when a more ambitious version adds support for the *vi* command set.

Compounds and arrows

Other Emacs commands and ANSI-standard arrow keys use multicharacter sequences beginning with `ESC`. So when the terminal driver is in Emacs mode and receives an `ESC` character, it sets a flag to record this and then returns to wait for the next character to arrive. This is similar to the way the literal-next character is already handled.

If the next character that arrives is also `ESC`, it falls through into the normal processing, so a *csh*-style filename completion facility can still be used by typing `ESC ESC`. If the character is `b` or `f`, then the sequence is the Emacs backward- or forward-word command and the cursor is moved backward or forward until a word boundary or the end of a line is reached.

If the character after `ESC` is `[` or `O`, then it is assumed to be part of the sequence for a VT100-style arrow key, and another bit is set to indicate this. When the following character arrives, if it is `A`, `B`, `C`, or `D`, then the appropriate arrow key action is taken.

It is a shame to hardwire these sequences into the program, but they are used by nearly every terminal, they are specified by an ANSI standard, and the code to support them is much less complicated than a more configurable version would be. The same features are still available on non-ANSI terminals by using equivalent Emacs commands rather than arrows.

History

Since the function of the up and down arrows (as well as Control-P and Control-N) is to move through the history list, this is an appropriate point at which to introduce the history mechanism. In an earlier (1997) implementation, I put all the code to manage the history list directly in the terminal driver, but the experience convinced me that history is too complicated and takes too much memory to make it reasonable to put entirely inside the kernel.

In the current version, the real work of maintaining the history list is done by a daemon, *ttyd*, which runs as a user process and makes *ioctl* calls to listen for instructions from the terminal driver. The driver can post requests for the previous or next item from a terminal's history list or to add a new line to the list. The daemon satisfies these by using additional *ioctl*s to query and set the contents of a terminal's current input buffer.

Each process on each terminal has a separate history list so programs do not interfere with each other. Typed input is added to the history list only when the terminal is in canonical mode and only when echoing is turned on, so there should be no risk of passwords ending up in the history by mistake.

Like the Emacs features, the history list can be enabled or disabled for a particular terminal by setting or resetting the `L_HISTORY` bit using *stty*.

The new *ioctl*s that were added to support the history features also turn out to have more general uses.

The header-editing feature in Berkeley *mail*, for instance, stuffs each header into the editing buffer by faking a series of keystrokes. When rewritten using one of the new *ioctl*s, it is simpler, shorter, and less prone to mysterious bugs than the current version.

Completion

Probably the most important element of an easy-to-use command line interface, and unfortunately the hardest to do well, is a completion feature that can automatically provide the rest of a partially-typed command or filename. There are several ways to implement this, none of which is completely satisfactory.

As mentioned above, *csh*-style completion still works with the modified terminal driver. The *csh* feature works by making the system return a partially typed line and then faking keystrokes to get the completed version back into the editing buffer. As the manual notes, this approach is ‘‘ugly and expensive,’’ and it requires each program that needs a completion feature to do all the work itself.

A variation on this theme gives control back to the user program by sending it a signal when the completion character is typed. The signal handler then uses one of the new *ioctl*s mentioned above to retrieve the contents of the current input line. It completes the line and uses another *ioctl* to put the modified line back into the terminal driver’s queue. This approach still requires the cooperation of each program, and also requires the addition of a new signal to the system. The current versions of NetBSD, OpenBSD, FreeBSD, and Linux each have only one slot left for a new signal, and it doesn’t seem very nice to take the last one.

A third approach gives the responsibility for completion to the daemon that is already providing history services. This eliminates the need to include completion code in every program, but it also means that programs cannot tailor the completion routines to meet their specific needs. This is also difficult to implement on a BSD system, because there is no easy way for a BSD user program to find out the working directory of another process, and filenames cannot be completed without knowing this.

Because of these problems, I am still experimenting with other ways the operating system might be able to provide a generalized completion facility.

Conveniences

Some keyboards have Backspace keys, some have Delete keys, some have both, and some have keys labelled Backspace that actually transmit the Delete character or vice versa. Most programs that do their own editing work around this by making both Backspace or Delete erase the previous character no matter which has been set as the erase character. This is just as easy to do in the terminal driver as it is to do in any other program.

In addition, the modified terminal driver can automatically set the erase character appropriately whenever either Backspace or Delete is typed in canonical mode, so raw mode programs are also informed which key is correct. This special treatment for Backspace and Delete can be enabled or disabled by setting or resetting the `L_SETERASE` bit with *stty*. People who prefer to use Delete as their interrupt character will obviously choose to disable it.

Finally, as mentioned earlier, a surprisingly popular feature of *tcsh* is its ability to clear the screen when a user types Control-L. Since the terminal driver does not have access to the *termcap* or *terminfo* database, it does not know what control sequence (if any) will clear the screen. It does, however, usually know how many lines tall the screen is, and outputting that number of blank lines is not a bad substitute.

Conclusions

I have been using various versions of the software described here since early 1997. During that time I have found it very useful to have command line editing features consistently available in every program. If you would like to try this software on your own computer, copies of the source code are available at

```
http://pobox.com/˜enf/ttyedit/
```