HOME TEXT EDITING

Larry Tesler
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto CA 94304

ABSTRACT

Your computer's text editor may be the program you use most. If you don't have an editor you like, it is best to copy the design of an existing system. If you decide to design your own, some guidelines are offered both for the design of the commands and for the design of the program. They focus primarily on "two-dimensional" editors that make full use of a display, but they pertain indirectly to "one-dimensional" editors such as those that can be programmed for teletypewriter style terminals.

1. Text editors don't really edit.

Almost every computer with a terminal also has a program that lets people type and alter text. Usually, that program is called an "editor". However, in English, an "editor" is a person who reads a document and improves its style, clarifies points, verifies arguments, and corrects grammar and spelling. Editing is a complex intellectual task best performed by humans.

A computer can be a powerful tool to assist a human editor. Changes can be made to a text without the person having to completely retype it. But the machine does not improve style, does not clarify points, does not verify arguments, does not correct grammar, and usually does not even correct spelling. So a "text editor" is not really an "editor" at all -- it's just a fancy pencil.

The term "editor" is a misleading name for such a program, not only because of what the program *can't* do, but also because of what it *can* do, which goes well beyond the improvement of someone else's prose. It can help a person to compose one's own prose -- or poetry -- by offering a neatly typed version of each new draft on command. It can do a certain amount of typography, including indentation, spacing, and sometimes justification of margins. It also can help a person to fill out a form, to provide data to a computation, to converse with a dialog system, and neither last nor least, to compose and alter a computer program.

A better name than "text editor" might have been "interactive text processor". Unfortunately, the terms "editor" and "editing" have come into common misuse, so I will reluctantly employ them myself in the rest of this tutorial. Henceforth, whenever I use the term "editor", I will not mean a person, but rather a computer program that lets one type and alter text. The human user of such a program will be called the "operator".

2. Designing a Text Editor.

Many computer users find that the program they use most is the text editor. If you don't like your computer's text editor, or if you want to take on a challenging programming task, it is often possible to concoct your own editor. You can design one from scratch, or you can imitate a design you've seen and liked on a friend's machine. The latter is a far safer course. Designing an editor is difficult, and you may put in a lot of work only to end up with a tool that is frustrating to use.

For those who are thinking of designing an editor, the remainder of this tutorial offers some advice to help in your adventure. Not every aspect of the problem is covered; it is assumed that you have used at least one editor and are familiar with the general concepts of editing.

There are two aspects to the design of a text editor: What will be the command structure? How will the program work? I will call these two aspects "command structure design" and "program design". Command structure design is part of a larger problem known as "user interface design", a problem too large to address here.

Although the design of the command structure and of the program are not completely independent tasks, we'll take them up one at a time. Mainly, we'll talk about command structure design. Most people with sufficient interest can manage the program design for an editor on their own.

3. Command Structure Design.

3.1 Rube Goldberg never sold anything but comic strips.

There are several ways to approach the design of commands for a text editor. If you're designing one just for your own use, then you can dream up any that you like. After all, if you like catsup on your cottage cheese, who's to object?

On the other hand, if you would like friends and family to be additional users of your editor, then it may help to consider the following. Each of us has unconscious idiosyncrasies that make our lives and the lives of our friends more interesting. But when one's isiosyncrasies show up in the design of a program, other people who try to use the program often find it confusing. Even if you do your design by brainstorming with a friend, it's likely that the friend is a Whatsisology major just like you, and that your technical background has prepared you both to think that a Whatsis command is just what the world needs in a text editor.

Therefore, I strongly suggest that you try to explain your command design to someone who majored in Southern Rumanian Poetry or to someone who was lucky enough to get out of school after the eighth grade. If the editor is to be special purpose, explain it to someone who would want to use it for that purpose. When the listener can not understand what you are talking about, please do not think that he or she is stupid. What has happened is that your design, like almost all others that people think up, is the computer equivalent of an Edsel. Learn from Ford's mistake; go back to the drawing boards. And reread the tips that follow.

3.2 You can't bake a cake on a hot plate.

The first constraint on the design of your editor is the physical hardware that your computer has. If there is a display screen, even a one-liner, then use it; it is chock full of advantages. If there is no display, a decent job can be done with your typeout terminal. If you have both a display and a printer, then the display should be used for editing and the printer just for typing neat drafts.

There are two ways to use a display: "one-dimensionally", in which it is just like a fast typeout terminal, but only shows you the last few lines of dialog, or "two-dimensionally", in which many lines of a page are displayed as they would appear on paper, without commands interspersed. From the operator's viewpoint, two-dimensional editors are superior in many ways to one-dimensional ones. However, they are harder to implement; how hard depends on your machine characteristics.

If you choose a one-dimensional approach, or if you have no display at all, then I recommend reading reference [1], which describes the commands and lists the program for a very well implemented one-dimensional editor. The suggestions that follow pertain mainly to a two-dimensional approach, because the problems are less obvious. However, if you are sticking to one dimension, you may still get ideas from the discussion; just translate the concepts from one medium to the other. Sometimes, I do this for you by including [notes in brackets for one-dimensional editors].

The most important part of a text editor is the means the operator must use to specify the part of the text that is to be manipulated; this task is often called "entity selection". In most two-dimensional editors, the operator can select any character or line in the text; the selected entity is often marked by a "cursor". [In most one-dimensional editors, the operator can select any line and by various subterfuges can get at characters within the line.]

It is often useful to be able to select words, sentences, form fields, or other entities of text, or to select the interstices between characters instead of the characters themselves. However, the benefit of such special selections is marginal for many applications. By designing them in, you may simply clutter the editor with unnecessary mechanisms and fill up scarce memory space with the software to implement them.

If your display has a pointing device to go with it (a tablet, joystick, mouse, light pen, etc.), then by all means use it to make selections. If you have an A/D board, you may be able to add a joystick for a few dollars.

Many people think they can control an editor faster from the keyboard than with a pointing device, but some objective studies have found the opposite to be the case.

Even ignoring the issue of speed, pointing devices have advantages over key control. It has been shown that, given a choice of selection methods, operators will usually gravitate to a single general-purpose method and ignore all others, even in cases where they may be faster. A pointing device provides the most general selection method, because every selection -- no matter how far it is from the current selection -- can be made by the same means.

Arrow keys, control keys, space bars, and so forth are less general; with them, the operator ends up devising strategies to get the selection made in an acceptable time. Imagine a car whose steering wheel was replaced by left turn and right turn keys for various angles. It may seem like fun to be devising strategies to get the best use out of a program. However, the fun of toying with the editor wears off when you are trying to get that term paper finished in time or when you are trying to write that neat program or an important love letter.

Another advantage of a pointing device is that it reduces the number of keys needed to run the editor. Sparseness is a virtue in interactive computer programs; the fewer commands and the fewer keys you need, the easier it is to operate the system, even if an occasional command requires extra keystrokes as a result.

Of course, if your computer has no pointing device, then make the best use of the keys. Tens of thousands of people do it every day. If the terminal has cursor control keys, use them if possible. Otherwise, use other keys, but preferably not typing keys. It is very confusing to people when the same key sometimes types the letter W and sometimes selects the next Word in the document.

[If your terminal has no display, then there are many ways to specify a line to select. Each line can have a number semi-permanently attached to it (number them 100, 200, 300, and so on). Insertions can use numbers in between the ones already used. There should be a renumbering command to use just before printing out a draft. It should be possible to print a draft with or without line numbers.]

[Other ways to identify lines are by current ordinal number (constantly changing during the session -- very confusing), by distance from the currently selected line (".+6", ".-10"), by special name ("*" for final line, "↑" for preceding line, etc.), or by sample content (" 'the stove' "). In my experience, most people prefer the last three methods over the others.]

There are other hardware attributes that can constrain your editor design. You may wish "control-M" to mean something special, but on an ascii terminal, that stroke often produces the same character code as a "return", so you can't tell them apart. The agility of the human hand is also a factor; it is easier to strike "control-A' or "control-L" than to strike "control-F" on a standard keyboard layout. It takes a long time to strike a key that is far from the home typing keys, even for a hunt and pecker. So think about which commands happen most often, and be sure to consider any physical disabilities that the potential operators may have.

3.3 Some cuisines.

There are many possible command structures for a text editor. I will try to give a sample to stimulate thought, and suggest some criteria to consider in choosing among them.

If you have have used several pocket calculators, you know that there are "algebraic infix" models where you say "2 + 3 =" and "Polish postfix" models where you say "2 enter 3 enter +". It is even possible to have "prefix" calculators, where you would say "+ 2 enter 3 enter". A similar classification exists among editors.

A "prefix" command structure has the operator specify an operation first and then the selection or selections on which to operate. For example, "Delete (select character) (return)" [or "Delete lines 2 to 10 (return)" in a one-dimensional editor]. Some people like prefix commands because they sound like English sentences and because they require confirmation of the command, which provides an opportunity to change one's mind.

A "postfix" command structure has the operator specify the selection(s) first and then the operation. For example, "(select character) Delete". Postfix commands are usually easier to use because the absence of confirmation requires fewer keystrokes, and because machines to which people are accustomed work that way. On a typewriter, you first move the carriage and then type. On a vending machine, you first make your selection and then pull the handle (and then kick it). On a clothes dryer, you first dial the time and then push start.

The trouble with postfix is that confirmation is not required. If you try requiring it, operators may complain about the superfluous keystrokes. To compensate for the lack of confirmation, each operation should have an obvious converse (e.g., "Insert" for "Delete", or "Undo" for any prior command).

There could in theory be "infix" commands in an editor. To move a paragraph from one place to another, you would say "(select source) MoveTo (select destination)". The trouble with this form is that the command ends with neither a confirmation nor an operation, and people feel uneasy: in postfix systems, they are used to an operation stroke at the end of each command, and in infix systems, they are used to a confirmation at the end. It is better in prefix systems to avoid infix altogether. In postfix systems, it can be approximated by a pair of commands: "(select source) Delete; (select destination) Insert".

In prefix systems, it is useful to allow the same command to be applied to a number of selections without having to repeat the command: "Delete (select something) (return) (select another) (return) (select another) (return)". In postfix systems, it is useful for each command to leave some logical entity selected when it is done (like the character after the one deleted), and for a command to omit a selection specification when that entity is the one desired, e.g., "(select character) Delete Delete Delete" could delete three consecutive characters. The latter is especially helpful if Delete is a repeating key on the keyboard.

If there is a display on the machine, and especially when there is a pointing device for it, there is another way to issue commands, namely, through a "menu". The menu displays alternatives and the operator picks one by selecting it. Menus are very helpful for commands that are not performed often, like "Print pages i through j" or "Find a text string".

An entry in a menu can have space for parameters to be typed in. In a prefix system, the logical thing to do would be to select the command first, then type in the arguments (if any have changed from last time) and then confirm the command. In a postfix system, type in the arguments first (if any have changed) and then select the command.

[The one-dimensional analogy to the menu is prompting. For example, after typing "Find (return)" the program can prompt with "key=".]

3.4 Some recipes.

What commands should you have?

The only command that almost every editor offers is "Delete". It is a good idea for the program to save the most recently deleted passage in a special place, and to offer an "Insert" command to put that passage back in the text before the specified selection. Such an Insert command not only can be used to undo blunders but also provides a way to move things around in a postfix system.

A problem with "Insert" is that one may hit "Delete" twice in a row by accident and thereby loses the ability to undo the first Delete. This can be solved by storing the last two deletions and allowing two inserts in a row to bring both back. More than two are of marginal utility (even two are a luxury).

Another useful command is "Copy", especially if the editor is to be used for writing programs. In a postfix editor, Copy should be like Delete in that it puts a copy of the selection in storage after which Insert can put it somewhere else.

For typing in text in a postfix system, it is best if *no* command need be given. After making a selection, the operator should simply be able to type in the normal manner, and the typing should be inserted at that spot. Requiring a command is prone to error; since people are used to typewriters, they may forget to give the command.

Speaking of typewriters, most people find an interactive program easier to use if it works a lot like a familiar machine. Thus, to retype the preceding character, the operator should be expected to type "backspace" (assuming there is a key that can be so labelled), not some arbitarily chosen key.

Unfortunately, it is not always possible to design an editor so that typing can be done without a command. With a terminal lacking control keys [or with a one-dimensional editor], it may be necessary to use the typing keys for specifying commands. In that case, a command will be needed to enter "insert mode" and some stroke will be needed to leave the mode. Many editing errors result from striking keys without first getting into or out of insert mode; so avoid having modes if you can.

A major problem with typein is what to do when the line fills up. Some editors act like a typewriter: a bell rings and characters soon get discarded. Others allow a line to be quite long, in which case there is generally a way to see the whole thing a little at a time. Still others automatically move the last word of the line to a program-created line below. In some editors of the last kind, a subsequent Delete command may cause words from created lines to move back up. Some people like this arrangement (called "paragraph editing"), but many are confused by it, even with a display. Furthermore, it is hard to implement, so I recommend skipping it on your home computer.

If you don't implement paragraph editing, you may still want a way to specify that a group of lines is a paragraph. This can be done every time a command is to be performed on it, or it can be done at any time, whether by a "Start paragraph here" command, or by establishing conventions, such as a blank line between paragraphs. It depends on what kind of text you will be working with and what you will be doing to it.

If you are editing long texts, rather than just short inputs to a dialog program, then other commands you may want are:
Find next occurrence of a text phrase
Change all occurrences of one text phrase to another ("substitute")
Indent/Unindent paragraph's first line/non-first-lines/all lines
Retype paragraph with as many words on each line as can fit
Retype and insert extra spaces to justify each line but last to right margin
Retype and insert extra spaces to center each line between margins
Paginate every n inches
Print pages i through j, or all pages

There should be a way to abort a command, at least if it is time-consuming, such as Substitute or Print. The keystroke chosen for this purpose should be uniform regardless of the command or the mode.

You will probably want a way to move text between files. A sample set of commands for this purpose are Include and Extract. Each takes a file name or number as an argument. The Include command inserts the whole text of the named file at the position selected in the current file. The Extract command moves the text selected from the current file to the named file. Although faster commands can be designed, these are both easy to implement and easy to use.

A fancier way to deal with multiple files is by dividing the screen into "windows", left and right or above one another, and displaying a different file in each window. The "current" window ought to be clearly distinguished to avoid confusion. In a postfix editor, there should be only one selection on the whole screen, not one in each window, otherwise, many people get confused.

You will want a way to browse through the text. The easiest way to implement browsing is to provide page turning, but it is nicer for the operator to scroll a line at a time. Fancier systems allow browsing by headings or through a table of contents. If you do work by pages, then there is a problem when a page fills up that is the same as that mentioned earlier in connection with lines filling up. The possible solutions are analogous.

There must be a way to move text from the floppy or cassette to the editor and vice versa. There are two philosophies about when to write an updated file back on a floppy disk. One possibility is to work it as you would with a cassette, namely, require the operator to issue a "store" command, either specifying a file name or address, or defaulting to the same file or sector as before, or defaulting to a higher numbered version of the same file. Another possibility is to update "continuously", whenever there is a pause in operation, or whenever a certain amount of work has been done, or whenever that page of the text has been scrolled off the display. This technique provides safety against machine failures or later operator errors. However, it is a good idea for the program first to back up the original version of the text on a temporary file.

If the text you will be editing will often be computer programs, it is nice (if the operating system supports it) to have an "assemble" or "compile" command that writes the text on a file and causes the system monitor to start the assembler or the compiler. Of course, any good interpreter should have a text editor as its front end to enhance the direct execution feature.

3.5 Feedback

When using an editor with either modes or prefix commands, operators often get confused about what they just did and what they can do next. In a two-dimensional editor, you can reserve a "feedback area" in which you announce these things in canned English. If you don't have time to display them continously, it is just as good to display them only when the operator pauses for at least a second, and almost as good to display them only on command. [In a one-dimensional editor, commands can be abbreviated and the program can expand them to supply feedback, but careful: operators get confused when the terminal types in and out at the same time.]

One of the characteristics of a computer editor is that invisible characters like (space) and (return) are usually represented by character codes just like visible characters. This is nice for the program, but when a naive operator is faced with joining two words together, he or she may not understand the concept of "deleting the space in between". It makes no sense, really, to delete something that is not there! Even experts often have trouble telling an invisible character apart from the absence of text.

There are two solutions to this problem, which can be used separately or together. One is to give feedback. Either always, or upon a special command, invisible characters can be replaced by special visible ones to let the operator tell them apart. Depending on your display controller, you may be able to show the difference in other ways (gray background, for example). An easier solution is to let selections be made in empty space on the screen. When typing is done there, have the program surreptitiously insert sufficient spaces before (and sufficient returns above) the selection to give the operator the impression that they were there all the time.

4. Program Design.

4.1 General Considerations.

The amount of "primary" (fast) memory in your machine limits both the program size and the amount of text you can handle quickly. Even if there is not much room for text in the primary memory, you can keep the text on a floppy disk or even on a cassette, except for the page or even for the few lines that the operator is currently dealing with. It is also possible to save program space by putting special features of your editor onto the floppy and bringing them into primary memory only when needed.

The operating system or program library usually provides services useful to a text editor. For example, there should be a buffered keyboard input utility so that the operator can "type ahead" while the system is performing a time-consuming operation.

If the secondary storage is organized into named files, use of the system is simplified. Otherwise, files can be referenced by number or by starting address. Another way to let operators specify files is through a menu. In that case, each file can be identified by a string of text that describes its content rather than by a "name". Even if two files have the same description, they are distinguished by having separate entries in the menu. This alternative is especially attractive if no file system is available and you must provide access facilities yourself.

4.2 Maps.

Editors really aren't that hard to program, so I won't give you much advice. The hardest aspects to deal with are the variable number of characters in lines, the variable number of lines in files, and (in a two-dimensional editor) the continuous update of the display to reflect accurately the current state of the text.

Most solutions to all these problems rest on the concept of a *map*. A map in the abstract is a list of pairs which sets up a correspondence between elements of a *domain* and elements of a *range*. For example, there could be a map from each line number to the memory location of the first character of the corresponding line; a map from line number to screen y-coordinate; a map from paragraph number to starting line number; a map from command name to subroutine location; and so forth.

In concrete terms, most maps have special properties that let them be implemented more efficiently than by a list of pairs. The simplest implementation is a pair of arrays of equal length. If the domain is simply the consecutive integers 1...N, then only the range array is needed. If the range is only defined for domain values from M...N, then no space is needed for the undefined values, as long as the values M and N are available to help access the map correctly.

If the domain is ordered, you can use a "binary search" to search the table for a given value in $\log_2 n$ steps. If the domain values are unique, you can use a "hash table". These techniques can be learned in any good text book on programming, e.g., [2]. However, a straight linear search is preferable on a small machine unless you can demonstrate that it is causing a performance problem.

A map can be inverted so that each line becomes a record whose fields specify the range elements to which certain maps map that line.

The reason to think of all of these diverse data structures as maps is that your program can be simplified if you use a uniform subroutine calling convention to look up in maps and modify maps, regardless of the way they were implemented. One advantage is that if you improve the implementation of a map, the code that uses it won't have to change. A more subtle advantage is that the program will be easier to construct, understand, and improve because of the uniform application of a general abstract concept.

4.5 Text Representation and Display Update

On a system with 8KB of RAM, the program code of a typical editor written in assembly language occupies 2K to 4K. Of course, to achieve such a small program, you must be sparing of features and careful in coding.

There are two basic approaches to the use of the 5K or so of leftover space. The simplest one is to store the current page of text right in the RAM, and to keep the data structures small enough so the largest page you'll ever need will fit. The more complicated approach is to cache only a small amount of text in the RAM, and to leave the rest on secondary storage. Some of the extra RAM space will now be needed to support the caching mechanism, but some may be gained for programming additional features.

The caching approach has the advantage of allowing "pages" of great length to be handled. However, it will either be sluggish or complicated (or both), depending on whether the secondary storage is disk or tape, and whether the operator processes the text sequentially or jumps around a lot.

On a home machine, it is generally better to restrict pages to about the size that can be displayed in one screenful, i.e., 1K or 2K bytes. This eliminates the need for scrolling within a page, as long as commands are supplied for page merging, dividing, and flipping. Furthermore, the text can be stored as a single contiguous string to which any edit can be performed faster than the operator can type.

With this approach, the map for display updates maps line numbers on the screen to starting character positions in the text string. Every edit must update the map as well as redisplay those lines that are affected. Note: Some displays can not redisplay a line without clearing all the lines below it. If this is the case with your system, defer screen update during typein until the keyboard input buffer is empty.

4.4 Reliability

It's never pleasant when a program fails, but when an editor fails, the operator (e.g., you) often attains new levels of agony. It seems as if the last three hours of work were just lost, and irreproducible work at that. You may find the CRT reduced to fragments of glass on the floor.

Consider providing means of backing up files, checkpointing the program, recovering from crashes, and so forth. In critical parts of the program, put validity tests for data structures and for important values, like disk or tape addresses. You can't fit too much of these aids in a small computer, but a little work in this area may save you a lot of distress later.

Careful command design can help, too. Make it more difficult to delete a whole file than to delete a paragraph, and more difficult to delete a paragraph than to delete a character. Avoid situations where if the operator is looking away from the screen, a couple of stray keystrokes can cause disasters.

When using your editor, don't go too many minutes without writing the text on a file, or too many days without getting printouts and backing up files. If the editor is new and still has lots of bugs, exercise these precautions more frequently.

5. Conclusions.

Designing a program is like writing a story: the possibilities are endless. It is helpful to establish guiding principles in the design to constrain it in a manner appropriate to one's goals.

In the design of an editor, I recommend that you consider hardware limitations, operator habits, how much awareness the operator can devote to the system as opposed to the task at hand, and requirements for speed and reliability.

The Author

The author was turned on to computers in high school in 1960. He helped finance undergraduate work at Stanford by programming in such areas as text processing, simulation, statistics, graphics, and compilers; ran a freelance software company in Palo Alto for five years; then was a research assistant at the Stanford Artificial Intelligence Lab doing cognitive computing, natural language understanding, higher level languages, and text formatting. Somewhere in there he took off a year to live in rural Oregon and develop other interests such as folk music, cooperatives, and extra-sensory perception. For the past four years, he has been a member of the research staff of Xerox Corporation, specializing in text processing, user interfaces, and programming systems. He likes programming as well now as in 1960. He has never learned to operate a soldering iron.

References

[1] F. J. Greeb, "A Classy 8080 Text Editor", in *Dr. Dobb's Journal*, Vol. 1, No. 6, June/July, 1976.

[2] D. Knuth, *The Art of Computer Programming* (especially Volume 3), Addison-Wesley, 1973.