Contents

Preface

Just over eleven years ago I was faced with selecting a topic for my thesis. At the time, I was a student at the Massachusetts Institute of Technology and was working on my bachelor's degree in Computer Science and Engineering. One of the degree requirements was a thesis, and you can't have a thesis without a topic.

During my four years at M.I.T., a new type of text editor had come into being and widespread use. This type of text editor was called "Emacs," and it was a major step forward in many ways. Implementations of this type of editor were appearing on many computer systems. Some people even used an implementation as the basis for their thesis. I took a different tack. The idea that I settled on for my thesis was a description of the technology that underlies all text editors, but with a special emphasis on Emacs-type editors. The thesis was written and published as a technical memo (Finseth 1980).

* * *

Ten years later, I was reading the USENET News news group Comp.editors, one of the many facets of that worldwide electronic bulletin board. A discussion thread had started up in which *both* sides of the discussion were citing my thesis as the authority in the field. Further inquiries (not by me: I was just reading along) showed that no one in that group was aware of any other document that described general text-editing technology.

My thesis was ten years old: it predated most personal computers and workstations. There had even been a chapter in an early draft that attempted to prove that it was not possible to implement an Emacs-type text editor on a small computer. (I invented a way, threw out the chapter, and with some friends started a software company to market such an editor. Oh well.) It was clearly time for a complete rewrite, and that rewrite is what you are reading now.

If you don't have a copy of my thesis (or the *Technical Memo*: the two have identical content), you won't miss anything. This book has all of the information from the earlier document, and is now updated. It also has a whole lot more. Every part has been completely rewritten and expanded, and major sections have been added.

As with my thesis, this book is written in an informal, almost chatty style. It is addressed directly to "you," who are assumed to care about how text editors are implemented. Be warned, however, that it also contains opinions about the "right" and "wrong" way of doing things and that these opinions hold that many of the current directions and trends are – shall we say? – not the "right" way. You should keep in mind that you should not accept everything said in here as the gospel truth, but understand why I say what I say and make your own informed judgment.

This book is addressed to anyone who implements large software systems or who wants to know the considerations that go into such systems. It focuses around text editors. Although not required, an understanding of programming will be helpful.

Questions to Probe Your Understanding

Each chapter ends with a set of questions and problems designed to probe your understanding of the material that was just presented. And, true to the Socratic method, some of these questions also introduce new material. The level of difficulty of the questions ranges from very easy to quite difficult, and each question is labelled to help you gauge how much effort is required. Just as with most programming issues, most questions have no single correct answer.

Acknowledgements

would like to thank those people who helped me in various ways:

Owen "Ted" Anderson
Joe Austin
Jeff Brown
Bernard Greenberg
Brian Hess
Mike Kazar
Richard Kovalcik
Scott Layson
Jason Linhart
David Moon
Robert Nathaniel
Lee Parks
Jeffrey Schiller
Richard Stallman
Seth Steinberg
Peter Steinmetz
Liba Svobodova
Daniel Weinreb

plus, of course, all of those people that I have left out. Special thanks to my wife Ann and daughter Kari, who put up with my typing away all the time.

Craig A. Finseth
St. Paul, Minnesota
February 1991

Introduction: What Is Text Editing All About?

'Twas brillig, and the slithy toves
  Did gyre and gimble in the wabe:
All mimsy were the borogoves,
  And the mome raths outgrabe.

In its most general form, text editing is the process of taking some input, changing it, and producing some output. Ideally, the desired changes would be made immediately and with no effort required beyond the mere thought of the change. Unfortunately, the ideal case is not yet achievable. We are thus consigned to using tools such as computers to effect our desired changes.

Computers have physical limitations. These limitations include the nature of user-interface devices; CPU performance; memory constraints, both physical and virtual; and disk capacity and transfer speed. Computer programs that perform text editing must operate within these limitations. This book examines those limitations, explores tradeoffs among them and the algorithms that implement specific tradeoffs, and provides general guidance to anyone who wants to understand how to implement a text editor or how to perform editing in general.

I do not present the complete source code to an editor, nor is the source code available on disk (at least from me: see Appendix B). For that matter, you won't even see a completely worked out algorithm. Rather, this book teaches the craft of text editing so that you can understand how to construct your own editor.

The first chapters discuss external constraints: human mental processes, file formats, and interface devices. Later chapters describe memory management, redisplay algorithms, and command set structure in detail. The last chapter explores the Emacs-type of editor. The Emacs-type of editor will also be used whenever a reference to a specific editor is required.

This range of topics is quite broad, and it is easy to lose sight of the forest with all of those trees. The remainder of this introduction will sketch the outlines of the forest by examining an editor-in-miniature: a get-line-of-input routine. We will start with a basic version of the routine, then make it more elaborate in a series of steps. By the end, you will see where the complexity of a text editor arises from.

The program examples are written in the ANSI version of the C language. Appendix A provides a brief introduction to the C language and explains all of the features used in examples.

The Basic Get_Line

The **Get_Line** routine accepts these inputs:

- a prompt string
- a buffer to accept the input; this buffer must be at least two characters long
- an indication of the buffer length

and produces these outputs:

- a success/fail status
- if the status is "success," the input is stored in the supplied buffer; the end of the input is marked with the NUL (^@, 0 decimal) character
- if the status is "fail," the buffer may have been modified but will not contain valid input

The editing performed by this routine is on the input buffer. This first version assumes that you are creating a new item from scratch each time.

Version One

```
FLAG Get_Line(char *prompt, char *buffer, int len)
        {
        char *cptr = buffer;
        int key;

        if (len < 2) return(FALSE);             /* safety check */
        printf("%s: ", prompt);
        for (;;) {
                key = KeyGet();
                if (isprint(key)) {
                        if (cptr - buffer >= len - 1) Beep();
                        else    {
```

```
                                *cptr++ = key;
                                printf("%c", key);
                                }
                        }
                else if (key == KEYENTER) {
                        *cptr = NUL;
                        printf("\n");
                        return(TRUE);
                        }
                else    Beep();
                }
        }
```

Version One accepts input until the user presses the Enter key. If a user's input will overflow the input buffer, the input is discarded and the program will sound an error beep. Once the Enter key has been pressed, the program appends a NUL character to terminate the string and returns True. Non-printing characters other than Enter also cause the program to sound an error beep. Simple, straightforward, and useless, as there is no way for the user to correct any typing mistakes.

Version Two

Here is version Two. It adds editing:

```
FLAG Get_Line(char *prompt, char *buffer, int len)
        {
        char *cptr = buffer;
        int key;

        if (len < 2) return(FALSE);             /* safety check */
        printf("%s: ", prompt);
        for (;;) {
                key = KeyGet();
                if (isprint(key)) {
                        if (cptr - buffer >= len - 1) Beep();
                        else    {
                                *cptr++ = key;
                                printf("%c", key);
                                }
                        }
                else    {
                        switch (key) {
```

```
                        case KEYBACK:
                                if (cptr > buffer) {
                                        cptr--;
                                        printf("\b \b");
                                        }
                                break;

                        case KEYENTER:
                                *cptr = NUL;
                                printf("\n");
                                return(TRUE);
                                /*break;*/

                        default:
                                Beep();
                                break;
                                }
                        }
                }
        }
```

Version Two starts developing problems that can no longer be swept under the rug.

Version One glossed over exactly what is meant by the Enter key. That's sort of okay. Most keyboards have only one key labelled "Enter" or "Return" or something similar. It almost always sends a Carriage Return character. The program can compare against just that character and almost always operate "correctly," *i.e.*, as the user expects. However, most keyboards have at least *two* keys for erasing: Back Space and Delete. Some people and computer systems use one of these. Other people and computer systems user the other. (We will ignore any extra "erase" or "delete character" keys that you might find. For now.) The program can handle this problem in several ways:

- accept only one or the other
- accept both
- if the operating system supports some sort of "terminal parameter configuration," ask the operating system what character to use provide a configuration option in your program to let the user set his or her preferred character; the option will most likely default to the operating system configuration setting if one is available

If you picked the first option, just over half of your users will be upset with you. The second option is much better: almost all users will like you, and this part of your program need not be operating system specific at all. (I often select this option when writing small programs that should have a minimum of operating system-dependant code.) The third option is a fine solution. Most users will like you, and you are building on other work (*i.e.*, the operating system) instead of reinventing the wheel.

If you picked the fourth option, you have already learned what an Emacs-type editor is about. Implicit in this option is recognizing that users should be able to control their environment as much as possible. Yes, it is more work to write such programs and, yes, it sometimes overlaps the existing operating system, but it can be well worth the effort.

Another problem appears in the statement:

```
printf("\b \b");
```

This statement is a crude attempt at erasing a character. As it turns out, there are pretty powerful conventions regarding how printing characters and newlines are handled by operating systems and output devices. These characters all move the cursor to the right or to the start of the next line. However, when you want the cursor to back up in any way or you wish to control it in any other way, you are on your own: there are no industry-wide conventions for specifying these operations. And, with no conventions to rely upon, your program has to implement a method of coping with the range of output devices.

Version Three

Version Three assumes that the input buffer contains some text. This text is used for the response if the user just presses Enter (*i.e.*, the text is the default value):

```
FLAG Get_Line(char *prompt, char *buffer, int len)
	{
	char *cptr = buffer;
	FLAG waskey = FALSE;
	int key;

	if (len < 2) return(FALSE);		/* safety check */

	for (;;) {
		ToStartOfLine();
		ClearLine();
		printf("%s: %s", prompt, buffer);
		key = KeyGet();
		if (isprint(key)) {
			if (!waskey) {
				*buffer = NUL;
				waskey = TRUE;
```

```
                                }
                        if (cptr - buffer >= len - 1) Beep();
                        else    {
                                *cptr++ = key;
                                *cptr = NUL;
                                }
                        }
                else    {
                        switch (key) {

                        case KEYBACK:
                                if (!waskey) {
                                        *buffer = NUL;
                                        waskey = TRUE;
                                        }
                                if (cptr > buffer) {
                                        --cptr;
                                        *cptr = NUL;
                                        printf("\b \b");
                                        }
                                break;

                        case KEYENTER:
                                printf("\n");
                                return(TRUE);
                                /*break;*/

                        default:
                                Beep();
                                break;
                                }
                        }
                }
        }
```

Version Three returns the supplied response if the user just presses the Enter key. Otherwise, the supplied response is erased completely the first time a printing key or Back Space is pressed. The only other changes worth noting are that the prompt has been moved to the inside of the loop and a few terminal interface routines have been added. The first one moves the "cursor" to the beginning of the line. The next clears the line.

Version Four

This version adds a number of features:

- commands to move the cursor left and right
- insert/replace editing
- a command to delete the character to the right of the cursor
- commands to move to the beginning and end of the response
- a command to clear the response
- a command to clear the changes and restore the default
- a way to insert arbitrary characters, including command characters, into the response
- a cancel key
- a redisplay key

This version of the routine also has a slight change to the interface: the addition of a separate default value parameter.

```
FLAG Get_Line(char *prompt, char *buffer, int len, char *default)
        {
        char *cptr = buffer;
        FLAG isinsert = TRUE;
        FLAG waskey = TRUE;
        int key;

        if (len < 2) return(FALSE);             /* safety check */

        strcpy(buffer, default);
        for (;;) {
                ToStartOfLine();
                ClearLine();
                printf("%s: %s", prompt, buffer);
                PositionCursor(strlen(prompt) + 2 + (cptr - buffer));

                key = KeyGet();
                if (isprint(key)) {
                        if (!waskey) {
```

```
                        cptr = buffer;
                        *cptr = NUL;
                        waskey = TRUE;
                        }
                if (isinsert) {
                        if (buffer + strlen(buffer) >= len - 1) Beep();
                        else    {       /* move rest of line and insert */
                                memmove(cptr + 1, cptr, strlen(cptr) + 1);
                                *cptr++ = key;
                                *cptr = NUL;
                                }
                        }
                else    {
                        if (*cptr == NUL) {
                                        /* end of input, so append to buffer */
                                if (buffer + strlen(buffer) >= len - 1)
                                        Beep();
                                else    {
                                        *cptr++ = key;
                                        *cptr = NUL;
                                        }
                                }
                        else *cptr++ = key;     /* replace */
                        }
                }
        else    {
                switch (key) {

                case KEYBACK:
                        if (!waskey) {
                                cptr = buffer;
                                *cptr = NUL;
                                waskey = TRUE;
                                }
                        if (cptr > buffer) {
                                xstrcpy(cptr - 1, cptr);
                                cptr--;
                                *cptr = NUL;
                                }
                        break;
```

```
case KEYDEL:            /* delete the following char */
        if (cptr < buffer + strlen(buffer))
                xstrcpy(cptr, cptr + 1);
        else    Beep();
        break;

case KEYENTER:
        printf("\n");
        return(TRUE);
        /*break;*/

case KEYLEFT:
        if (cptr > buffer) cptr--;
        waskey = TRUE;
        break;

case KEYRIGHT:
        if (cptr < buffer + strlen(buffer)) cptr++;
        waskey = TRUE;
        break;

case KEYSTART:          /* move to start of response */
        cptr = buffer;
        waskey = TRUE;
        break;

case KEYEND:            /* move to end of response */
        cptr = buffer + strlen(buffer);
        waskey = TRUE;
        break;

case KEYQUOTE:          /* insert the next character,
                                even if it is a control char */
        if (!waskey) {
                cptr = buffer;
                *cptr = NUL;
                waskey = TRUE;
                }
        key = KeyGet();
        if (isinsert) {
                if (buffer + strlen(buffer) >= len - 1)
```

```
                                Beep();
                        else    {       /* move rest of line and insert */
                                memmove(cptr + 1, cptr,
                                        strlen(cptr) + 1);
                                *cptr++ = key;
                                *cptr = NUL;
                                }
                        }
                else    {
                        if (*cptr == NUL) {
                                        /* end of input, so append */
                                if (buffer + strlen(buffer) >= len - 1)
                                        Beep();
                                else    {
                                        *cptr++ = key;
                                        *cptr = NUL;
                                        }
                                }
                        else *cptr++ = key;     /* replace */
                        }
                break;

        case KEYCLEAR:          /* erase response */
                cptr = buffer;
                *cptr = NUL;
                waskey = TRUE;
                break;

        case KEYDEFAULT:        /* restore default response */
                strcpy(buffer, default);
                cptr = buffer;
                waskey = FALSE;
                break;

        case KEYCANCEL: /* abort out of editing */
                return(FALSE);
                /*break;*/

        case KEYREDISPLAY:      /* redisplay the prompt and resp */
                break;
```

```
                        case KEYINSERT: /* set insert mode */
                                isinsert = TRUE;
                                break;

                        case KEYREPLACE:        /* set replace mode */
                                isinsert = FALSE;
                                break;

                        default:
                                Beep();
                                break;
                                }
                        }
                }
        }
```

Version Four does all that was claimed for it, but not as well as one would like. In particular:

- it did not check to ensure that the default response fits within the buffer
- there was no way for the user to determine whether the program is in insert or replace mode except by typing a character and finding out what happens
- it assumes that all characters are the same width when displayed
- it did not address the question of what the commands are nor how the user supposed to remember them all

The Forest

The examples presented in this chapter bumped into these problems:

- What characteristics of the display and keyboard affect text editing?
- How should the program cope with presenting output on different displays?
- What view of the text should be presented to the user?
- How should the text be managed so that large amounts of text could be edited efficiently?
- How should display updating occur so that editing changes are efficiently presented to the user?

- How should the command set be designed? What should the meanings of the various commands be?
- How should the program be designed so that the user can change how it operates?

These and other questions will be addressed in the remainder of this book.

Questions to Probe Your Understanding

Modify the latest version of **Get_Line** to accept only numeric responses. What sort of error messages should be given? (Easy)

Modify the latest version of **Get_Line** to accept only responses from a list that is passed in as a parameter. What sort of error messages should be given? (Easy)

What are two good formats for such a list (Easy for those familiar with C, Medium otherwise)

What is the appropriate degree of control (key definitions, enable / disable features, etc.) that the calling program should have over the input editing? (Medium)

Chapter 1: Users

"Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!

The saying goes: "Business would be great if it weren't for customers." Well, programming would be easy if it weren't for users. In the simple case, there would be exactly one user for your program, - yourself - and you would use it only once. Most programs, however, are used many times by many people. You must take those users into account when designing your program.

This chapter will only review those aspects of users that are most relevant to text editing: full discussions of users and design can and do fill many books in themselves, some of which are listed in the Bibliography. This chapter (and this book) does not address the question of non-people users.

User Categories

Each user can be placed in a category. Each category is described in terms of the *amount* and the *type* of experience. It is important to understand users: each user creates a model of how a new program works based on his or her experience with other programs combined with the "hints" that your program's user interface gives to him or her. It is up to you to either match your program's behavior to your users' model(s) or to give them enough information so that they generate a model that is well-matched to your program.

Amount of Experience

The amount of experience that a user has is a point on a continuous scale. All users start with no experience and accumulate experience as they learn. Although the scale is continuous, I have divided it into five regions in order to simplify discussion. Also, this list is not intended as a self-rating scale: most of you who are reading this book will be programmers.

Neophyte users barely know what a computer is. They lack understanding of such "basic" terms as "file" and "file name" (the concepts behind these terms are actually quite sophisticated). This lack of understanding does *not* mean that they are unintelligent people, only that they have never had a reason to learn these concepts. If you are designing a program for this type of user, you may feel both blessed and cursed. Cursed because it can be so difficult, and blessed because this area of program design has such a pressing need for good designs.

Experience from the field of artificial intelligence can shed more light on this issue. AI researchers found it (comparatively) easy to write programs that can handle advanced mathematics such as freshman calculus. However, as the researchers pushed on to handle such easy (to most people) areas as filling in coloring books, the programming problems got harder and harder. Some of this difficulty is due to the fact that the task of teaching college-level courses is well understood–especially by college professors–but teaching coloring is not. For example, how many textbooks have you seen on "how to color"? More to the point, computers have been designed to process information in a certain way, one that is mathematically elegant, but not necessarily related to how people's minds work. As people write programs for more and more "basic" tasks, this difference becomes increasingly apparent.

Many programs have been (mis-)designed for neophyte users. They often offer a few simple commands, yet leave intact such difficult concepts such as that of a "file." They solve the wrong problem, sort of like travelling to a place where a foreign language is spoken, and trying to communicate by speaking your native language slowly and distinctly. As a program designer, you must understand the thought structure of your users, and design programs that match that structure. The blessing comes from designing programs that are very different from "conventional" programs and which are well-matched to their users.

Novice users have used a computer before, perhaps for text editing, word processing, spread sheet, or database applications. In any event, novice users have some familiarity with the idea of typing things into a box and seeing a response that somehow reflects their typing. They understand how a shift key works, that a lowercase letter 'l' is not the same as a digit '1', and so forth. They even have some understanding of the idea of "context:" that keys do different things at different times. Users with this amount of experience are able to operate almost any program that has a good design and a decent manual.

Basic users are like novice users, only more so. They understand such programming concepts as thread of control, variables, and statements like "A = A + 1" (in fact, many people call such users "programmers"). These users can operate any program, even one with a poor design. Given source code to the program they are able to customize and extend it, albeit in what might be an awkward fashion.

Power users know one or more application programs thoroughly. They understand not only how to use those programs fully, but can often go beyond the bounds of what the original designers intended. They may write large programs often in the form of application macros, but do not *design* these programs. These users understand the fine points of the programs that they use.

Programmer-level users understand the theory of programming. When writing a large program, they design the program before implementing it. They generalize, applying their experience and their knowledge of one program to guess how another program will operate.

Type of Experience

The amount-of-experience scale is one-dimensional: people start at the beginning and proceed along the scale as they gain experience. This type of experience scale is more like a collection of baseball cards. A user can collect the experience types (cards) in any order, and two people with the same number of experience types (cards) may have no experience types (cards) in common.

The experience types do not necessarily carry over: experience gained on one type of system may or may not prove useful on another. Actually, experience gained on one system may make it more difficult to learn another. And, if users grow to like one type of system, they may then dislike another one, thus making any experience transfer problematic.

These experience types can have a major effect on the design of your programs, as it is usually important for new programs to appear and operate in a manner similar to existing programs. Thus, the (possibly bad) designs of those existing programs may have to be carried into the design of your program.

"Religion"

This section might also be titled "religious preference." In the computer field, "religion" is a technical term that refers to the usually irrational and extreme preference of one program, style, or method to another. Although you cannot really do anything about this phenomenon, you can keep it in mind when analyzing comments on your design.

It has been observed that people often "get religion" over the first application (for example, a word processor) that they use. I can't recall the number of people who have tried to convince me that the program that they just discovered (*i.e.*, the first one they used) is the best one in the world. This form of "religion" is normal and derives from the facts that (1) the move from manual to automated methods (*e.g.*, from typewriters to word processors) involves a major increase in capabilities: even the simplest word processor provides vastly more capabilities than does a typewriter, and (2) new users do not have the experience to realize that all programs (*e.g.*, word processors) are not equal. This form of religion usually fades away over time as new users gain experience.

In a hauntingly close parallel to the "second system effect" (Brooks 1982), the "second program users" are the ones to watch out for. These people started using one program, then gave that program up in favor of a second one. The problem is that they think that since the second program is better than the first one (which it usually is), it must therefore be better than all the rest.

There is nothing in particular that you can do about users that feel religious about a program: rational arguments are in general ignored. You can, however, be aware that such users exist, and recognize when you are dealing with one.

User Goals

Knowing your user's experience is essential, but a program design must incorporate knowledge of what task or tasks the user is trying to accomplish. For text editors, he or she might want to create:

- some jottings, format not important;
- some jottings, format important;
- some jottings within a structure (*e.g.*, outliners);
- something with a specialized format (*e.g.*, a business letter or a poem);
- a short narrative document (*e.g.*, a school paper or short story);
- a longer document (*e.g.*, a long paper or book);
- a long document with complex formatting (*e.g.*, a mathematics textbook);
- a computer program;
- data for a program; or
- something else.

The frequency of doing these tasks can range from occasionally to continuously. Different tasks can be performed by the same user with different frequencies.

The style of doing these tasks can also vary. One person may do all of one task, then start on the next. Another person may be frequently switching among two or more tasks.

Physiological Constraints

Users are people. There are limits to what people can do. These limits must be considered when designing a program.

Hands have a limited reach. The very act of reaching for one key draws a hand away from other keys. Thus, commands that you expect to follow one another should be assigned with that constraint in mind. Function keys are often difficult to find and awkward to press. While there are almost always two shift keys, most keyboards only have one control (or equivalent) key and may only have one of other types of shift keys. Thus, it is difficult to press some shifted keys (such as control-P) with just one hand.

Non-keyboard devices such as mice draw a hand far away from the keyboard – and you don't in general know whether it is the left or right hand that is drawn away. A sequence such as control-mouse button may be very difficult for some (*i.e.*, left-handed) users to type.

Eyes can focus on a limited area of high resolution surrounded by a large area of lower resolution. However, areas of strong contrast such as reverse video are still visible in low-resolution areas. Blinking items are not only visible, but will draw the eye to them. "Status" displays should therefore change as quietly as possible so as not to draw the eye away from the text under edit. For example, it may make sense to place such status areas on the top part of the display if insert/delete line operations cause visible motion of the bottom part.

The mind (or brain), however, places the greatest constraints on editor design. It is only capable of processing a few thoughts ("instructions") per second. In order for users to be productive, it is important that these thoughts be directed as much as possible to useful editing operations. There are several things to consider regarding these thoughts.

First, mental effort (thought) is required to translate between the display representation of the text being edited and the user's internal representation. The WYSIWYG ("what you see is what you get") principle reduces this effort by reducing the amount of thought required. Note that in general WYSIWYG does *not* mean "fancy output on a graphics display." Rather, it means "it is what it appears to be, no more and no less."

Second, the mind has expectations: it sees (and in general senses) what it *expects* to see. In extreme cases, if something totally unexpected happens, it can take many seconds for the mind to even recognize that there is an unexpected image, in addition to the time required to process the image and make a decision. Thus, it is important for the program to anticipate what the mind will expect to see and to arrange the display accordingly.

Third, it takes mental effort to handle special cases. For example, if the delete operation deletes everything except for newlines, it takes effort to remember that difference and to monitor each command that is being given to ensure that it conforms to the restriction. Fourth, it takes mental effort to plan ahead. The design of the editor should make it easy for the user to change his or her mind.

Last, it takes mental effort to track modes. (Chapter 9 goes into modes in detail.) Each time a new mode is introduced, it takes mental effort to track the state of the mode and adds effort to the process of switching modes.

The mind's short-memory can hold from five to seven "chunks" of information (Norman 1990). These chunks are organized in a cache-like form. When the chunk cache fills up, chunks must be stored in "main memory," a process that takes time. Considering that some of these chunks are used to remember what is being edited, why the editing is being done, and other such context, it becomes clear that the editor should be designed to use as few of these "chunks" as possible.

The mind is poor at thinking numerically. It is much easier to think in terms of "put that there" than "put object 12856 at location 83456." These last two points mean that the computer should do as much remembering as possible for the user.

Applying These Physiological Constraints

Let us examine how these principles apply to a particular user: me. I select myself as the example for the simple reason that I understand how my mind works better than I understand anyone else's.

First, I almost always work with plain ASCII files. Hence, I can take advantage of WYSIWYG on even a simple ASCII terminal.

Second, the program/computer combination that I use can (mostly) keep up with my typing in real time.

Third, the Emacs command set that I use is very regular, so my mind need only keep track of a few special cases.

Fourth, the basic paradigm behind the Emacs command set is "move to desired position, make desired change." This paradigm applies *even in the case where I made a mistake*, as I simply add the mistake to the list of changes to be made and continue to apply the pardigm. I never have to change mental gears. The penalty for making a mistake is thus minimized.

Fifth, the program minimizes what I need to remember: the text being edited is there to be seen, exactly as is, and there are very few state variables to track. In addition, the Emacs command set is defined mainly in terms of objects (character, word, sentence, etc.) and has a convenient way of saying "some," "a lot," "a whole lot," and "a huge amount." (Various aspects fo Emacs command set are discussed in later chapters.)

Going beyond these principles, I have used the Emacs command set so long (thirteen years) that I quip that most of my editing is performed by my spinal cord and not my brain. Although this quip is not true since the spinal cord can only handle purely reflex actions, we will look closely at how my mind functions when editing text. The mind of any other experienced user should operate in a similar fashion.

As I write this text, part of my mind is articulating the point that I am trying to make, while another part is expanding those words into their component characters. Call these parts the "source process." Another part of my mind is translating those characters into finger motions. Call this part the "keystroke process." Other parts of my mind are reading the text as it appears on the screen, turning it back into words, and matching these words against the original word stream. Call this the "feedback process."

These three processes work in any sort of writing: using a computer, typewriter, or pen. All people who write use them. However, if the resulting text is to have few errors, one of two things must have happened: either the user made very few mistakes (thus minimizing the number of errors to be corrected) or the user must have written slowly, giving the feedback loop enough time to recognize an error before too much time has elapsed and the error becomes difficult to correct (such as an omitted character on the previous line or page).

With the advent of computers, and their ability to make seamless corrections, a third option appeared: a new, fast feedback loop. This loop operates by giving the keystroke process the ability to recognize that it made a mistake. This extra ability is not useful without seamless editing, as it takes a long time to use the eraser or correction tape. However, with (a lot of) practice, a fourth process can be "running:" the "editing process."

The editing process takes the feedback from the keystroke process and inserts editing commands into the character stream created by the source process. Here is an example of how this editing might operate to correct an error when writing the text "the quick red fox."

- The source process generates the appropriate character string.
- An error occurs. What is actually typed on the keyboard is the string "teh".
- The keystroke process recognizes the error just after the "h" was typed.
- The editing process takes time to run. In this time, let us suppose that the characters " quick" (that is a space, followed by "quick") were placed in the "output queue." It is reasonable to suppose that all of the characters for each word will be placed on the queue in one operation.
- The editing process then places its own string of characters on the output queue. This string will correct the error. For the Emacs command set, the sequence might be "^[b^B^T^E". This sequence means "move back a word ("^[b"), move back one more character ("^B"), interchange the two switched characters ("^T": swaps "eh" to "he"), return to the end of the line ("^E").
- The rest of the line is then processed as usual, with the characters " red fox" placed on the queue in two chunks and eventually typed correctly. At some later time, the feedback process confirms that the phrase was typed correctly.

(Other users may have variations on this process. For example, they may always delete all of any word with an error and retype the word.) With the extra fast feedback loop, the fingers were kept typing at full speed all the time. Granted, an extra five characters were typed, but consider what would happen without the extra loop. It could well be that the entire phrase would have been typed before the error was noticed. The source process would have already started on the next phrase. When the feedback process notices the error, the smooth typing of characters would stop as the user's mind determines exactly which corrections are required and how to perform them. It must then start the pipeline going again. The stopping, correcting, and starting again takes several seconds. A fifty word-per-minute typist is typing about five characters per second. The Emacs correction string would take one second to type. There is thus a direct saving of some seconds and an indirect saving due to not having interrupted the smooth flow of thinking.

Note that the design of the command set played an important part in making this loop usable. For example, if no "go backward word" operation were available, the editing process would have to compute how many characters were in the "output buffer," an operation that is quite time-consuming (quick: how many letters in "brown"?) as well as not well matched to how the mind works.

Some recent industry trends illustrate how some "user friendly" designs clash with this editing process. Consider a typical, modern window system. In some ways, it acts to frustrate an experienced user. For example, when a user closes a modified file, the computer may put up a dialog box that says "Discard changes? Yes, No, Cancel" (or words to the same effect). This prompt will be displayed in a beautiful dialog box, neatly centered on the screen. Each response will have its own button. Unfortunately, even if the user is expecting the dialog box, he or she may have wait for the system to catch up for these reasons:

- The operating system does not know about the dialog box until the program has informed it of the box. Hence, if the mouse button is pressed too early, the button-press event will be sent to the main window and not the dialog box.

- While the user knows that the dialog box will appear in the center of the screen, it is in general too difficult to predict precisely where the (dialog box) button will show up to be able to "mouse ahead." Hence, the mouse button cannot be pressed until the box is drawn.

For these reasons, an experienced user's editing process may be interrupted. These interrupts no doubt contribute to the feeling of sluggishness that many experienced users still feel when using such systems. The challenge is to design your program so that experienced users can productively use your program. The steps that you can take to faciliate this use include:

- keep dialog box choices consistent

- provide keyboard responses for all choices

- provide for typeahead

- be prepared to handle mis-direcdted events

In general, the goal is for an experienced user to be able to accurately predict which responses will be required, and to reliably supply those reponses in advance of the prompts. In this way, experienced users can continue to do their work, without being slowed down by the system.

Users Who Have Handicaps

When someone has a significantly reduced ability to do something, that person is considered to be handicapped in that area. The reduced ability might be physical, such as reduced hand motion or poor eyesight, or it might be mental, such as a reduced ability to remember things.

While the number of people who have severe handicaps in many areas is small, a large number of users have at least limited handicaps in a few areas. As it is important for programs to accommodate as wide a range of users as possible, programs must accommodate users with handicaps.

It is also important to keep in mind that those users that have severe and/or multiple handicaps can benefit greatly from the use of computers.

Sometimes, even users without a handicap benefit from designs intended to aid users with handicaps. For example, adding a wheelchair ramp to an old building also allows other people to roll heavy objects up the ramp instead of having to use stairs.

The main design principles to follow to take into account users with handicaps are:

- Reduce mental complexity: have the user deal with only one object or concept at a time.
- Reduce visual complexity: keep displays clean and to the point and avoid clutter.
- Reduce manual complexity: allow the user to do everything with just one finger (this does *not* mean to *force* the user to do everything with one finger). Keep commands simple. Allow shortcuts and shorthand where applicable.
- Provide for customization. If you are lucky, the operating system will do this for you.

It is not surprising that these are also good design rules for users without handicaps.

Questions to Probe Your Understanding

(Some of these questions refer to marketing decisions. A designer must also take into account those people who are not yet users. Remember that purchasers are "users" too.)

Consider the case where the higher you go in an organization, the less computer experience people have. Assume that product purchase decisions are made at a higher level than the product user. How does this inversion affect product design? Product marketing? (Medium)

Many product reviews include "feature checklists" or "scoreboards." These checklists in general include all features found in all related products. What are the pros and cons of these checklists for manufacturers? For users? (Medium)

I have observed that, all other things being equal, people will buy the *more* expensive of two application programs. Why? (Easy)

Productivity falls off as computer response time increases. However, the fall-off is not linear, but happens in a series of thresholds, where slight increases in response time cause large drops in productivity. Why do these thresholds exist? What information do you need about human physiology in order to calculate where the thresholds are? (Hard)

How would you design a *program* to best be used by someone with dyslexia? What about the entire computer system? It is okay to be extreme and to make it less usable by other people. (Medium)

Chapter 2: User Interface Hardware

Beware the Jubjub bird, and shun
The frumious Bandersnatch!"

User interface hardware is the collection of devices you use when interacting with the computer. The currently available user interface hardware usually consists of a display screen for output and a keyboard and perhaps a mouse or other graphical input device for input. This chapter will first discuss the output side: the screen. It will then discuss the input side: the keyboard. Finally, it will discuss the communications paths that tie the two parts together.

Display Types

In the old days (*i.e.*, the early 1980s), almost all displays were part of character-based terminals. Differences in capabilities among the terminals were often crucial. These differences play an important part in the types of redisplay schemes that are workable (redisplay is discussed in Chapter 7). Thus, it is worth reviewing the old display types.

TTY and Glass TTY

A TTY is the canonical printing terminal. Printing terminals have the property that what is once written can never be unwritten. A glass TTY is the same as a TTY except that it uses a screen instead of paper. It has no random cursor positioning, no way of backing up, and no way of changing what was displayed. They are quieter than printing terminals, though.

When a text editor is used on one of these displays, it usually maintains a *very* small window (*e.g.*, one line) and either echos only newly typed text or else constantly redisplays (*i.e.*, reprints) that small window. Once a user is familiar with a display editor, however, it is possible – in a crunch – to edit from a terminal of this type, but this is not generally a pleasant way to work.

Although one would hope that this type of display was gone for good, it does crop up from time to time in poorly implemented window schemes. Some window schemes offer window interfaces that resemble printing terminals – all too well.

You may encounter one other type of "write only" scheme: a Unix-style output stream. As an editor writer, you may want to check for this and either:

- alter your output accordingly, or
- *don't* alter your output

You may want to alter your output if you feel that the user wants to create some sort of "audit trail" type file. On the other hand, you would not want to alter your output if the user is attempting to diagnose problems by recording the data that is sent to the display.

Basic Displays

A basic display has, as a bare minimum, some sort of cursor positioning. It will generally also have "clear to end of line" operation (put blanks on the screen from the current cursor position to the end of the line that the cursor is on) and "clear to end of screen" (ditto, but to the end of the screen) functions. These functions can be simulated, if necessary, by sending spaces and newlines. A typical basic terminal is (was) the DEC VT52.

Such displays are quite usable at higher speeds (for example, over a 9600 bps connection) but usability deteriorates rapidly as the speed decreases. It requires patience to use basic displays over a 1200 bps connection, and a dedication bordering on insanity to use them at 300 bps.

Advanced Displays

Advanced displays have all of the features of the basic displays, along with editing features such as "insert" and "delete line and/or character." These features can significantly reduce the amount of data sent to the display for common operations. A typical advanced (*circa* 1980) terminal is the DEC VT100. Most terminals currently manufactured are at least as powerful as this one.

There is a subtle difference among some of the advanced terminals. An "insert line" operation adds one or more blank lines at the cursor: the lines that "drop off" the bottom of the screen are lost. A "delete line" operation deletes one or more lines at the cursor: blank lines are inserted at the bottom. A "scroll window" operation (move lines x through y up/down n lines) affects only the specified lines: the other ones remain stationary.

The "scroll window" operation is more pleasing than the others to see when there is some stationary text being displayed at the bottom of the screen. With "insert/delete line," the appropriate number of lines must be deleted and then inserted; the text at the bottom thus moves within the display's memory. Such jumps are often visible to the user. With "scroll window," the whole thing is performed as one operation and the lines at the bottom do not jump.

"Memory Mapped" Displays

This designation covers a wide range of displays. Their common characteristic is that display memory can be read or written at near-bus speeds. The display is usually built into the computer that is running the text editor. Many personal computers and workstations follow this design. But be warned: some computers have very fast display *hardware*, but the *software* that is used to interact with the display is very slow. It is probably better for a redisplay scheme to consider such displays to be "advanced" or even "basic." Examples of such displays are the ROM BIOS calls on the IBM PC and Sun workstations. In both cases, third-party drivers operate many times faster than the manufacturer-supplied ones.

The use of such fast displays has several implications for the redisplay process. First, many of the advanced features are typically not available. However, it may be possible to emulate the missing features quickly enough that the lack of advanced features is almost always not significant. Second, it may be possible to use the display memory as the only copy of the data on the screen. (This optimization is discussed in Chapter 7.) Third, if reading from the screen does not cause flicker but writing does, the screen can be read and the incremental redisplay process will run and compare the buffer against it, changing it only when necessary. Finally, if you can write to the screen without flicker, the redisplay process merely boils down to copying the buffer onto the screen, as copying is generally faster than comparing.

Graphics Displays

Most personal computer and workstation displays are actually bitmap-oriented graphics displays. Software is used to make them appear to display characters. With a graphics display – and the appropriate software – a program can not only display text, but display text using proportional spacing (where different letters take up different amounts of space), take advantage of different sizes, styles, and display fonts, and even incorporate graphical elements.

Keyboards

This section presents a review of salient keyboard features. Although most of us won't ever get the chance to design a keyboard, we all purchase keyboards, and more importantly we design programs with existing keyboards in mind.

The keyboard is the main way of telling the computer what to do. In some cases, it is the only way of doing so. Many thousands of characters will be entered in the course of a normal working session. Someone who types for a living (such as a typist, writer, or computer programmer) can easily type *ten million* characters each year.

The keyboard should thus be tailored for the ease of typing characters. While this statement might seem trite, there are a large number of keyboards on the market (*i.e.*, most) which are pretty poor for entering characters. Below is a discussion of the various keyboard features and why they are or are not desirable.

***N*-KEY ROLL-OVER** is a highly desirable feature. Having it means that you don't have to let go of one key before striking the next. The codes for the keys that you did strike will be sent out only once and in the proper order. (The *n* means that this roll-over operation will occur even though every key on the keyboard has been pressed before the first one is released.) The basic premise behind *n*-key roll-over is that you will not hit the same key twice in a row. Instead, you will hit a different key first and the reach for that key will naturally pull your finger off the initial one. *N*-key roll-over loosens the timing requirements regarding exactly when your finger has to come off the first key. Thus, typing errors are reduced. Note that *n*-key roll-over is of no help in typing double letters. Note also that shift keys are handled specially and are not subject to roll-over.

Some keyboards implement "2-key roll-over/*n*-key lockout." This means that only the first two keys of a continuous sequence will be sent and the rest ignored (until all keys are released). This "feature" is actually a way of turning the statement "we don't offer *n*-key roll-over" into a positive-sounding statement "we offer 2-key ..."

AUTO-REPEAT means that if a key is pressed and held down, the code for that key is sent repeatedly. It is a very desirable feature. It can cause problems (say, if you put something down on the keyboard), but such problems are worth living with. Older terminals sometimes followed typewriter design in that only certain keys would repeat (such as space, 'x', and dash). Repeating just these few keys is not useful. Other terminals repeat the printing characters but not the control characters. This is also not useful. As we will see later, it is the control characters that we are most likely to want to repeat.

There are three parameters associated with auto-repeat: the initial delay to the first repeat, the rate at which a key will repeat, and the acceleration of the repeat. Ideally, the user should be able to set these parameters. If they cannot be set, the values selected by the manufacturer become an additional consideration.

"**TYPEABILITY**" (I trust that the English language has not sunk to the point where this is considered to be a valid word) is the single most critical feature. It is simply the ability to type the useful characters without moving your fingers from the standard touch-typing position (the "asdf" and "jkl;" keys). As more and more people who use (computer) keyboards are touch typists and can thus type reasonably fast, they should not be slowed down by having to move their hands out of the basic position. It can take one or two *seconds* to locate and type an out-of-the-way key. The row above the digits is out of the way, as are numeric key pads and cursor control keys. One second is from three to ten characters of time (at 30 - 100 words per minute). Thus, it takes less time in general to type a four- or five-character command from the basic keyboard than to type one "special" key.

Because of the desire for typeability, it is worth at least considering doing away with such keys as Shift Lock or Caps Lock. They are rarely, if ever, used, and the keyboard space that they occupy is in high demand. (Yes, I realize that my anti-uppercase bias is showing here.)

Keyboard manufacturers have done other things that reduce typeability. Two examples are illustrative. First, the timing on the shift keys can be blown. The result of doing so is that when "Foo" is desired, "FOo," "fOo," and "foo" are as likely to result. The other example is having a small "sweet spot" on each key. Missing this "sweet spot" will cause both the desired and the adjoining key to fire or not. Thus, striking "i" could cause either "io" or nothing to be sent.

PACKAGING or physical keyboard design is also very important. Sharp edges near the keyboard or too tightly packed keys can cause errors and fatigue. Can the keyboard be positioned so as to be comfortable? Is there a palm ledge (this may be either good or bad)? Does the keyboard meet "ergonometric" standards? (In my experience, "ergonomic" standards equate to "hard to use.")

Special Function Keys

Keyboard manufacturers seem to have decided that a plethora of special keys is more useful than adding shift keys. Thus, you can get keyboards with Insert Line or "cursor up" or – gasp – PF1 (if not LF1, F1, *and* RF1). These keys, when pressed, will either do the function that they name, do something totally random, or send a (usually pre-defined and unchangeable) sequence of characters to the program.

With the advent of windowing systems, manufacturers have realized that the keyboard/display combination simply does not have the information required to properly perform the function locally. They have also decided that random operations don't sell devices well. This is actually a change from the terminals made a few years ago.

That leaves us with character sequences. Ideally, the sequences would be programmable. Thus, the editor could save the current set of programmed sequences (if any), load a set that would not interfere with any editing commands, then restore the user's settings upon exit. However, this is the real world and it is often the case that the sequences are not programmable.

Given this, the keys may or may not be useful. For example, the "cursor up" key might send Escape 'E'. You may wish this particular sequence to perform a "move to end of sentence" operation (I do). Thus, pressing the "cursor up" key will move you to the end of the sentence!

Okay, you say, I won't use Escape 'E' to move to the end of the sentence. You then look up all of the sequences that may be sent by function keys and design your command set around them. All is well and good until you try to use a different keyboard. Your new keyboard will in general use different sequences than the old one. The sequences may even conflict: for example, the "cursor down" key on the new terminal might send Escape 'E'.

We got into this situation for two reasons: a major one and a minor one. The minor one is easy to deal with. All that we have to do is tell the editor which keyboard we are using and have the editor perform any required adjustments. On UNIX systems, for example, the required information can be found in the */etc/termcap* or *terminfo* facilities.

The major reason why we are in this situation is that the program cannot tell when we are pressing a function key and when we are typing the same sequence of characters explicitly. After all, there are only 128 or 256 possible characters, and they must be shared by regular keys and function keys.

Some systems that support directly attached terminals use timing information to make this determination. If a string of characters comes in with no delays between them, they assume (usually correctly) that it is a single function-key press. This timing approach does not work if the terminal (or other computer) is coming in via a network.

The problem could best be solved by standardizing the character sequences sent by function keys so as to (1) have a single, obscure prefix (say, Escape, control-_) and (2) have a consistent syntax so that all devices can easily determine when the sequence is over. Command set designers would just have to live with the hole in the command set, but that would be a small price to pay.

Aside from the problems of compatibility with whatever software is being run, the placement of the function keys is also a problem. As was mentioned before, keys that are off to one side take a long time to hit. Thus, typing is slowed down considerably. The keys are best used for infrequently used functions or functions where the extra time is not a significant factor (*e.g.*, Help).

There is yet one more problem. Additional keys are not free and so the number of them that you'll want to pay for is limited. However, it is desirable to have the ability to specify a large number of functions (*i.e.*, have a large number of codes that can be specified by the user). The number of function keys required grows linearly with the number of codes.

Extra Shift Keys

The other way to increase the number of codes available to the user is to provide extra shift keys. Shift keys are keys that modify the actions of the other keys. Shift and Control are the two most common examples of such keys. The IBM PC has an Alt key, the Apple Macintosh has its "cloverleaf" key, and some terminals have a Meta key option.

As an example, a Meta key would set the top (value 128 decimal) bit of the character that is typed. Thus, while typing shift-A would send the code for uppercase A (65 decimal), meta-shift-A (often abbreviated as simply meta-A or ˜A) would send the code 128 + 65 or 193 decimal. A user can thus specify 256 codes instead of the usual 128 from a full ASCII keyboard.

The number of possible codes grows exponentially with the number of extra shift keys. Thus, 512, 1024, and even 2048 code keyboards (with 2, 3, or 4 extra shift keys) are conceivable. You will have to use system-dependent techniques to take advantage of this extra information.

Finding room on the basic keyboard for these extra shift keys is not easy. That is one reason why the removal of the Shift Lock key was suggested earlier. These keys must be on the basic keyboard in order to preserve touch-typeability.

Key Placement

A computer is not a typewriter. There are things that you do with a computer that simply do not apply to typewriters. Hence, a computer keyboard should have more keys than a typewriter, and yet these keys must be conveniently placed.

Several computer manufacturers have achieved good keyboard designs. Unfortunately, most of them have retired their good designs in favor of poor ones. (See the next section for examples.) Here are some of my criteria for good key placement:

- Basic QWERTY keyboard (Dvorak keyboards are discussed later)
- top row has: Escape, 1/!, 2/@, 3/#, 4/$, 5/%, 6/ˆ, 7/&, 8/*, 9/(, 0/), -/_, =/+
- second row has: Tab, QWERTYUIOP, [/{,]/}
- third row has: Control, ASDFGHJKL, ;/:, ’/”
- fourth row has: Shift, ZXCVBNM, ,/<, ./>, //?, Shift
- extra shift keys (Alt, Meta, etc.) should be immediately below the shift keys
- the Back Space and/or Delete keys should be on the upper right, as close in as possible
- the keys ‘/˜ and \/| should fit into the right somewhere
- the Return (Enter, etc.) key should be on the right, as close in as possible, in the third or second and third rows
- the Break key should be on the far upper right

These are the positions that have come to be accepted as standard for computer keyboards. However, some manufacturers have gotten scared that their computers might actually resemble computers. Thus, necessary keys such as Escape and Control get moved out to the far reaches of the keyboard, and ”<” and ”>” characters get moved from their convenient, traditional positions above ”,” and ”.” to who knows where.

Dvorak keyboards are an underground fad. Their proponents swear by them and claim significant performance improvements (*i.e.*, you can type faster on them). As the story goes, the standard QWERTY layout was designed to slow typing on the early typewriters in order to keep the mechanism from jamming. And, since jamming is no longer a consideration, one can (and Dvorak did) design a layout that is ”better.” Regardless of the truth of the story (and I believe it to be true), all keyboard layouts can take advantage of the improvements in technology. For example, modern keyboards are actually a grid of switches. These switches are scanned electronically. Their travel, feel, and other characteristics can be adjusted as desired. They have been adjusted so that both key travel and effort are much reduced from old, manual typewriters. Hence, hand and finger motions are reduced overall and the benefits to be gained from switching layouts are thereby reduced.

Considering that there are hundreds of millions of existing keyboards that use the QWERTY layout, and that there are billions of people trained to use it, it becomes clear that only an enormous gain in productivity (*e.g.*, greater than 100%) would be able to justify a switch to another layout. And while there are a number of isolated success stories, not even the proponents of Dvorak layouts offer any controlled studies that show the requisite gains (Norman 1990). Hence, these keyboards are not being adopted on a large scale.

Example Keyboards

This section will briefly review a number of widely available keyboards. The keyboards reviewed are the ones actually named: the review does not transfer to ”clones.” The comments are, of course, my personal opinions.

DEC VT100 terminal: The keyboard layout is excellent. The feel is klunky. The control keys don’t repeat.

DEC VT200 terminal: The keyboard layout is poor (badly placed Escape key and ”<” and ”>” keys). The feel is pretty good.

IBM PC 83-key keyboard: This is the one sold with the original IBM PC. Its layout is almost excellent (the ”\”/”|” key placement is a little weird). It makes a clacking sound which I happen to like although many people do not. The feel is excellent. If only they didn’t try to ”improve” it with...

IBM PC 101-key keyboard: This is the only one that you can get from IBM now, and it is enough in itself to keep me from buying a new IBM PC. The Escape and control keys are very poorly placed. The feel is excellent.

Apple Macintosh original "slab" keyboard: The layout isn't too bad you consider that Apple intended this machine to be its own universe, and not try to incorporate outside software. On the whole, however, it suffers from not having quite enough keys (especially Escape), so that terminal emulator programs are awkward to use. The feel is fair.

Apple Macintosh "Standard" keyboard: Perfect layout, enough keys, great feel.

Apple Macintosh "Enhanced" keyboard: This keyboard is for people who like the IBM PC 101-key keyboard. Enough said.

Sun Microsystems SPARCstation keyboard: Excellent layout, poor feel, too many function keys.

Graphical Input

Another way of interacting with a computer is by means of a graphical input device. The advantage of a graphical input device is that it can reduce the number of commands needed. Such a device is used for pointing at sections of the screen. It is possible to specify items (*i.e.*, "operate on that") without having to specify the numerical address of the location or a command string to move there.

When a graphical input device is used, the screen is treated as one menu with the device pointing to one entry. A cursor is used to provide feedback to the user about which menu "item" is currently selected. There are usually one or more flags that can be specified conveniently from the device. These flags provide control information and are analogous to shift keys.

The basic way to use these devices is to track the position implied by the graphical input device with the cursor. When a signal is given, the action implied by the current position is performed. The screen is logically broken up into two or more sections. One section has the text that is being edited. Moving the cursor here provides a convenient way to move the point around; typing a character could cause it to be inserted wherever the cursor is. Other portions of the screen can specify menus of possible actions to select from. Graphical input is thus a very sophisticated way of specifying a position as an argument to a function.

The following sections discuss the advantages and disadvantages of a variety of graphical input devices. Bear in mind that the comments are generalizations: there are exceptions to each of the advantages and disadvantages mentioned.

Touch Sensitive Display

A Touch Sensitive Display (TSD) is just what it sounds like. The screen is covered with a special transparent material (or a grid of LEDs and receptors or other devices) that you touch with your finger: the absolute (x,y) coordinates of where you touched are then reported. The only available flag is the "touch/no touch" flag. (Actually, experimental pressure-sensitive displays exist that report all three positions and three pressure axes.) The well-engineered touch-sensitive displays are quite pleasant to use for low-usage applications. For high-usage purposes such as text editing, it is tiresome to keep raising your hand to the screen, and your finger tends to cover the most interesting part of the display (*i.e.*, the part that you are about to edit).

Tablet

A tablet is a special surface that reports the position of the input device as an (x,y) coordinate. The input device can be a "puck" (a small box) or a special pen. At least one flag ("touch/no touch") is always available: some pucks have four, sixteen or even more extra flags. Tablets are very handy for converting paper documents such as maps into computer form. They are less useful for text editing, as they tend to be large and therefore require a long reach and a lot of uncluttered desk space.

Mouse

A mouse is a small box on wheels (or, in some cases, on felt pads over a special pad). As you move it around on the floor, desk, books, a leg, or most anything else, it reports the relative movement of the mouse (*i.e.*, "I was just moved n units up and m units left"). It can have several flags (buttons), although the correct number is one, as having extra buttons means that program designers will try to put extra functions on them. And, while the functions themselves are not a problem (I do advocate extra shift keys, after all), the presence of these functions usually implies a poor program design. Fortunately, if the mouse has extra buttons, the software can easily correct this "defect" just by making them all do the same thing.

Trackball

A trackball is an upside-down ("dead") mouse. Instead of moving the wheels by moving the box, you spin the slightly larger wheel directly.

Joystick

A joystick is a small stick mounted on a couple of potentiometers. They typically can report either absolute position, first derivative (relative movement) or second derivative (acceleration). As the stick is moved only over a small distance, it is difficult to construct one with good resolution and that avoids "stickiness" and "jumpiness." It is generally not as nice to use as a mouse or trackball. Flags are simulated by regular keyboard keys.

A Different Mouse

Finally, an imaginary but useful device should be considered. That device is a foot-operated mouse (perhaps called a "rat?"). Using your feet rather than your hand to operate the mouse solves one of the most nagging problems of any of these devices, which is that your hands must leave the keyboard with the usual, aforementioned results. Of course, this device makes it harder to edit with your feet up on your desk...

Other Devices

New types of input devices appear all the time. Thus, no listing of such devices can ever remain complete. An example of a recent such device is "pen" input. The points to remember are that each device should be judged on its own strengths and weaknesses and that the devices should be judged on how they help your users: not whether the devices are "neat" or "new."

Conclusion

These devices all assume a reasonably high bandwidth connection to the computer (say, 2400 bps or faster). If you have a slow-speed connection, the cursor tracking must be performed in the local display device, which must somehow be programmed with the knowledge of when to report events and what to do with the cursor (sometimes the cursor changes shape as it crosses from one part of the screen to another). In this way, it is possible to supply the necessary immediate feedback. A slow-speed connection would be quite satisfactory for communicating the significant events, but probably not satisfactory for the screen refresh that would follow, say, the selection of a menu.

Communications Path Issues

This section covers a number of miscellaneous issues concerning the communications path between the computer and the display/keyboard device.

Speed and Character Format

It almost goes without saying that the faster the communications path, the better. Consider it said.

It also almost goes without saying that a full-duplex communications path is necessary. Fortunately, we are long past the days when users were forced to wait until the computer let them type. Except on automated teller machines.

If the communications are over an asynchronous serial path, character format is an issue. The considerations are:

- Seven or eight data bits? Pick eight if possible. In this way, you at least have the potential to use extra key codes. The choice of eight data bits also lets you use international character sets.

- Even, odd, or no parity? Pick one to go with the data bits, as this field has no effect on the data that the editor sees. The combinations that you will tend to find all over the place are seven data bits and even parity (older systems) and eight data bits and no parity (newer systems).

- Number of stop bits? One, unless your system wants a different number.

Operating system designers make the quite valid and reasonable assumption that they should be doing some processing of the input characters. Fortunately, they usually also offer the ability to turn such processing off. A text editor should follow these steps:

On entry to the editor:

- Record the current processing parameters.

- Turn off all character processing.

On exit:

- Restore the saved processing parameters.

In this way, the text editor has complete control over what happens with the input characters. This places an extra burden on you as the writer of the editor, as you must replace the operating system handlers with versions of your own that mimic the existing functions. On the other hand, your versions will probably differ from the operating system versions in a number of crucial ways. For example, if the operating system lets you suspend your process (for example, under a Unix that supports job control), you need to restore the terminal and input processing parameters before you turn control back to the operating system. When resumed, you need to return the settings back to those used by the editor (noting any changes such as a new window size) and probably refresh the display. If you hadn't replaced the normal handlers, the user would find yours to be a very unfriendly program to use.

Flow Control

The faster the communications path, the less time the display has to process each character. As the speed of the communications path is increased, a point will be reached when the display can no longer keep up in real time. This is the point at which flow control is required. There are three methods currently in use to implement flow control.

The first is in-band control. Two characters are reserved for flow control purposes, typically the control-S and the control-Q characters. The first is used to mean "hold on, I can't keep up and my buffer is almost full." The second means "okay, I've caught up and you can proceed." This method works for the most part, but has the annoying property of using up two valuable control characters. Using any control characters causes problems for some programs. For example, there exist some communications protocols that use all 256 characters and allow no characters to be reserved.

The second method is out-of-band control. This method uses a variety of mechanisms, none of which interfere with sending data. Examples of such methods are hardware "handshake" lines and network protocol mechansims. This method is clearly superior to in-band.

The final method is flow control avoidance. This method takes advantage of the facts that displays take different amounts of time to process different characters and that some characters (called padding characters) take very little time to process. The program send the data as a mix of useful characters and padding characters. The specific mix is computed so that the average time required to process each character is less than the time taken to send a character over the communications path and that the terminal's input buffer does not overflow.

For example, let's say that we have these (fairly typical) figures:

- communications path speed is one character per msec
- time to process a printing character is .6 msec
- time to process a line feed character is 17 msec
- time to process a pad character is .1 msec

If we were just sending full lines of text to the display, we would send 81 characters in 81 msec. These 81 characters would take 80 * .6 msec + 17 msec = 65 msec to process. Hence, no padding would be required.

On the other hand, if we were just sending single-character lines of text to the display, we would send 2 characters in 2 msec. These 2 characters would take 1 * .6 msec + 17 msec = 17.6 msec to process. Padding would be required as the 17.6 msec processing time is greater than the 2 msec transmission time. As it turns out, 18 padding characters will be sufficient (1 * .6 msec + 17 msec + 18 * .1 msec = 19.4 msec, which is less than the 20 msec of transmission time). It is not difficult to calculate the correct number of padding characters required, given the character mix and the communications path speed.

This third method is the preferred method for text editors, as it works on any communications path (*i.e.*, even those with no out-of-band flow control) and it allows full use of all input characters. If used over a network, it has the disadvantage of creating a modest additional amount of network traffic.

The ideal method would be for the editor to determine whether out-of-band flow control is used along the entire communications path. If such control is in use, no padding characters need to be sent. Unfortunately, it is usually not possible to reliably determine the type of flow control in use.

Echo Negotiation

Echo negotiation was devised for the Multics computer system. It is a protocol for use by computer networks which can cut down on response time by reducing communications overhead. It is potentially useful in an environment where the user's terminal is at one node and the computer which is running the text editor is at another. In such an environment, it can take a long time to send a character back and forth, and yet it takes little more time to send many characters.

Echo negotiation requires that it be easy to describe exactly what is to be done with each character to a communications processor/terminal combination and that the combination be capable of doing enough of the editing to make it worthwhile.

Typically, echo negotiation can only be used when the editing point ("cursor") is at the end of a line. The text editor sends a list of approved characters to the terminal or other nearby processor. As long as the user types only those characters and does not reach the end of a screen line (thus necessitating a wrap), the terminal can safely echo the input characters to the display and hold onto the input text. When any non-approved character is typed (or the line fills up), the terminal reports all of the held input characters and the reason why the input was sent on (*i.e.*, non-approved character or line wrap) to the text editor. The editor then processes the input data and the cycle repeats.

The Xylogics Annex terminal server incorporates an advanced version of echo negotiation called the LEAP Protocol. It incorporates all of the above design.

Both standard echo negotiation and the LEAP protocol suffer from the same problem. This problem is severe enough to call into question the desirability of using them at all: Echo negotiation is only potentially useful when the terminal is separate from the computer that is running the text editor *and* when the computer is overloaded. The principle behind echo negotiation is that waking up the text editor process for each character is inefficient. In extreme cases, the wake-up may take so long that input echoing is significantly delayed. The fix that echo negotiation offers is to perform the updates in batches, thus waking up the text editor process fewer times and thereby reducing overhead.

The problem with the fix is inherent in its own success. With no echo negotiation, input is echoed slowly but evenly (user typing is in general much slower than process-switching times) and the text-editing process tends to stay in memory. With echo negotiation, input is echoed quickly until the non-approved character is typed, then a (comparatively) long pause is encountered while the text-editing process must be woken up and possibly even swapped in (we are talking about a situation where resources are tight, after all). Even though the *average* per-character processing time might be lower, the *variance* in per-character times is much larger with echo negotiation. It is usually the case that the variance is so high that the system as a whole becomes unpleasant if not impossible to use. In one extreme test that I performed, I found the variance of times to be so great that editing was all but impossible: until you stopped typing for many seconds, you could never tell whether the computer had processed all of your input and hence couldn't safely continue typing (editing commands – not new text to be inserted). In conclusion, echo negotiation is not a good feature to include.

Fancy Modems

High-speed modems (9600 bps and higher) are starting to become quite common. The main problem with them is that the advertising for them is focused around file-transfer protocols and dumping large quantities of text through them. The manufacturers add a variety of compression techniques to improve their modems' throughput in these areas.

However, text editing is interactive. Low response time is more important than high throughput. This is where the compression schemes implemented by the modems can cause problems. The simple solution is to turn off all such compression. Do not forget to turn off control-S / control-Q flow control while you're at it.

Questions to Probe Your Understanding

Devise at least three different ways of encoding cursor positioning coordinates. Which is the most extensible? (Easy)

Why can character-oriented displays handle blinking text more easily than graphics displays? Does it matter? (Easy)

If you could change one physical attribute of the display (*e.g.*, size, phosphor) that you use most, what would it be? (Easy)

Give an example of an application that can make effective use of function keys. (Easy)

Devise an efficient, extensible encoding scheme for function keys. (Easy)

Some keyboards (such as that used by the IBM PC and compatible computers) assign a priority to shift keys and only pay attention to the highest priority key pressed. For example, pressing both Control and Shift gives the same code as does just pressing Control. Is this better or worse than giving a different code to the combination key presses? Why? (Easy)

How does the amount of buffering affect the need for padding? Does it matter where in the system additional buffering is placed? (Medium)

A fourth way to handle flow control used to be common practice but is no longer. It is called "ETX / ACK" after the codes for the characters that were used to implement it. In this method, the sender sends a block of text followed by an ETX character. It then waited for the receiver to return an ACK character. Why has this scheme dropped from favor? How does it interact with terminals on computer networks? (Medium)

Chapter 3: Implementation Languages

He took his vorpal sword in hand:
  Long time the manxome foe he sought –

The choice of implementation language has a major effect on the design of a text editor. In some environments, only one language is available. In such environments, you do the best that you can and your editor may end up different from what it would be if the ideal language was available. However, most environments offer at least two languages. You thus have a choice, and this chapter offers guidance in making that choice. Of course, this may be a choice between Scylla and Charybdis...

General Considerations

The general considerations in selecting a language to use for implementing a text editor are:

- availability and implementation quality
- text handling power
- support for extensibility
- support for large projects
- efficiency

Each of these considerations will be explored in detail.

Availability and Implementation Quality

You can only use those languages that are supported on the system that your text editor is first implemented upon. Nonetheless, you should be thinking about the second, third, and later systems that your text editor will be ported to, and which languages all of those systems support in common.

In addition to the mere presence of a language processor on a system, you should take into consideration the quality of implementation of such systems. An implementation's speed of operation, quality of diagnostics, quality of code produced, and other such factors can make a large difference in the usability of the language on a particular system.

Text Handling Power

It may appear redundant to say that a text editor must handle text, but consider a spread sheet program: most of its work is in handling control flow, figuring redisplay, and setting up to execute commands. Only a small fraction of its time is spent in the floating point instructions that most users think is the program's "real work."

At any given moment, a text editor – or most any other similar interactive program – is mainly doing all of the following:

- waiting for user input
- parsing that input
- setting up to execute the commands
- executing the commands
- determining the effect of those commands on the screen
- updating the screen

Most of these operations involve processing text in some way or other. Text editors differ from other applications only in that the "executing the commands" item also involves manipulating text.

It is important to note that "text handling" does not necessarily mean "string handling." In many cases, the language's native string operations are not sufficient, and you must write your own string primitives. For example:

- Fortran does not support strings with dynamically varying lengths.
- C does not support strings that contain the NUL (0 decimal) character.
- Many implementations of Pascal do not support arbitrarily long strings (the leading byte count is often only 8 or 16 bits wide).

Support for Extensibility

If they do nothing else, text editors change. The language should make it easy to make and maintain changes. In some cases, the source code must be changed and the editor recompiled. However, it is very desirable to allow users to change some of the editor to suit their tastes. The language should offer such support.

This support can take many forms:

- late binding of names to procedures through indirect calls, dynamic linking, or other techniques
- retaining and using the symbol table information at run time so that the user can think of changes in terms of names, not addresses
- internal error and consistency checking under program control so that users can be protected from their mistakes
- the ability to add code to the executing editor

Not all languages offer these features. You will have to simulate the missing features when using those languages that lack them.

Large Project Support

Text editors are apt to grow quite large. All of the techniques useful for any large project are useful here. Examples of these techniques are:

- division of the program into separate modules
- division of the program into separate files
- separate compilation
- a way to organize the global name space
- a way to keep objects out of the global name space
- automatic verification of procedure call/declaration compatibility
- conditional compilation
- compilation constants
- a way of constructing "data abstractions" that package procedures and private state information
- a way of dynamically allocating memory

Add your own favorites to the list.

Efficiency

Programs spend most of their time doing simple operations such as:

```
A = B
A = B +/- 1
A = B +/- C
```

No other expressions occur often enough to matter (Knuth 1971). Thus, the language should support these common operations well. Control structure implementations – in particular, a procedure call – should be kept efficient. Most languages do all right in this respect: the main thing is to ensure that they keep simple things simple.

Specific Language Notes

This section briefly examines a number of popular and/or interesting language choices. It is important to keep in mind that at some level, all languages are equivalent: anything that you can do in one, you can do in any other, given sufficient CPU time, memory, and programmer elbow grease. However, each language is intended to make solving one type of problem easy, and in most cases that type of problem is not text editing.

TECO

TECO (Text Editor and COrrector) was developed at the Massachusetts Institute of Technology. It was one of the first text editors ever written. It grew over the years, gaining both popularity and features. During one of its more stable periods, Digital Equipment Corporation took a "snapshot" of its commands and produced (subset) versions for all of DEC's computers.

But TECO kept growing. Along the way, it turned into a Turing-complete programming language. Several sets of editor macros were developed and used. Sometime around 1975, Richard Stallman organized these Editor MACroS into the first Emacs-type text editor.

TECO is clearly a language capable of supporting text editing. However, unless you have a DECSYSTEM 20 computer to run it on, you're out of luck: M.I.T.'s version of TECO is written in assembly language and only runs on such systems.

The TECO command set is described in Appendix D. There are two reasons why it is not a good choice as an implementation language:

- As has been mentioned, its only implementation is on the PDP-10/DEC 20 series of computers. Implementations on other machines involve answering the question of what you write the TECO in – the very question that this chapter discusses.

- It is the only language less readable than APL. A listing of a TECO program has a more than passing resemblance to transmission line noise. Writing and maintaining TECO programs is a definite problem.

Lisp

Lisp – especially Common Lisp – is an excellent choice. It is readily extensible, as even compiled Lisp code usually has provisions for evaluating new expressions. It thus provides an interpretive language that can be readily used to write even complex editing macros. Modern implementations usually have excellent string support. The language has features such as macros and packages that support large projects well, and Lisp programs are fairly readable (if you don't mind lots of parentheses (like these (and these))). Compiled Lisp code is usually as efficient as that of any other language.

Its view of memory management makes it well suited to the linked line form of buffer management (described in Chapter 7).

C

C was designed by people who wrote operating systems and utilities. Since text editors are among those utilities, it is not surprising that C would be a good choice.

C supports extensibility as well as any other compiled language, and better than most. For example, it provides the ability to call procedures through a pointer.

C lacks a built-in string type, but this lack is not a hindrance, as you would probably need to re-implement strings anyway. There is a strong tradition in C of creating new data types, so the requirement is well supported.

C supports many of the features needed for large projects. In addition, as the language was designed by its users, and only came into widespread use after it was stable, there is a large existing base of compatible implementations. Due to this heritage, you don't need "improvements" in the language in order to get useful work done.

C's basic data types are focused around characters, integers, and pointers. These are exactly the core data types needed by text editors. C allows the ready manipulation of complicated data structures and yet remains generally readable.

C++ is a variant of C that provides much improved support for object-oriented programming. It, too, is a good choice.

PL/1

PL/1 is another example of a "systems language." Thus, most of the comments regarding C also apply to PL/1. However, its main failing is a lack of multiple implementations: the only vendor that seriously supports it is IBM Corp.

Other Systems Languages

There are a number of other systems languages (*e.g.*, Modula). However, like PL/1, they have only a limited availability. Many were designed as research projects. None of them even distantly approach C in the number of implementations or trained programmers.

Fortran

Well, some people think that it's a great language for writing astronomy programs. I have even written a text editor in it. Not by choice.

Pascal

Many people consider this language to be a good alternative (read "better") to C. It is worth reviewing Pascal's history: it was originally intended as a language to present (relatively small) algorithms in an academic setting. It was also targeted to introductory programming courses. For those purposes it is an excellent choice.

However, the standard language is not targeted towards developing large projects and does not provide the features that make developing a large project practical. On the other hand, each Pascal vendor has supplied those features. Unfortunately, they have in general chosen different ways to provide the features. Thus leading to incompatible implementations that make porting code difficult.

Basic

Basic has Pascal's problems, only more so: the core version is not even standardized (by the industry: there is an ANSI standard which is honored in the breach). Implementations range from "Tiny Basic," which can be run in only a few Kilobytes of memory to "True Basic," as defined by Kemeny & Kurtz (Kemeny 1985), which offers all the advanced features that you could want and all but omits line numbers. But "True Basic" bears little resemblance to what most programmers think of as the Basic language.

Ada

Ada was designed as a language to support embedded, real-time systems. It has many features which allow compilers to validate code and use external information to produce small, reliable object modules. However, these features do not mesh well with the need for extensiblity (for example, there is rarely a need to reprogram an altimeter while in flight). Further, the general computing environment that is the home for most text editors is simply outside the scope of what Ada is intended for. However, it should be seriously examined as a choice if the text editor is to execute in an embedded, real-time system.

Sine

Sine (Anderson 1979) was a Lisp-like language tailored for text applications. Its only implementation to date is on Interdata 7/32 (or Perkin-Elmer 3200) minicomputers running the MagicSix operating system developed at M.I.T.'s Architecture Machine Group. It is interesting because it is a language tailored for implementing editors. It is an example of an "ideal" implementation language.

Sine is composed of two parts. Sine source code is assembled into a compact format. This object code is then interpreted. It allows function rebinding and other such niceties. In addition, the interpreter implements such things as memory management and screen redisplay. Thus, the resulting editor is nicely structured, with "irrelevant" details hidden away. This mention of Sine leads nicely into...

Custom Editor Languages

No traditional language (except perhaps for Common Lisp) offers complete support for text editing. The solution, used by virtually every implementation of Emacs-type text editors, as well as many implementations of other editors, is the creation of a custom editor language.

An existing language – very often C – is selected. This language is used to write an interpreter for the custom editor language. The interpreter manages memory, handles display refresh, and in general provides all of the necessary utility functions. The editor language is then used to write the logic of all the user-visible commands.

As the editor language is implemented using an interpreter, the command set is readily extensible. Also, because the editor language is designed around text editing, it can offer excellent text-handling power.

The division of the programming tasks into two components provides an excellent base for supporting large projects. And, since the interpreter is usually implemented in a language such as C, the interpreter can be quite efficient.

For these reasons, custom editor languages are the preferred method for implementing text editors.

Questions to Probe Your Understanding

What is a good way of implementing a command dispatch table in C? Fortran? Pascal? Ada? (Easy)

Why is a string-oriented language such as SNOBOL not a good choice? (Easy)

How much compilation is appropriate for the custom editor language (none, just interpret the text; tokenization; full)? (Medium)

Following on the previous question, how would an opcode-oriented interpreter compare to a threaded-code interpreter? (Medium)

Chapter 4: Editing Models

So rested he by the Tumtum tree,
And stood a while in thought.

An editing model is the view of the file that the editor presents to the user. This chapter describes several editing models. You can build other models by varying and combining these models.

The following discussions review the models themselves, not the commands available to the user. You should assume that essentially the same commands are available in all models.

One-Dimensional Array of Bytes

The most general form of a data file is a one-dimensional array of bytes. The one-dimensional editing model presents this form of a data file directly to the user. In it, the bytes of the file are displayed uninterpreted for the user to see. The basic editing operations are "insert" and "delete bytes."

This model is very pure, but it is a little difficult for most users to deal with. Text editors that appear to use this model actually use a slightly modified form of the model where some characters – in particular, the tab and newline characters – are interpreted during the display process. Thus, text files appear as a series of lines.

In this model, line breaks may or may not require special handling. Whether they do depends on how they are represented. Various representations are described in detail in the next chapter.

This model supports both insertion and replacement editing equally well. Replacement editing is probably best implemented as a hybrid scheme where it automatically switches to insert mode to prevent replacing a line break.

Two-Dimensional Array of Bytes

This model is the basic two-dimensional form. Instead of editing a line, the user is editing in a quarter-plane, with the origin usually in the upper-left corner. Conceptually, the user can move freely in the two-dimensional quadrant. In practice, the editor usually only stores the non-blank portions, as storing an infinite-quadrant's worth of data can be prohibitively expensive. Some systems may impose fixed upper bounds on the width or length of the quadrant.

Line breaks are implicit in the editing model itself. Hence, implementations usually provide explicit commands to split and join lines.

Both insertion and replacement editing are possible, although the model lends itself to replacement editing in a natural manner. Editors that use this model often have explicit commands to insert (and delete) both rows and characters within a row.

While the pure form of this model arranges the text into a rectangle, most implementations actually impart a left-to-right, then top-to-bottom (or one of the other seven combinations) bias. This bias affects all of the editing operations. For example, it is often the case that implementations offer many commands for editing within a line, but only a few commands for editing entire lines.

List of Lines

This model is halfway between the first two. It consists of a one-dimensional array of lines. Each line is then a one-dimensional array of bytes. From the user's viewpoint, this model differs from the two-dimensional model in that text exists only where it has been entered. If the user wants to extend a line to the right, he or she must go into insert mode and type space characters.

In the two-dimensional model, on the other hand, the quadrant is assumed to be filled with blanks. Hence, there is no concept of extending the line to the right, as the line is assumed to extend infinitely far. To add text to the right, the user simply moves to the desired position.

Implementations that use this model usually make a very sharp distinction between editing within a line and editing lines. For example, lines may have a maximum length or cut and paste operations may only operate on line boundaries.

Paged Models

It once was popular to divide the text into a series of pages. Editing was performed within a page, and explicit commands were required to move to another page or to re-paginate the text. Any of the models could be used for editing within a page. This division was thought to be natural: not coincidentally, it just "happened" to make it easier to write editors on systems that had very limited amounts of memory.

Most modern editors show page breaks as a "framework" that "floats" over the underlying text. This framework can be placed over any of the other underlying editing models.

Objects

Editing is a very general concept: there is no reason to limit the basic objects being edited to characters (or bytes) and lines. It may make sense in some cases to provide ways of editing such objects as words, sentences, paragraphs, sections, chapters, and other "natural" objects as explicit objects. Most editors provide commands to manipulate these objects without having them affect the fundamental editing model.

Other objects cannot be readily simulated. Examples of these objects are links to other documents, "opaque" objects included from other objects (*e.g.*, bitmaps), and graphical objects (lines, boxes, circles, etc.).

In addition, text can be viewed in more than two dimensions. For example, multiple files can be "stacked" into a third dimension, multiple versions of a single file can be combined into a time-like dimension, or portions of a file can be viewed and manipulated as a tree or list structure. The possibilities are endless.

Dealing with Real Text

The models just listed are more or less pure forms. Each model has its advantages and disadvantages because text has a more complex structure than is represented by any of the models.

On the one hand, text is composed of a hierarchy of lexical units:

- characters
- words
- phrases
- sentences
- paragraphs
- subsections
- sections

- chapters
- documents

These units reflect the *meaning* of the text. When the user is thinking in terms of meaning, the editor should provide an editing model – and commands – that reflect these units. Since text is read sequentially, the one-dimensional model is well-matched to this mode.

On the other hand, the printed page is composed of:

- characters, which are arranged into
- words, which are arranged into
- lines, which are arranged into
- pages, which are arranged into
- documents

These units reflect the *layout* of the text. When the user is thinking in terms of appearance, the editor should provide an editing model – and commands – that reflect these units. As a page is a two-dimensional object, the two-dimensional model fits this mode well.

Many "simple" editors and word processors support this mode of thinking. This mode is attractive for new users. After all, isn't the whole purpose of a word processor to put characters on a page? So doesn't it follow that users should be thinking in terms of placing each character on the page, one after the other? If taken to extreme, the user is forced to make every placement decision, a situation that doesn't leave the user with much time or energy left to decide what to write.

While layout is important, it does not directly relate to the meaning of the text. And while meaning is important, the user sees the text in a particular layout, so layout-oriented editing is also important. The challenge, then, is to design an editing model – and an editor – that allows the user to select the most appropriate features of each model with minimal effort. Thus, it can take advantage of the best of both models while avoiding the disadvantages.

Questions to Probe Your Understanding

Explore the ramifications of a two-dimensional editing model where the origin is in the center of the document instead of the upper-left corner. What additional commands might be required? What operations (if any) does such a model make easier? Harder? (Easy)

Provide an algorithm for transforming between the one-dimensional and two-dimensional models. (Medium)

What is a good way to support proportionally spaced text in the *pure* two-dimensional array of bytes model? (Hard)

What problems are encountered when trying to support more than one model at the same time? (Easy) What is a good solution to these problems? (Hard)

Chapter 5: File Formats

And, as in uffish thought he stood,
  The Jabberwock, with eyes of flame,

This chapter surveys the range of file formats that a text editor might encounter.

Text Files

Each operating system has a standard way of storing text files. Text editors must be able to edit these standard system text files. From the user's point of view, such files consist of a series of reasonable-length lines of "reasonable" characters.

Line Boundaries

From the program's point of view, system text files consist of a sequence of characters, divided into lines in a variety of ways. Each of the most popular methods will be described.

Card and Print Images: These files are a series of lines, all exactly the same length (typically 80, 132, or 133 characters long). They may also include another form of line divisor (*e.g.*, 80 characters, then a CR/LF sequence). These files will mostly be found on older systems.

Newline Character: Marker bytes are used to signal the end of one line and the start of another. Popular choices are:

- Line Feed: Used by UNIX systems.
- Carriage Return: Used by Apple computers and some DEC computers.
- Carriage Return/Line Feed combination: Used by CP/M and MS/DOS computers. The two-character sequence can be awkward to use when editing. You can usually get away with dropping the CR (but only when it appears as part of a CR/LF sequence), use the LF as a newline character, and put the CR back when the file is written out. When editing an existing file, record whether you found CRs to remove, so that you don't put extra CRs in when writing out binary files.

Character Count: Some systems use an initial count of characters (typically the count is one or two bytes long), followed by that many characters. There may or may not be padding between lines in order to align their start on a word boundary.

Record Markers: Some operating systems store one line per record, and store the record markers "out of band." In this case, you must read and write one line at a time, and record the line break information somehow. (If the operating system lets you read multiple lines at once, it must have some method of indicating what the line boundaries are, which leads us to one of the earlier methods.)

Line Contents

Some systems place restrictions on the contents of each line. The most frequently encountered restrictions are:

Long Lines: Some systems have no limit on the length of a line. Others place a fixed limit. Typical limits are 80, 127 or 128, 255/6, 511/2, 32,767/8, and 65,535 characters. If a program attempts to write lines that exceed the system limit, some systems return an error, others split the line, and still others will silently truncate the line.

Short Lines: Most systems support zero-length (empty) lines quite well. However, some systems do not allow such lines while others allow them in theory but not in practice. For example, the system-supplied text editor may not allow the entry of empty lines. Because of this limitation, no files will be created that have such lines. Hence, the code to handle such lines may not be tested well, and some programs may not behave properly when such lines are encountered.

Partial Last Line: This problem can only occur in systems that use a newline character. As with short lines, the system-supplied text editor may not allow the entry of partial last lines (*i.e.*, a missing newline character) and some programs may not behave properly when such lines are encountered.

Non-Printing Characters: All systems generally allow all printing characters and the Space character to appear in text files. These character have codes that range from 32 to 126 decimal in ASCII or the equivalent characters in EBCDIC. Difficulties arise in how programs handle other characters. For example, are Tab characters treated as one character or the appropriate number of spaces, and if the latter, what is the appropriate width? Limitations on non-printing characters usually fall into the following groups:

- Tab
- Form Feed
- Back Space
- Carriage Return (this is a "bare CR")
- other control characters (in the range 0 to 31 decimal and 127 decimal)

- meta characters (128 to 255 decimal)

Typically, systems that allow a given group allow all preceding groups. Given that the characters are allowed, the next question is how should the character be displayed. Typical methods are:

- Just send the character as is without translation.
- Expand into caret notation (see Appendix E).
- Expand into octal or hexadecimal notation.

End of File

Most systems record the exact file length and make this information available to the program. However, there are two special cases to be considered:

CP/M systems only record the file length to the next multiple of 128 bytes. By convention, a control-Z (^Z) character is used to mark the end of the file. Data after the first Z character is ignored. Note that if the file ends exactly on a 128-byte boundary, some programs do not add the trailing ^Z character. Some programs filled the entire remainder of the block with the ^Z character: other programs relied on this convention and only removed trailing ^Z characters.

MS/DOS systems started off following the CP/M convention but later changed to omit the ^Z character. The safest algorithm to use on these systems is:

- If you are editing an existing file, record whether the file originally ended with ^Z. When the new version is written, add a ^Z if the file had one.
- Otherwise, do not add a ^Z.

As always, the user should have a way of selecting both methods.

Binary Files

From a text editor's point of view, a *binary file* is any file that is not a text file. These files have none of the following restrictions found in text files:

- Files may not be divided into lines at all.
- Lines may be any length.
- Lines may contain any character.

As a general rule, it is a nice feature to be able to edit a binary file. The rules to be followed are these:

- You should be able to read a file in and write the identical file back out.
- It should be possible to move to and usefully view any portion of the file.
- It should be possible to insert any character.
- It should be possible to precisely control any deletions (*e.g.*, "delete the following three characters").

Structured Files

If your editor will only encounter standard system-text files and binary files, you can skip the rest of this chapter which describes considerations for designing file formats for holding information in addition to pure ASCII text.

Basic text files use 94 printing characters and a Space character. They also need some way to indicate line breaks. Often, users will want to include the Bell, Back Space, Tab, and Form Feed characters in their text files. Thus, a total of 99 characters are reserved for representing themselves. This leaves 29 codes (with 7 bit characters) or 157 codes (with 8 bit characters) available for other uses.

If all computer manufacturers used only the ASCII character set, the analysis could stop here. However, IBM Corp., Apple Computer Corp., Hewlett-Packard, and other vendors all support "extended" character sets that make use of many of these other codes. (Not to worry, though, the world is still safe: all of the manufacturers support *different* extended character sets.) What were previously unused codes are now in use to the extent that your users wish to be able to make use of the extended characters.

(Actually, as this book is being written, many vendors are jointly developing a 16-bit character set that is intended to encompass most characters and glyphs in use, although not doing a complete job on Chinese, Japanese, and Korean.)

Where to Store the "Extra" Information

Whether or not "extended" character sets are supported, it is likely that you will want to store more information than can fit into the unused character codes. This leads us to the basic choice that will affect many other aspects of the implementation: is the extra information stored in-band or out-of-band?

In-Band

Storing information *in-band* means that some of the character codes are used to signal the presence of this additional information. Once the presence of this information is indicated, all character codes can potentially be used to represent the information.

The use of these character codes for non-character purposes has two ramifications. First, those codes are not available for representing characters. Second, if those characters are present, redisplay must know how to display them (and their associated information) and the user commands must know how to process them.

Depending upon the purpose of the extra information and your users' expectations, it may be appropriate to allow this in-band information to be visible to the user, at least in some display modes. Further, it may be appropriate to allow the user to edit the information directly. On the other hand, the best choice might be to hide this information from the user and allow only indirect manipulation.

Note that programs should be able to parse in-band information in either direction (*i.e.*, both when working forwards through the buffer and when working backwards). It is also important in this representation that it be reasonably easy to determine how to display a file when starting from an arbitrary point in the middle of the file. In particular, the program shouldn't have to examine the entire previous contents of the file in order to figure out how to display something.

Out-of-Band

Storing information *out-of-band* means that none of the character codes are used for any special purpose. Rather, the additional information is stored somewhere else and is tied back to the text by means of pointers and offsets.

The disadvantage of choosing the out-of-band method is that you must find some place to put the information. While a file is being edited, the information can (and probably should) be stored in special purpose structures within the editor. However, when the file is stored, the additional information must be put somewhere. This place can be a separate file or a separate part of the same file (either a different file "fork," or at the file's beginning or end).

Conclusion

There may be enough additional information that manipulating it can itself require significant overhead. The techniques described in the next chapter can apply to the additional information as to the text itself.

There is no preferred choice: both in-band and out-of-band have their good points and their bad. The choice must be made on a case-by-case basis.

Actually, they are almost two ends of a continuous scale. The difference between them could also be considered like this:

- *in-band* data is parsed at each use
- *out-of-band* data is parsed at file load

The Additional Information

This section describes some of the categories of additional information that you may wish to store in files. These categories are illustrative examples only: you will probably want to store other types of information or the same types in different ways.

Fonts, Sizes, Attributes

A *font* describes the shapes of the characters. *Size* information describes how large a character should be. *Attributes* are variations on a font such as boldface, italics, or underscoring. Together, these are used in word processors to provide character formatting.

These three share common qualities such as the ability to change at a character boundary, and the ability to change one without changing the others. The representation that you select needs to take these qualities into account.

Line, Paragraph, Page, and Other Formats

This information determines such things as line margins, justification types, tab stops, page headings and footings, page length, and so forth. This information has a major effect on the redisplay code described in Chapter 7.

Non-Text Objects

These can be arbitrary non-text objects such as graphical bitmaps or object, spread sheets, database excerpts, or other information used by non-editor applications. The editor needs to know such things as:

- how to display them
- how much space they occupy
- how to invoke the application that defines them
- how to obtain current or updated versions of them

Internationalization

This section lists some of the U.S. and English language biases that might be encountered in text files. Techniques for removing these biases from the program are outside the scope of this book. By their nature, these biases are hard to sort out: my apologies if I have missed some.

Except for this section, this book contains U.S. and English language biases. However, the programming and design techniques described in the rest of this book are applied in pretty much the same way in non-U.S. and non-English language editors.

The first bias is the character set used to represent information. There are many different international character sets and, while they tend to incorporate the U.S. ASCII character set (presented in Appendix E) in them, they all differ in the other characters.

The second bias is the character size (*i.e.*, the number of bits required to represent the number of distinct characters that can be stored in the document). If you are limiting your users to ASCII, 7-bit characters are sufficient. However, international character sets may require 8, 16, or even 32 bits per character. In the case of the larger character sizes, it may make sense to store most characters as 8-bit codes and to have multiple-byte characters for the others. So long as your implementation handles them consistently and can interchange data with the other programs on your system, the exact representation does not matter.

The third bias is the language direction. English uses left-to-right, then top-to-bottom. Other languages use different patterns. You must also properly handle cases where you mix languages (say, English and Arabic).

The fourth bias is the general conventions for handling such things as character case (some languages do not have English's upper/lowercase distinction), characters changing representation depending upon their position within a word (contextual forms), and so forth.

The fifth bias is in handling numbers. For example, in the U.S., numbers are written as "1,000.5". In Europe, they are written as "1.000,5". In addition, languages differ in the order that digits are entered (left to right vs. right to left) and the placement of the most significant digit.

The sixth bias is in handling dates: day - month - year, month - day - year, and year - month - day are all popular, as are differing punctuation characters between them.

The seventh bias is in handling calendars. Gregorian and Julian are both in use and quite similar, but there are lunar and other calendars also in use.

The eigth bias is how punctuation characters are handled. For example, in Spanish, questions are introduced with an inverted "?" character and terminated with "?".

The last bias is how hyphenation is handled. In English, it is often difficult or impossible to determine how a word should be hyphenated. In Portuguese, for example, it is very easy to determine how to hyphenate a word and is considered mandatory to handle hyphenation properly.

Questions to Probe Your Understanding

How visible should the representation of line boundaries in standard system-text files be to the user? (Easy)

Why is the ability to edit binary files useful? (Easy)

Is it reasonable to require that font, size, and attribute definitions always be properly nested? (Medium: note that the program can automatically make non-nested change requests into nested ones)

Define a representation for fonts, sizes, and attributes. (Medium)

Define a *good* representation for fonts, sizes, and attributes. (Hard)

Identify a bias that I missed. (Easy for non-U.S. readers, probably Hard for U.S. readers)

Chapter 6: The Internal Sub-Editor

Came whiffling through the tulgey wood,
  And burbled as it came!

There are many ways to decompose the implementation of a text editor into smaller pieces. This book analyzes one particular decomposition: that into a sub-editor to manage the text being edited, redisplay, and the user-oriented commands. (There are no other pieces: when these have been assembled, the editor is complete.) This particular decomposition was chosen for two reasons. First, it is a natural one with relatively simple interfaces between the parts. Second, it has been chosen for many different implementations: it is thus known to be a decomposition that works well. This chapter covers the internal sub-editor. The following chapters describe the other parts.

The purpose of the internal sub-editor is to hide all of the details of how the text is stored from the redisplay and the user-oriented commands. This chapter will begin by presenting some basic concepts and definitions. It will then list internal data needs and describe a procedural interface. Finally, it will present a number of ways to implement the actual sub-editor and discuss the trade-offs between them.

Basic Concepts and Definitions

A *buffer* is the basic unit of text being edited. It can be any size, from zero characters to the largest item that can be manipulated on the computer system. This limit on size is usually set by such factors as address space, amount of real and/or virtual memory, and mass storage capacity. A buffer can exist by itself, or it can be associated with at most one file. When associated with a file, the buffer is a copy of the contents of the file at a specific time. A file, on the other hand, can be associated with any number of buffers, each one being a copy of that file's contents at the same or at different times.

A *write* operation replaces the contents of a file with the contents of a buffer. Thus, the two have identical contents for at least one moment. A *read* operation replaces the buffer with the contents of a file. Again, the two have identical contents for at least one moment. An *insert* operation adds the contents of the file to the buffer: the two will not have identical contents unless the buffer was empty just before the insertion.

The buffer interface presented here follows the "one-dimensional array of characters" model as described in Chapter 5. As will be seen, however, the implementation need not follow that model. When stored as text in the buffer, line breaks will be represented by a single character, called *newline*.

At any one time there is one and only one special position within the buffer. This position is called the *point*. All operations that change the contents of the buffer occur at this position. There are also operations whose sole purpose is to move the point. The point can exist only *between* two characters: it is never on a character. On those displays that can only position a cursor on a character, the convention is to place the point at the left edge of the cursor.

The start of the buffer (which corresponds to the first location in the file) is considered *before* or *backward from* the point. The end of the buffer is considered to be *after* or *forward from* the point.

From time to time, it is useful to be able to remember positions within a buffer. A *mark* is an object that can remember a position. There can be any number of marks within a buffer, and more than one mark can remember the same position. As with the point, a mark is always located between two characters.

When there is exactly one mark, the range of characters between the point and the mark is called the *region*. It does not matter which order the point and mark are in.

Here are two examples of how marks are used:

- A mark may remember a specific location for future reference. For example, a command might paginate a file. In this case, a mark would remember where the point was when the command was invoked. Thus, the point could be moved during the re-pagination and returned to its initial starting place.

- A mark can serve as bounds for iteration. For example, the "fill paragraph" command might place a mark at its starting place, move to the end of the paragraph and place a mark there, then move to the beginning of the paragraph. It then performs a "fill region" operation, filling from the point to the location of the second mark.

There are two types of marks. They differ only in how they behave in the case that an insertion is made at the location of the mark. *Normal marks* move with the insertion. Thus, the newly inserted character will be just before the mark. *Fixed marks* remain in place: the newly inserted character will be just after the mark. An example of the difference is in the case where a command is to identify the characters that are inserted. The command merely needs to create both a fixed and a normal mark at the same place. After the insertion, the two marks will bracket the new characters.

A *mode* is a set of alterations to the user-oriented command set. For example, "C mode" might alter the definitions of the word-, sentence-, and paragraph-oriented commands to apply to tokens, language statements, and block structure. Modes are described in more detail in Chapter 8.

Finally, the term *character* denotes the basic unit of change within a buffer. While characters can be any size, they are most often eight bits long. In such a case, the term *byte* may be used interchangeably with *character*.

Internal Data Structures

This section discusses the sub-editor's data structures. All of the sub-editor's state information is defined in this chapter. Thus, if your implementation retains this information across invocations, you can offer the user the ability to resume editing where he or she left off, thus reducing the amount of work required to edit a file.

The other place where state information is kept is in the screen manager which is described in the following chapter. The screen manager is that part of the software which knows what was displayed on the user's screen. If that knowledge is not retained across invocations, the information displayed on the user's screen may change when the user exits and re-enters the editor. If that knowledge is retained, the information can be recreated exactly.

All of the specifics of the data structures listed here are examples only. You will undoubtedly wish to change some or all of them.

The *world* contains all of the buffers in use by the editor. It is a circular list of buffer descriptors and a variable that indicates which is the current buffer. In C syntax:

```
struct {
        struct buffer *buffer_chain;
        struct buffer *current_buffer;
        } world;
```

Each buffer descriptor has this internal information:

```
struct buffer {
        struct buffer *next_chain_entry;
        char buffer_name[BUFFERNAMEMAX];

        location point;
        int cur_line;
        int num_chars;
        int num_lines;

        struct mark *mark_list;

        struct storage *contents;
```

```
char file_name[FILENAMEMAX];
time file_time;
FLAG is_modified;

struct mode *mode_list;
};
```

Next_chain_entry is the mechanism used for implementing the circular list of buffers. The list is circular because there is no preferred (or "origin") buffer and it should be possible to get to any buffer with equal ease.

Buffer_name is a character string that allows the user to be able to refer to the buffer.

Point is the current location where editing operations are taking place. It is defined in terms of a private data type, since different implementations will use different representations. As it turns out, there is never a need for any code outside of the sub-editor to ever be aware of the representation of this data type.

Cur_line is optional. If implemented, it provides a high-speed way to track the current line number.

Num_chars is optional. If implemented, it provides a high-speed way to track the total number of characters in the buffer (its length).

Num_lines is optional. If implemented, it provides a high-speed way to track the total number of lines in the buffer.

Mark_list is the list of marks defined for this buffer. The mark structure is defined later.

Contents indicates the actual buffer contents. As with the location data type, its specifics will vary with the implementation.

File_name is the name of the file associated with the buffer, or the empty string if there is no associated file.

File_time is the last time at which the contents of the file and buffer were identical (*i.e.*, the time of the last read or write). On multi-process systems, this value can be used to determine whether the contents of the file have been changed by another process, and thus whether the copy being edited is in synchronization with the actual file.

Is_modified indicates whether the buffer has been modified since it was last written out or read in.

Mode_list is the list of modes in effect for the buffer. The mode structure is defined next.

```
struct mark {
        struct mark *next_mark;
        mark_name name;
        location where_it_is;
        FLAG is_fixed;
        };
```

This structure is a linked list and is repeated for every mark. The chain is not circular. It probably is a good idea to keep the list sorted in the order that the marks appear in the buffer.

Next_mark is a pointer to the next mark in the chain. A NULL pointer indicates the end of the chain.

Name is the name of the mark. This name is returned by the mark creation routine and provides a way for the user to refer to specific marks. If your implementation permits, you can just return a pointer to the mark structure instead of making up names.

Where_it_is is the mark's location.

Is_fixed indicates whether the mark is a fixed mark.

```
struct mode {
        struct mode *next_mode;
        char *mode_name;
        status (*add_proc)();
        };
```

This structure is a linked list and is repeated for every mode that is in effect for the current buffer. The chain is not circular. While modes should be defined in such a way that it does not matter what order they are invoked in, it is probably not possible to meet this requirement in actual practice. Thus, this list must be kept sorted in invocation order. Modes are discussed in more detail in Chapter 8.

Next_mode is a pointer to the next mode in the chain. A NULL pointer indicates the end of the chain.

Mode_name, if non-NULL, is the name added to the list of names of modes in effect. This list is ordinarily displayed somewhere on the screen. Note that there should be a mechanism for defining modes that do not have displayed names.

Add_proc is a pointer to a procedure to execute whenever the command set for this buffer needs to be created or re-created. The procedure should make all required modifications to the global command tables and return a success/fail status.

Procedure Interface Definitions

This section defines the interface provided by the sub-editor. The procedures will be described in terms of their logical function only, leaving out specific implementation details. An example of such a detail is a method of determining whether the operation succeeded. (The undefined type "status" will be used to indicate places where status information is especially desirable.) All data types mentioned (*e.g.*, string) are intended to be generic, and no specific implementations are assumed.

One question that is important to your implementation but not addressed in this definition is whether the caller or callee allocates the data structures. This chapter will assume that the callee allocates all data.

The names are selected for their mnemonic value. Actual implementations may be forced to change them to conform to local limits. In addition, you may wish to add a unique prefix (such as "Buf" or "SE") to them all to prevent name conflicts.

```
status World_Init(void);
status World_Fini(void);
status World_Save(char *file_name);
status World_Load(char *file_name);
```

World_Init is the basic set-up-housekeeping call. It is called once, upon editor invocation. It should perform all required one-time initialization operations. No other sub-editor procedure except for **World_Fini** can be legally called unless **World_Init** returns a successful status. After this call, one empty buffer exists (perhaps called "scratch" or something similar).

World_Fini terminates all sub-editor state information. Once called, **World_Init** must be called again before other sub-editor calls can be legally made.

World_Save saves all editor state information in the specified file.

World_Load loads all editor state information from the specified file. These two routines implement state-saving across editor invocations. The possibility of retaining multiple saved environments is interesting but, while it has been implemented, is not a feature that receives much use. It is perhaps too difficult for users to keep track of multiple editing environments or users may prefer to be able to switch among tasks without having to perform a save and load.

If you are creating a "stripped down" editor, the **World_Save** and **World_Load** routines would not do anything. They can be put in as stubs if there is a reasonable possibility that the editor will be embellished later.

```
status Buffer_Create(char *buffer_name);
status Buffer_Clear(char *buffer_name);
status Buffer_Delete(char *buffer_name);
status Buffer_Set_Current(char *buffer_name)
char *Buffer_Set_Next(void);
```

```
status Buffer_Set_Name(char *buffer_name);
char *Buffer_Get_Name(void);
```

These routines all manipulate buffer objects. Their definitions assume that character string names are used to specify buffer objects. As with marks, pointers to buffer structures can also be used if your implementation permits.

As with many other questions, it is an implementation choice as to whether the sub-editor retains a "current buffer" or whether a buffer is explicitly provided to all remaining sub-editor calls. This definition chooses the former; as buffer changes are performed only (comparatively) rarely, and hence it is probably helpful to the sub-editor to be able to cache the information relating to the current buffer.

Note that most of these calls are not useful for single buffer implementations as might be found in very resource-limited environments (*e.g.*, a toaster). Such implementations should only include the calls if there is a reasonable chance of expanding to a multiple buffer editor in the future.

Buffer_Create takes a name and creates an empty buffer with that name. Note that no two buffers may have the same name.

Buffer_Clear removes all characters and marks from the specified buffer.

Buffer_Delete deletes the specified buffer. If the specified buffer is the current one, the next buffer in the chain becomes the current one. If no buffers are left, the initial "scratch" buffer is automatically re-created.

Buffer_Set_Current sets the current buffer to the one specified.

Buffer_Set_Next sets the current buffer to the next one in the chain, and it returns the name of the new buffer. This mechanism allows for iterating through all buffers looking for one which meets an arbitrary test.

Buffer_Set_Name changes the name of the current buffer to that specified.

Buffer_Get_Name returns the name of the current buffer.

```
status Point_Set(location loc);
status Point_Move(int count);
location Point_Get(void);
int Point_Get_Line(void);
location Buffer_Start(void);
location Buffer_End(void);
```

Point_Set sets the point to the specified location.

Point_Move moves the point forward (if *count* is positive) or backward (if negative) by *abs(count)* characters.

Point_Get returns the current location.

Point_Get_Line returns the number of the line that the point is on. Note that while characters are numbered starting from zero, lines are numbered starting from one.

Buffer_Start returns the location of the start of the buffer.

Buffer_End returns the location of the end of the buffer.

```
int Compare_Locations(location loc1, location loc2);
int Location_To_Count(location loc);
location Count_To_Location(int count);
```

These are miscellaneous utility routines.

Compare_Location returns 1 if location *loc1* is after *loc2*, 0 if they are the same location, or -1 if *loc1* is before *loc2*.

Location_To_Count accepts a location and returns the number of characters between the location and the beginning of the buffer. The point's percentage position can be computed by:

```
((float)Location_To_Count(Point_Get()) * 100.) / ((float)Get_Num_Chars())
```

Count_To_Location accepts a non-negative count and converts it to the corresponding location. You can set the point to a position specified by an absolute character count by:

```
Point_Set(Count_To_Location(count));
```

```
status Mark_Create(mark_name *name, FLAG is_fixed);
void Mark_Delete(mark_name name);
status Mark_To_Point(mark_name name);
status Point_To_Mark(mark_name name);
location Mark_Get(mark_name name);
status Mark_Set(mark_name name, location loc);
FLAG Is_Point_At_Mark(mark_name name);
FLAG Is_Point_Before_Mark(mark_name name);
FLAG Is_Point_After_Mark(mark_name name);
status Swap_Point_And_Mark(mark_name name);
```

These routines manage marks. They allow for creating both normal and fixed marks, deleting marks, and otherwise manipulating them. Except when creating them, there is no difference in usage with these routines between normal marks and fixed marks (although their behavior will differ).

Mark_Create creates a new mark of the specified type and returns its name. The new mark is positioned at the point.

Mark_Delete deletes the specified mark.

Mark_To_Point sets the location of the specified mark to the point.

Point_To_Mark sets the point to the location of the specified mark.

Mark_Get returns the location for the mark. This is not actually used all that much, as the location value can change whenever any sub-editor call is made.

Mark_Set moves the specified mark to the specified location.

Is_Point_At_Mark returns True if the point is at the specified mark.

Is_Point_Before_Mark returns True if the point is before the specified mark.

Is_Point_After_Mark returns True if the point is after the specified mark.

Swap_Point_And_Mark swaps the locations of the point and the specified mark.

With these definitions, the basic way of doing something over a region would be like this:

```
status Do_Something_Over_Region(mark_name name)
        {
        FLAG was_before = Is_Point_Before_Mark(name);
        mark_name saved;
        status stat = OK;

                /* ensure that the point is before the mark */
        if (!was_before) Swap_Point_And_Mark(name);

                /* remember where we started */
        if (Mark_Create(&saved) != OK) {
                if (!was_before) Swap_Point_And_Mark(name);
                return(NOT_OK);
                }

                /* loop until you get to the mark */
        for ( ; !Is_Point_At_Mark(name); Point_Move(1)) {
                if (<do something> != OK) {
                        stat = NOT_OK;
                        break;
                        }
                }

                /* all done, put the point back */
        Point_To_Mark(saved);
        Mark_Delete(saved);
                /* put the point and mark back where they started */
        if (!was_before) Swap_Point_And_Mark(name);
        return(stat);
        }
```

The way that this procedure records the initial positions of the point and mark (a flag and a saved mark) is a little confusing. Unfortunately, it is less confusing than the alternative of creating two saved marks.

```
char Get_Char(void);
void Get_String(char *string, int count);
```

```
int Get_Num_Chars(void);
int Get_Num_Lines(void);
```

These routines return buffer-related information.

Get_Char returns the character after the point. Its results are undefined if the point is at the end of the buffer.

Get_String returns up to *count* characters starting from the point. It will return fewer than *count* characters if the end of the buffer is encountered.

Get_Num_Chars returns the number of characters in the buffer (*i.e.*, the length of the buffer).

Get_Num_Lines returns the number of lines in the buffer. It is undefined whether or not one counts an incomplete last line.

```
void Get_File_Name(char *file_name, int size);
status Set_File_Name(char *file_name);
status Buffer_Write(void);
status Buffer_Read(void);
status Buffer_Insert(char *file_name);
FLAG Is_File_Changed(void);
void Set_Modified(FLAG is_modified);
FLAG Get_Modified(void);
```

These routines provide file-related operations.

Get_File_Name returns the file name that is currently associated with the current buffer. *Size* is the size of the buffer allocated for the returned file name.

Set_File_Name sets the file name for the current buffer.

Buffer_Write writes the buffer to the currently named file, making any required conversions between the internal and external representations. The modified flag is cleared and the file time is updated to the current time.

Buffer_Read clears the buffer and reads the currently named file into the buffer, making any required conversions between the external and internal representations. The modified flag is cleared and the file time is updated to the current time.

Buffer_Insert inserts the contents of the specified file into the buffer at the point, making any required conversions between the external and internal representations. The modified flag is set if the file was not empty.

Is_File_Changed returns True if the file has been changed since it was last read or written.

Set_Modified sets the state of the modified flag to the supplied value. It is most often used manually to clear the modification flag in the case where the user is sure that any changes should be discarded. This flag is set by any insertion, deletion, or other change to the buffer.

Get_Modified returns the modification flag.

```
status Mode_Append(char *mode_name, status (*add_proc)(), FLAG is_front);
status Mode_Delete(char *mode_name);
status Mode_Invoke(void);
```

These routines manage the multiple mode capability.

Mode_Append appends a mode with the supplied name and add procedure to the mode list. If *is_front* is True, the new mode is added to the front of the mode list. Otherwise, it is added at the end.

Mode_Delete removes the named mode from the mode list.

Mode_Invoke invokes the "add" procedures on the mode list to create a command set.

```
void Insert_Char(char c);
void Insert_String(char *string);
void Replace_Char(char c);
void Replace_String(char *string);
status Delete(int count);
status Delete_Region(mark_name name);
status Copy_Region(char *buffer_name, mark_name name);
```

These routines manipulate the buffer. All of them set the modification flag.

Insert_Char inserts one character at the point. The point is placed after the inserted character.

Insert_String inserts a string of characters at the point. The point is placed after the string.

Replace_Char replaces one character with another. This routine is logically equivalent to:

```
Insert_Char(c);
Delete(1);
```

but is potentially more efficient. If the point is at the end of the buffer, the routine simply does an insert.

Replace_String replaces a string as if **Replace_Char** is called on each of its characters.

Delete removes the specified number of characters from the buffer. The specified number of characters are removed after the point if *count* is positive or before the point if *count* negative. If the specified count extends beyond the start or end of the buffer, the excess is ignored.

Delete_Region removes all characters between the point and the mark.

Copy_Region copies all characters between the point and the mark to the specified buffer, inserting them at the point. The basic Emacs Wipe Region command is actually implemented as:

```
Copy_Region(kill_buffer, mark);
Delete_Region(mark);
```

This example also shows that even though an implementation presents only a single buffer to the user, a multiple buffer implementation may actually be required.

```
status Search_Forward(char *string);
status Search_Backward(char *string);
FLAG Is_A_Match(char *string);
status Find_First_In_Forward(char *string);
status Find_First_In_Backward(char *string);
status Find_First_Not_In_Forward(char *string);
status Find_First_Not_In_Backward(char *string);
```

These routines handle searching and matching strings. While it is possible to implement these routines in terms of routines that have already been defined, because of their repetitive nature, it helps performance if they are built into the sub-editor. (Actually, the same can be said of several of the other routines that have been defined such as **Insert_String**.)

Search_Forward searches forward for the first occurance of *string* after the point and, if found, leaves the point at the end of the found string. Successive searches will thus locate successive instances of the string. If not found, the point is not moved. Types of searches are discussed in Chapter 9.

Search_Backward works like **Search_Forward**, except that the search proceeds backward and the point is placed at the start of the found string (*i.e.*, the end closest to the start of the buffer).

Is_A_Match returns True if the string matches the contents of the buffer starting at the point. In other words, it returns True if **Search_Forward** would move the point *strlen(string)* characters forward.

Find_First_In_Forward searches the buffer starting from the point for the first occurence of any character in the supplied string. Thus, *Find_First_In_Forward("0123456789")* would leave the point before the first digit found after the point. Unlike the **Search_*** routines, this routine leaves the point at the end of the buffer if no characters in the string are found. A typical use of the **Find_*** routines is this sequence, which skips over the first number after the point:

```
Find_First_In_Forward("0123456789");
Find_First_Not_In_Forward("0123456789");
```

Find_First_In_Backward works in the obvious way.

Find_First_Not_In_Forward searches for the first occurence of any character *not* in the supplied string. Thus,

Find_First_Not_In_Forward("0123456789") would leave the point before the first non-digit found after the point. Unlike the **Search_*** routines, this routine leaves the point at the end of the buffer if no characters in the string are found.

Find_First_Not_In_Backward works in the obvious way.

Here are some examples of using the **Find_*** routines:

To move to the start of the next line:

```
Find_First_In_Forward(NEWLINE);
```

To move to the start of the next word:

```
Find_First_In_Forward(word_chars);
Find_First_Not_In_Forward(word_chars);
```

```
int Get_Column(void);
void Set_Column(int column, FLAG round);
```

Get_Column returns the zero-origin column that the point is in, after taking into account tab stops, variable-width characters, and other special cases, but *not* taking into account the screen width. (After all, the width of the display that the user happens to be using should not affect the actions of an editing command.)

Set_Column moves the point to the desired column, stopping at the end of a line if the line is not long enough. If the specified column cannot be reached exactly (due to tab stops or other special cases), it uses the *round* flag. If the flag is set, the point is "rounded" to the nearest available column position. If the flag is clear, the point is moved to the next highest available column position.

Characteristics of Implementation Methods

This section describes how implementation methods may be characterized, and then describes three of those methods in detail. All of those methods are assumed to store the buffers in the equivalent of main memory. Depending upon the physical characteristics of the computer, "main memory" can be actual memory, in virtual memory, or readily mappable into virtual memory. A later section describes methods for dealing with files that do not fit into main memory.

The implementation methods discussed here use two-level "divide and conquer" strategies. The first level divides the buffer into pieces. The size ranges for each piece are:

- one character
- a small number of characters (*e.g.*, 16 to 64)
- a line
- a large number of characters (*e.g.*, 512 to 4,096)
- the entire buffer

The pieces can be kept in an array, a linked list, or some other structure. The second level describes how the pieces are managed:

- no management
- extra space at the end
- buffer gap

Each of these second-level techniques will be described in detail.

No Management

In this technique, the piece is allocated exactly enough memory to hold it. The length of the piece is the only "overhead" information. A deletion is done by allocating a new piece of the desired (smaller) size, then copying the non-deleted portions from the old piece to the new one. An insertion is done by allocating a new piece of the desired (larger) size, then copying the old piece to the new one, inserting the new characters on the way. In code:

```
struct piece {
        int length;
        char data[1];   /* length characters */
        }

        /* delete LEN characters starting from START */
struct piece *Delete_From_Piece(struct piece *pptr, int start, int len)
        {
        struct piece *newptr;
        int newlen = pptr->length - len;

                /* allocate new piece */
        newptr = (struct piece *)malloc(sizeof(struct piece) +
                newlen - 1);
        if (newptr == NULL) return(NULL);

                /* copy non-deleted parts */
        memmove(&newptr->data[0], &pptr->data[0], start);

        memmove(&newptr->data[start],
                &pptr->data[start + len],
                pptr->length - (start + len));
```

```
        newptr->length = newlen;

        free(pptr);
        return(newptr);
        }

        /* insert LEN characters starting at START */
struct piece *Insert_Into_Piece(struct piece *pptr, int start,
                int len, char *chrs)
        {
        struct piece *newptr;
        int newlen = pptr->length + len;

                /* allocate new piece */
        newptr = (struct piece *)malloc(sizeof(struct piece) +
                newlen - 1);
        if (newptr == NULL) return(NULL);

                /* copy existing parts */
        memmove(&newptr->data[0], &pptr->data[0], start);
        memmove(&newptr->data[start + len], &pptr->data[start],
                pptr->length - start);
        newptr->length = newlen;

                /* copy new part */
        memmove(&newptr->data[start], chrs, len);

        free(pptr);
        return(newptr);
        }
```

Extra Space at the End

In this technique, the piece is allocated enough memory to contain it, and possibly additional memory as well. The length of the piece and the amount of the piece currently in use are kept as overhead information. A deletion never requires a re-allocation. An insertion will require a re-allocation only when the free space is used up. As the bulk of insertions are one character at a time (*i.e.*, as the user types), insertions will only require a re-allocation at relatively infrequent intervals. In code:

```
struct piece {
        int length;
        int used;
        char data[1];   /* length characters */
        }

        /* delete LEN characters starting from START */
struct piece *Delete_From_Piece(struct piece *pptr, int start, int len)
        {
        memmove(&pptr->data[start], &pptr->data[start + len],
                pptr->used - (start + len));
        pptr->used -= len;
        return(pptr);
        }

        /* insert LEN characters starting at START */
struct piece *Insert_Into_Piece(struct piece *pptr, int start,
                int len, char *chrs)
        {
        struct piece *newptr;
        int newlen;
        int amt = min(pptr->length - pptr->used, len);

                /* do as much as will fit */
        memmove(&pptr->data[start + amt], &pptr->data[start],
                pptr->used - (start + amt));
        memmove(&pptr->data[start], chrs, amt);
        pptr->used += amt;
        len -= amt;
        if (len <= 0) return(pptr);     /* done */

        start += amt
        chrs += amt;
        newlen = Round_Up_To_Block_Size(pptr->length + len);

                /* allocate new piece */
        newptr = (struct piece *)malloc(sizeof(struct piece) +
                newlen - 1);
        if (newptr == NULL) return(NULL);
```

```
        /* construct new contents */
memmove(&newptr->data[0], pptr->data[0], start);
memmove(&newptr->data[start], chrs, len);
memmove(&newptr->data[start + len],
        pptr->data[start],
        pptr->length - start);

newptr->length = newlen;
newptr->used = pptr->used + len;

free(pptr);
return(newptr);
}
```

When this version of the delete routine is compared to the "no management" version, it is simpler and will run faster. The insert routine is more complex, but most of the complexity will be executed only rarely. The path most often followed is again simpler and faster than before.

This technique has an additional benefit. In the "no management" version, memory is allocated in character-size units ranging from one character to the entire piece. In the "extra space" technique, memory is allocated in (typically) sixteen byte chunks. Typical allocation units will range from eight bytes to the piece-size limit (if any), in steps of sixteen bytes. (Actually, you probably never want to allocate a piece that is not a multiple of sixteen bytes). In any event, the dynamic range of the size of allocated units will be much smaller than in the "no management" technique. Thus, memory management will consume less overhead, and less memory will be lost to allocation fragmentation.

Buffer Gap

The buffer gap technique system stores the text as two contiguous sequences of characters with a (possibly null) gap between them. Changes are made to the buffer by first moving the gap to the location to be changed and then inserting or deleting characters by changing pointers. It thus uses memory efficiently, as the gap can be kept small and so a very high percentage of memory can be devoted to actually storing text. The overhead information includes the length of the piece, the location of the start of the gap, and the location of the end of the gap.

Here is an example buffer which contains the word "Minneapolis".

```
0   1   2   3   4   5   6   7   8                       9  10  11
| M | i | n | n | e | a | p | o |   |   |   |   |   | l | i | s |
-----------------------------------------------------------------
0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
 30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45
                P              GS                  GE
```

In this example, the buffer is 11 characters long and it contains no spaces. The blanks between the "o" and the "l" show where the gap is and do not indicate that the memory has spaces stored in it. The point is between the "n" and the "e" at location 4 and is labeled with a "P" in the bottom line (legal values for the point are the numbers from zero to the length of the buffer, 11 in this case). There are also three different sets of numbers (coordinate systems) for referring to the contents of the buffer.

First is the user coordinate system. It is displayed above the buffer. The values for it run from 0 to the length of the buffer (11). As you will note, the gap is "invisible" in this system. The coordinates label the positions between the characters and not the characters themselves. Thought of in this way, the arithmetic is easy. Thought of as labeling the characters, the arithmetic becomes fraught with special cases and ripe for fencepost errors.

Second is the gap coordinate system. It is displayed immediately under the line. The values for it run from 0 to the amount of storage that is available and it, too, labels the positions between the characters. The internal arithmetic of the buffer manager is done in this coordinate system. The start of the gap (labeled "GS" in the bottom line) is at position 8 and the end of the gap (labeled "GE") is at position 13.

Conversion from the user coordinate system to the gap coordinate system is quite easy. If a location (in the user coordinate system) is before the start of the gap, the values are the same. If a location is after the start of the gap (*not* the end of the gap!), its corresponding location in the gap coordinate system is (GapEnd - GapStart) + the location in the user coordinate system. It is a good idea to isolate this calculation either in a macro or a subroutine in order to enhance readability. Most routines (*e.g.*, the search routines) will then use the user coordinate system even though those routines are essentially internal.

The third coordinate system is the storage coordinate system. It is the bottom row of numbers in the diagram. It is the means whereby the underlying memory locations are referenced. It is labeled from X to X + the amount of memory that is available. The origin (the value of X) was chosen here to be 30 to help distinguish between the various coordinate systems. Its absolute value makes no difference. Note that it labels the memory locations themselves and so caution must be taken to avoid fencepost errors.

This technique has a very low overhead for examining the buffer. The user coordinate location is first converted to the gap coordinate system. The memory location is then looked up and its contents returned. Essentially, one comparison and a few additions are required. The purpose of the conversion is to make the gap invisible. Note that the contents of the buffer are not moved.

However, there is further overhead associated with inserting or deleting, since the gap may have to be moved so that it is at the point. There are three cases:

- The gap is at the point already. No motion is necessary.
- The gap is before the point. The gap must be moved to the point. The characters after the gap but before the point must be moved before the insertion or deletion can take place. The quantity ConvertUserToGap(point) - GapEnd characters must be moved. This quantity is numerically point - GapStart.
- The gap is after the point. The gap must be moved to the point. The characters after the point but before the gap must be moved before the insertion or deletion can take place. The quantity GapStart - ConvertUserToGap(point) characters must be moved. This quantity is numerically GapStart - point.

After the gap has been moved to the point, insertions or deletions are performed by moving the GapStart pointer (or the GapEnd pointer – it makes no difference). A deletion is a decrementing of the GapStart pointer. An insertion is an incrementing of the GapStart pointer followed by placing the inserted character in the memory location that was just incremented over.

Note that after the first insertion or deletion the gap is already in the correct place. Thus, the insertions or deletions that follow can take place without moving the gap . Further, the point can be moved away and back again with no motion of the gap taking place. Thus, the gap is only moved when an insertion or deletion is about to take place and the last modification was at a different location.

This scheme has a penalty associated with it. The gap does not move very often, but potentially very large amounts of text may have to be shuffled. If a modification is made at the end of a buffer and then one is made at the beginning, the entire contents of the buffer must be moved. (Note, on the other hand, that if a modification is made at the end of a buffer, the beginning is examined, and another modification is made at the end, then no motion takes place.) The key question that must be asked when considering this scheme is, when a modification is about to be made, how far has the point moved since the last modification?

How far can the point be moved before the shuffling delay becomes noticeable? Assume that an interval of 1/10 second is noticeable and that the editor is running on a dedicated system. Assume 250 nanosecond, 16-bit wide memory. Assume also that ten memory cycles are required for every two bytes moved (load, store, and eight overhead cycles for instructions). Then, 80,000 bytes can be moved with a just noticeable delay.

Because of the locality principle and because most files that are edited are less than 80,000 bytes in size, it seems reasonable to conclude that the average distance moved will be less than 80,000 bytes and so the shuffling delay will not be noticeable. Note that the size of the gap does not affect how long the shuffling will take and so the gap should be as large as possible.

Multiple Gaps and Why They Don't Work

Assume that we were still uncomfortable with the shuffling delay and a possible fix was put forth. This fix would have, say, ten different gaps spread throughout the buffer. What would the effects be? The idea behind this discussion is to help one understand the buffer gap system by seeing how it changes to the scheme fail.

First, the conversion from the user to the gap coordinate system would be more complex and take longer. Thus, some ground would be lost. However, this is a small loss on every memory reference in order to smooth out some large bumps, so it might still be a reasonable thing to do.

Second, the average amount of shuffling will go down, but not by anywhere near a factor of ten. Because of the locality principle, most shuffling occurs over short distances and so cutting out the "long shots" will not have a large effect.

Third, unless the writer is very careful, the gaps will tend to lump together into a smaller number of "larger" gaps. In other words, two or more gaps will meet with the GapEnd pointer for one gap the same as the GapStart pointer for the next gap. There is just as much overhead in referencing them, but the average amount of shuffling will increase (or, more precisely, not be decreased).

On the whole, the extra complexity does not seem to return proportional benefits and so this scheme is not used.

The Hidden Second Gap

On some computers, for example the two-dimensional memory system used by Multics, a second gap at the end of the buffer is provided with almost no extra overhead. The key to this gain is that the buffer is not stored in a fixed-size place. Rather, the size of the memory (or address space, to be more precise) that is holding the buffer can also increase.

The extra overhead is a check to see whether a modification is taking place at the end of the buffer. If so, the modification can be made directly with no motion of the gap.

The second gap has a greater effect than one might think because a disproportionately high percentage of modifications take place at the end of the buffer. This distortion is due to the fact that most documents, programs, etc., are written from beginning to end and so the new text is inserted (or changed) at the end of the buffer.

The increased overhead due to the second gap method is low because the check for the end of the buffer is already there (in the system hardware). There is no problem of the gaps coalescing because one of them is pegged into place. The gains are not all that great, but neither are the costs and so it should be used where supported by the operating system.

Implementation Method Overview

We started by not internally managing the pieces at all. We then added some slack space at the end of each piece. We moved to the buffer gap technique, which allowed that slack space (now called a gap) to move within the piece. Finally, we reviewed an optimization of the buffer gap technique that in some cases added the slack space at the end again.

This table presents a summary of the implementation methods. A "-" means that the combination makes no sense. An "=" means that the combination tends to be very inefficient to implement. A "*" indicates those combinations that are plausible.

size of separately	no	extra space	
managed piece	management	at the end	buffer gap
character	*	-	-
16-64 chars	*	*	*
line	=	*[2]	*
1/2-4K chars	=	=	*[3]
buffer	=	=	*[1]

Combination [1] is the standard buffer gap management method. Combination [2] is the "linked line" management method. Combination [3] is the "paged buffer gap" management method. The following sections will describe each of these methods. Later sections will compare and contrast them.

Buffer Gap

This method was first used by TECO. This method treats the entire buffer as a single object. The buffer gap technique is used to handle insertions and deletions. It is simple and straightforward, easy to implement and easy to debug. As the text is contiguous, the buffer can be transferred to or from a file with just one or two system calls. It also translates easily to the modern world of workstations with large virtual address spaces.

Linked Line

This method came into common use for Emacs-type editors when they began to be implemented on top of Lisp environments. In this method, the buffer is stored as a doubly linked list of lines. The following information might be stored for each line:

```
struct line {
        struct line *next;
        struct line *previous;
        struct piece *theline;
        int version;                    /* optional */
        struct marks *mark_lists;       /* optional */
        };
```

The **next** and **previous** fields implement the doubly-linked list. They point to the following and preceding lines, respectively.

The **line** field is managed using one of the management techniques described in the earlier characteristics of implementations section. Typically, the "leave space at the end" technique is used.

The **version** field is optional. If implemented, it is for use by the redisplay code and will be discussed in Chapter 7.

The **mark_lists** fies is optional. If implemented, it records which marks are located on this line.

A buffer *location* in this method is typically represented as a (line pointer, offset) pair. It follows from this representation that marks are always associated with a line (think about it). Marks can thus be efficiently implemented by a per-line mark list. By doing so, less time is required to update the marks after insertions or deletions because only those that are on the affected line can possibly be changed.

The operation of this method is straightforward. New lines, when created, are simply spliced into the list at the appropriate place. Note that no characters are stored to indicate line breaks. If the new line is inserted into the middle of an existing line, some movement of the text on the end of the old line to the newly allocated line is all that is required.

The line itself is typically stored by the "extra space at end" technique. However the buffer gap technique could also be used. Regardless of the technique used, it is important to ensure that no limits are placed on the length of a line.

If your implementation includes automatic word wrap, do *not* split lines on "soft" newlines, because the overhead of shuffling the line allocations while the user types will be large. Instead, use a buffer gap technique within a line and only split lines on hard newlines (*i.e.*, paragraphs).

Paged Buffer Gap

In this method, the buffer is divided into "pages" of one to two Kilobytes each while the file is read in. Each page is then managed with the buffer gap technique. The pages are organized into an array or linked list. This method has two points in its favor:

- Since each page is small, the gap need never be moved very far.
- Since all pages are the same size, memory management is kept simple.

These points are very important in resource-limited environments. For example, this method was used in the Mince text editor that initially ran on CP/M systems with 48 Kilobytes of memory and small floppy disks. That editor implemented a complete paged, virtual memory environment for its buffers. (The implementation included all of the then-current optimizations found in virtual memory operating systems.) The size of the buffer was limited only to the amount of available disk space.

Since main memory was so limited (in some systems, only three or four pages could be kept in memory at once), the excess pages were swapped to disk. A descendant of the Mince editor called The FinalWord used the disk storage to even greater advantage: the control information was written to disk as well, thus allowing the complete editing state to be saved between invocations as well as being recoverable in the event of a system crash.

Other Methods

The other methods involve tracking small chunks of characters or even individual characters. While they are in principle do-able, their small object size serves to increase the amount of memory and CPU overhead, unfortunately without offering any compensating advantages. Thus, they remain largely unused.

Method Comparisons

This section compares the three main methods in a variety of ways.

Storage

These comparisons are on a per-buffer basis. They also assume eight-bit characters. Our sample buffer will consist of 150, 60-character lines.

A buffer gap implementation requires a fixed-size header (say, eight bytes) plus one byte per character of text. Total size is 9,008 bytes.

A linked line implementation requires a fixed-size header (say, eight bytes), plus a fixed-size header per line (say, twelve bytes) plus one byte per character of text, plus on the average eight bytes of fragmentation per line. Total size is 8 + 150 * 12 + 9,000 - 150 (don't store newline characters) + 8 * 150 = 11,858 bytes.

A paged buffer gap implementation requires a fixed-size header (say, eight bytes), plus a fixed-size header per page (say, twelve bytes), plus the pages (say, two Kilobytes each). Total size is 8 + 12 * 5 + 2,048 * 5 = 10,308 bytes.

The linked line method pays a large storage price because of its relatively high per-line overhead. In this example, the per-line overhead was about 33 percent.

The paged buffer gap method pays a large price in this example because of the mismatch between the page size and the buffer. If memory is tight, a smaller page size can be selected. However, the extra overhead is paid only once per buffer, since it occurs only at the end.

Crash Recovery

These comparisons assume that a recovery program is examining the core image of an edit session that was interrupted.

In the buffer gap method, crash recovery is relatively easy and fail safe. In general, the start and end of the buffer can be found if a marker is left around the buffer (say, a string of sixteen strange (value 255) bytes) and the buffer is everything between them. The gap can be recovered and manually deleted by the user or, if it too is filled with a special marker, can be automatically deleted.

In the linked line method, crash recovery is harder. Recovery is greatly aided by erasing freed memory. Basically, you perform the recovery by picking a block at random and examining it. If it can be parsed into a line header (*i.e.*, the pointer values, etc., are reasonable), continue (a careful selection of header formats will help). Otherwise, pick a different block. You can then follow the next and previous pointers and parse them. If this works three or four times in a row, you can be confident that you have a handle on the contents. If a header doesn't parse, it is because it is either a part of a line (either pick again at random or go back one chunk and try again) or a header that was being modified (in which case you are blocked from continuing down that end of the chain). In the latter case, go in the other direction as far as possible. You now have one half of the buffer. Repeat the random guess, but don't pick from memory that you have already identified as part of the buffer. You should get the other half of the buffer. Leave it to the user to put the two halves together again. If the freed blocks are not erased, the chance of finding a valid-looking header that points to erroneous data is very high.

In the paged buffer gap method, crash recovery is easier than with linked line, but harder than with buffer gap. As with buffer gap, marker bytes can help you locate the buffer pages, and the gap can be recovered either manually or automatically. The pages are strung together just like lines were: it is just that there are fewer pages to work with.

Efficiency of Editing

These comparisons examine the typical types of effort required to insert a character or line.

	insert character	insert line	maximum motion
buffer gap	move gap pointer update	same as insert character	buffer
linked line	move/scroll line pointer update	allocate header line splice into line	
paged buffer gap	if full, split page move gap pointer update	same as insert character	page

As might be expected, the buffer gap scheme is the most efficient, although you will occasionally encounter a comparatively long pause. The linked line scheme adds lots of overhead to the simple operations and cuts out the occasional comparatively long pauses. But you will often be hit badly if, for example, you insert in the middle of a 4,000-character line. The paged buffer gap method removes the pauses at the price of a moderate increase in complexity.

Efficiency of Buffer/File I/O

This section compares the way that the methods handle buffer/file I/O.

The buffer gap method is extremely efficient. Reading a file into a buffer consists of these operations:

- Determine the file's length.
- Allocate enough memory to hold the file, plus some extra for growth.
- Read the file in.

On some systems, even this can actually be improved. For example, on many systems you can just map the file into the address space of your process. No actual data motion takes place until you modify one of the pages. At that time, the page is copied and the modifications are written to the new copy (this is sometimes called "copy-on-write"). Writing the file out can take two calls (one to cover the text in front of the gap and the other to cover the text after the gap).

The linked line method has a very obvious but poor algorithm to read the file in. This code fragment illustrates the algorithm:

```
if (fd = fopen(FILE, "r")) == NULL) {
        ...error...
        }
while (fgets(buf, sizeof(buf), fd) != NULL) {
        if (!Allocate_Line(strlen(buf)) {
                ...error...
                }
        Build_Line(buf);
        }
fclose(fd);
```

This algorithm has a system call (in the *fgets*) and allocation for every line in the buffer. An improved algorithm would read the whole file into memory (or at least read in large chunks), then allocate lines out of that memory. This improvement at least reduces switching between the system and program contexts. (A sufficiently good *fgets* implementation would effectively do this. Unfortunately, the libraries that come with many C compilers are not sufficiently good...)

The paged buffer gap method could be just as efficient at reading as the buffer gap method. It would operate by reading the entire file in as a block, then dividing that block up without moving any data. The first insert on each page will cause a page split, though. Writing would have a worst case number of system calls equal to twice the number of pages in use.

Efficiency of Searching

This section compares the way that the three methods handle searching.

If your implementation is such that the search time dominates the setup time, all three methods are equivalent. In the case where the setup time dominates the search time, the methods do perform differently, and that is the case that will be examined.

This comparison assumes that the search routines are "built in" to the buffer management code for performance reasons. While they could call **Get_Char** for every character, doing so would probably not be very efficient. Given equivalent implementations of the actual search code, the main difference among the methods is the number of times that the inner search loop is called. In other words, it is the number of distinct pieces that must be searched.

The buffer gap method calls the inner loop twice: once for the text in front of the gap and once for the text after the gap. While an implementation could move the gap so that the search routine only needs to be called once, doing so goes against the reason for having the gap in the first place. Keep in mind that searching happens a lot. For example, two searches are done whenever the "forward word" command is given. The whole point of the buffer gap method is to avoid moving the gap until it is necessary. An even worse way to go astray is to move the gap as you search. This replaces one large, efficient gap move with many smaller ones. We have already observed that even fairly large gap moves are not very noticeable, so "optimizing" them out is not a wise move. In conclusion, invoking the search loop twice is quite efficient.

The linked line method invokes the inner search loop once for every line in the buffer. In our earlier example, this means that it would be invoked 150 times. What's worse, the linked line method does not store newline characters. Rather, they are implied by the line structure. Hence, whenever a newline character is in the search string, that character must be handled in a special manner. While some optimizations can (and should) be made (for example, searching for "x<newline>" means that you only have to look at the last character of each line), the code complexity required to make these optimizations adds its own performance penalty.

The paged buffer gap method lies somewhere between the other two. In the earlier example, the search routine would have to be invoked ten times. This is not enough to incur a significant performance penalty, but it is one more reason not to use this method if buffer gap will work.

Multiple Buffers

This section compares the way that the methods handle multiple buffers.

The buffer gap method offers no choice: the buffers must follow one another in memory (where else could they be?). This arrangement becomes bad when the total size of all buffers becomes large enough that an objectionable pause occurs when switching buffers. The arrangement can be improved by leaving extra "gaps" between buffers.

The linked line method has two choices. First, all lines can be allocated out of a common pool. Thus, over time, the buffers tend to "intertwine" (*i.e.*, the lines of one buffer are mixed in with the lines from other buffers in physical memory). This choice tends to maximize the density of text and thus makes the most efficient use of memory. (See also the discussion in the next section about paged environments.) The other choice is to allocate memory among buffers, then allocate all lines for a buffer from within each buffer's allocation. There must, of course, be some way to change the buffer allocations.

The paged buffer gap method has the same two choices as linked line.

Paged Virtual Memory

This section compares the way that the methods perform in paged, virtual memory environments. It concentrates on the effects that occur when main memory is full and paging is going on ("tight memory").

Some operations, such as searching the entire buffer, require that the buffer be accessed sequentially. In situations where the entire buffer does not fit into memory, no management method can avoid some page swapping. That type of situation is not analyzed here.

The buffer gap method generally works well in this environment. Its highly compact format allows for accessing large portions of the buffer with only a few pages in memory. Its sequential organization also implies that it has a very good locality of reference and so the nearby pages are heavily referenced and likely to be around.

Its major problem is, as usual, the worst case situation of a large gap movement. In tight memory situations, moving the whole buffer implies that all of the buffer's pages must be swapped in and – most likely – swapped out again. Overall, the buffer gap method does as well as can be expected. The nearby portions of the buffer will tend to be in memory because of locality of reference, but distant portions may in general have to be paged in.

The linked line method has many disadvantages and no real advantages in tight memory situations. First, if an intertwining multiple buffer scheme is used, over time the effective page size is reduced by a factor that tends to increase over time to equal the number of buffers. This reduction is due to the random nature of the buffer memory allocations, the fact that many lines tend to fit into one virtual memory page, and the consequence that over time a virtual memory page tends to hold lines from as many buffers as possible. However, when a given buffer is in use, the storage used by the other buffers is consuming memory.

Even separating the buffer memory does not resolve this problem. When buffers get large, different parts of the same buffer may act in the same fashion as the separate buffers to decrease the effective page size. In extreme cases, a desired "target" line may be in memory, but in the process of following the linked list, the target line may be swapped out!

Notwithstanding the above, this method does not pack data as tightly as the others. An earlier example showed that the overhead for the linked line method is about 33 percent. Thus, the page size is effectively reduced by 25 percent.

When all factors are combined, a typical linked line system is probably reducing its effective page size by about 50 percent. The result is that, for example, if a computer has one Megabyte of memory in 512, two Kilobyte pages, the linked line method would effectively treat this as 512 Kilobytes of memory (512, one Kilobyte pages).

The paged buffer gap method is essentially the buffer gap method, modified to improve performance in tight memory situations. It removes the lengthy gap moves and consequently lowers the probability of thrashing. When designing such a system, the buffer page size should be set the same or as a multiple of the virtual memory page size. Thus, in even the tightest memory situations an insert or delete of one character will only affect at most two pages of buffer memory.

Conclusions

Use the buffer gap method if at all possible.

Only use the linked line method if you are implementing in an environment that likes to manipulate lists of small objects; for example, Lisp environments.

Only use the paged buffer gap method if resources are tight.

Editing Extremely Large Files

This section examines techniques for editing extremely large files. The first type of extremely large file is those files that are so large that reasonable assumptions based on current workstation and mainframe architectures are no longer valid. Given the current generation of computing hardware, this starts happening around 512 Megabytes (ten years ago, I had set this number at 64 Megabytes). At that size, even simple operations such as a string search can take several minutes to run on a fast processor with the whole file in memory.

Although there are one or two interesting hacks you can do to stay alive, life is simply not bearable when trying to edit such a large unstructured file. The alternative which large data base implementors have known about for years, is to structure the file. This alternative is palatable because an unstructured editor can still be used to edit the pieces of a structured file. The other reason why this limitation is not bothersome is that there aren't all that many such files to edit. (For example, the largest file on a computer system that I often use is only about 33 Megabytes.) The vast majority of files are much smaller. Gigantic files call for special tools.

The other type of "extremely large" file is encountered on resource-limited systems. In these cases, files that would otherwise be handled easily can now cause the system to bog down. For example, in the first generation of microprocessor systems, an extremely large file might have been 100 Kilobytes. There are several ways of dealing with such files.

One way is to divide the file into chunks, each of which fits into memory. You read in the first chunk, edit it, write it out, read in the second chunk, and continue until you are done. While arbitrary editing can be done within a chunk, in general you cannot back up to a previous chunk without finishing the file and starting over. This method was used by the original TECO editor.

Another way is to use a three-file system. As the user moves down in the file, the "from" file is read and a "to" file is written. When the user wants to back up, the "to" file is read and a "holding" file is written (the chunks will appear in reverse order in this file). When the user moves forward again, the "hold" file is read (backwards) until exhausted, then the "from" file is read from again.

The best method to use if main memory is tight is paged buffer gap. If disk storage is also tight, serial chunking is best.

Difference Files

There is another type of buffer management that has been used to good advantage in several cases. It is called the *difference file* method. It works best when recording relatively few changes, and those changes are small when compared to the size of the buffer. In this method, the buffer is not kept in memory at all. Instead, only a list of differences between the buffer and the "original" file is kept. When information is being retrieved from the buffer, it is read from the file as needed and the differences applied.

This method has much promise. For example, in many cases, a file will be read into a buffer, looked at by the user, and the buffer deleted. In this example, the difference file method essentially acts as a file viewer. This is particularly encouraging when you realize that the larger a file is, the less likely it is to be dramatically changed.

On the other hand, this method does not scale well. For example, I have been editing this chapter continuously for several hours. As it turns out, the current version bears little resemblance to the original. The *best* description of the changes is "throw out everything and insert X," where "X" is the entire chapter. I would expect most "reasonable" descriptions of this chapter to wind up being several times as large as the chapter itself. Hence, you now have to address the question of how to edit the description of the differences. Let's see, we can use buffer gap, linked line, or paged buffer gap...

In addition, this method does not work well in tight memory situations. As I write this chapter, it occupies about 60 Kilobyes of the roughly 100 Kilobytes of free RAM disk space on my lap-top computer. I simply don't have the room to store what amounts to both the "old" and "new" versions at once.

But, one might argue, you don't have to store the "old" version as it appears on disk. Well, that's true, but the disk is sitting on the table by my side, not in the floppy drive. So how can it be read?

In conclusion, this method works well when one is essentially viewing files. However, it breaks down badly as changes to the file accumulate. It can easily end up taking several times as much memory to track all the changes as to simply store the modified version. Finally, the changes can easily become so large that either a "real" buffer management method must be implemented to simply track the changes or "snapshot" files must be created so that changes can be tracked from a new base. Hence, why bother with the extra overhead?

Questions to Probe Your Understanding

Rectangular regions include only those characters between the point and mark that are also in columns between those of the point and the mark. Define a set of interface procedures to handle rectangular regions. (Easy)

Come up with a situation where it would be a good idea to implement the buffer as a linked list of characters. (Medium)

The first buffer gap editor was TECO, which was also among the first text editors ever written. It was written in the early 1960s. Explain why many people spent the next fifteen years reinventing hard-to-use, limited-functionality line editors. (Medium, but if you succeed I would like to hear your explanation)

Devise a buffer management scheme better than buffer gap. (Hard, but if you succeed, you can probably get a Ph.D. thesis out of it)

Chapter 7: Redisplay

> One, two! One, two! And through and through
> The vorpal blade went snicker-snack!

The previous chapter described a way of dividing the implementation into parts and covered one of those parts, the internal sub-editor. This chapter describes the redisplay part.

This chapter will start by discussing the general constraints that affect redisplay. It will then describe the external interface and some of the internal interfaces used by redisplay (the "procedure interface" definitions). It goes on to discuss many of the considerations that affect implementations of the algorithms. Finally, it describes the redisplay algorithms.

Constraints

Redisplay, or incremental redisplay, to give it its full name, is that part of the implementation that is responsible for ensuring that all changes to the buffer are promptly reflected on the user's display. As is evident from the definition, there are two parts to redisplay's job.

The first part is to ensure that all changes are indeed tracked. In the absence of the second part, this part would be quite easy.

The second part of the job is to ensure that the changes are made "promptly." In this context, "promptly" means that the amount of clock time required to make the updates visible is minimized. Clock time is the combination of transmission time, CPU time, and disk access time that is perceived by the user as the delay from when a command has been entered to when the display has been updated.

In general, the buffer's contents will change only a small amount during any one command. The screen will thus only have to be changed by a small amount in order to reflect the changed buffer contents. Hence, the algorithms concentrate on incrementally redisplaying the buffer; the entire process is thus referred to as incremental redisplay. Fortunately, it turns out that in cases where the buffer is changed drastically, the increment-oriented approach to redisplay works quite well and so there is no need for multiple algorithms.

This discussion of incremental redisplay assumes a model of the system where the editing is done on a main processor which communicates with a display. If the main processor is the same as the display, the bandwidth of the CPU to display communications channel can be very high. However, the considerations remain unchanged: only the relative weights change. Incremental redisplay is an optimization between CPU time, display-processing time, and communications-channel time, with a few memory considerations thrown in.

The first major constraint is the speed of the communications channel. Typical speeds that are available are 300, 1200, 2400, and 9600 bps. Memory-mapped and other built-in displays run at bus speeds: communications speed is essentially infinite.

A typical video display has a 24 x 80 character screen. At 300 bps, it takes three seconds to reprint a line and over a minute to refresh the whole screen. At 1200 bps, less than one second is required to reprint a line and about sixteen seconds to refresh the screen. At 9600 bps, it will take one to two seconds to refresh the screen. The speed of the communication thus greatly affects the amount of optimization that is desired. At 300 bps, the user may notice even one extra transmitted character, while at 9600 bps, reprinting entire lines does not take an appreciable amount of time. One dimension of the optimization is thus clear: the importance of optimizing the number of characters sent increases in proportion to the slowness of the communication line.

The second major constraint is the speed of the display device. It takes time for the display to handle each command, and this time can affect the choice of commands sent to the display. For example, if a line ending in:

```
whale
```

was changed to:

```
narwhale
```

the redisplay code could elect to position the cursor just before the "w", insert three characters, then send "nar". Alternatively, it could position to the same place and just send "narwhale". The latter would be more efficient unless the display could accept and perform the "insert three characters" command in less than five character times (possible). As has been mentioned before, some memory-mapped displays actually process commands very slowly. As the effective display speed is so low, it is important to use good redisplay algorithms on those displays.

User interface considerations also affect which command sequences should be sent. For example, while it might be acceptable from a pure clock-time point of view to reprint an entire line, users do not like to see text which has not changed in the buffer "change" by being reprinted. The flickering that is generated by the reprinting process attracts the user's attention to that text, which is undesirable as the text has not, after all, changed. Thus, avoiding extraneous flickering and movement of text is good. The amount of perceived flicker will vary from display to display, being highly dependent upon such factors as display command set, display speed, internal display data-structures, timing, and phosphor.

The third major constraint is the CPU speed. On some computers, computing an optimal redisplay sequence takes longer than is saved by the optimizations ("optimal" considering only the communications channel and display). On those machines, the correct optimization is to send the "less-than-optimal" sequence.

CPU time must be spent in order to perform any optimizations. If the CPU time spent exceeds some small amount of clock time, the user will perceive response to be sluggish. It is therefore desirable to minimize the CPU time spent on optimizing the redisplay. However, the communications channel speed also makes a difference. If the line is slow, extra CPU time can and should be spent (at 300 bps, it is worthwhile to spend up to 30 msec. of CPU time to eliminate one character from being transmitted (which takes about 30 msec.)). However, at higher speeds it is generally not practical to heavily optimize the number of characters sent, as it can easily take longer to compute the optimizations than to transmit the extra data. This relaxation of the optimization is subject to the user interface constraint outlined above.

The fourth constraint is the memory size. For example, one technique stores a copy of the entire screen, character by character. This technique works quite well in general. However, where memory is tight this technique may not be feasible.

Procedure Interface Definitions

This section describes two interfaces. The first one is the (external) interface that redisplay presents to the rest of the editor implementation (*i.e.*, the sub-editor and the user-oriented commands). The second interface is the internal interface used by the redisplay to isolate the display-specific portions of its internal code.

In this section, the term *display* refers to the hardware operated by the user. A display has a keyboard, screen, and perhaps a graphical input device. The *screen* is the part of the display that shows output. A *window* is a logical screen. One window can occupy the entire screen, or more than one window can share the screen, perhaps even overlapping. A window can never be larger than the screen.

Editor Procedures

```
status Window_Init(char *display);
status Window_Fini(void);
status Window_Save(FILE *fptr);
status Window_Load(FILE *fptr);
```

Window_Init is the basic set-up-housekeeping call. It is called once, upon editor invocation. It should perform all required one-time initialization, including keyboard and screen initialization. No other editor interface call except for **Window_Fini** can be legally called unless **Window_Init** returns a successful status. The parameter indicates the display type. Presumably, if the parameter is null, the routine will determine a default display.

Window_Fini terminates all state information. Once called, **Window_Init** must be called again before other editor interface calls can be legally made.

Window_Save saves the current redisplay state information in the file opened on the specified file descriptor. Presumably, **World_Save** opened the file, then called this routine.

Window_Load loads all redisplay state information from the file opened on the specified file descriptor.

If you are creating a "stripped down" editor, then these routines would not do anything. They can be put in as stubs if there is a reasonable possibility that the editor will be embellished later.

```
void Redisplay(void);
void Recenter(void);
void Refresh_Screen(void);
```

Redisplay performs one incremental redisplay. If it runs to completion, the screen will accurately reflect the buffer. However, this routine also checks for type ahead. If the user does type (or use the graphical input device) before the redisplay has finished, this routine will notice that event and abort the redisplay in a safe manner. Presumably, the user will quit typing ahead at some point and redisplay can then complete.

Recenter operates as does **Redisplay**, except that it moves the point to the center of the window (technically, the preferred percentage). This procedure is needed because there is typically a user-level command to perform this operation.

Refresh_Screen operates as does **Recenter**, except that it assumes that the screen has been corrupted. Thus, this routine ensures that the screen is correct no matter what else may have happened.

```
void Set_Pref_Pct(int percent);
int Get_Point_Row(void);
int Get_Point_Col(void);
```

Set_Pref_Pct sets the preferred percentage. After a **Recenter** or other similar operation, the point will be on a line approximately this percentage of the way through the window. A good default value is about forty percent. With this default on a 22-line window, the point will be on line 9.

Get_Point_Row returns the number of the row within the window that the point is at.

Get_Point_Col returns the number of the column within the window that the point is at. This may not be the same as the column returned by **Get_Column**, as that routine does not take into account line wrap.

```
window_data Window_Create(window_data wind);
status Window_Destroy(window_data wind);
status Window_Grow(window_data wind, int amt);
int Get_Window_Top_Line(window_data wind);
int Get_Window_Bot_Line(window_data wind);
location Get_Window_Top(window_data wind);
location Get_Window_Bot(window_data wind);
```

These routines are used to manipulate multiple windows, if you choose to offer that feature. The definitions provided here only allow for horizontal windows (*i.e.*, all windows occupy the full width of the display). Vertical windows and overlapping windows are not covered, although they are often implemented.

Window_Create creates a new window. It operates by splitting the supplied window in two. Both windows initially show the same data. It returns a window descriptor for the second window.

Window_Destroy destroys the supplied window. The window above this one expands to occupy the vacant screen space. Note that *Window_Destroy(Window_Create(wind))* results in no change.

Window_Grow grows the specified window by the specified number of lines. The window is grown by moving the top line up.

Get_Window_Top_Line returns the screen line that contains the top line of the specified window.

Get_Window_Bot_Line returns the screen line one after the bottom of the specified window. This is the same value as **Get_Window_Top_Line** of the next lower window.

Get_Window_Top returns the location in the buffer just before the character that is at the top left part in the window. Several of the user commands use this information.

Get_Window_Bot returns the location in the buffer just after the character that is at the bottom right part of the window (or as close as you can get to it). This is the same location that would be returned by **Get_Window_Top** if the window were exactly one window below its current position. Several of the user commands use this information.

Display Independent Procedures

This section describes the display functions used by redisplay. The routines that implement these functions are not part of redisplay itself. A full discussion of this topic is beyond the scope of this book (but is covered in Linhart 1980). In essence, the problem is that every display manufacturer has decided on a different set of features and offers different ways of accessing those features. To solve the problem, a set of routines is needed which can isolate these differences, as well as a way of selecting among different sets of such routines as the display changes.

Although not recommended, it *is* possible to cover a lot of displays by assuming that the display accepts the ANSI escape sequences (*i.e.*, the display is a DEC VT100). Most modern displays accept these sequences. However, many older displays do not. In addition, not all displays take the same amount of time to process a given command. Thus, there is still per-display information to consider.

There is one piece of existing technology to mention, and that is exemplified by the *curses* package available on many UNIX systems (similar packages are available under other names for other systems). It not only provides display independent functions, albeit with a somewhat different interface, it also performs the redisplay for you! However, as the purpose of this chapter is to explain how redisplay works, that package will receive no further mention.

The following set of procedures will allow display-independent operations for most displays. The procedure interfaces isolate the operations that are used by redisplay.

```
status Key_Init(char *display);
status Key_Fini(void);
char Key_Get(void);
FLAG Key_IsInput(void);
private Key_FunctionKeys(void);
```

Key_Init and **Key_Fini** operate in the by now familiar fashion. They will be called by the **Window_Init** and **Window_Fini** routines. In particular, though, these routines make sure that all processing of input characters is turned off (*i.e.*, set it for "raw" input) and all configuration information is loaded.

Key_Get waits for a key to be pressed and returns it. Keys that send multiple characters (*e.g.*, function keys) are returned one character at a time.

Key_IsInput returns True if input is available or False if it is not. It is used, for example, by **Redisplay** to determine whether to abort.

Key_FunctionKeys returns information about the function keys available on this keyboard. This information includes key placement, key labelling, and the codes returned by the keys. The information is returned in an implementation-defined manner (*i.e.*, you get to invent your own representation).

```
status Screen_Init(char *screen);
status Screen_Fini(void);
int Screen_Rows(void);
int Screen_Columns(void);
private Screen_Atrributes(void);
```

Screen_Init and **Screen_Fini** operate in the by now familiar fashion. They, too will be called by the **Window_Init** and **Window_Fini** routines. In particular, though, these routines make sure that all processing of output characters is turned off (*i.e.*, set it for "raw" output) and all configuration information is loaded.

Screen_Rows returns the number of rows in the screen. In this case, a row is the granularity of screen output. On a graphics screen, a row would be one pixel.

Screen_Columns returns the number of columns in the screen. In this case, a column is the granularity of screen output. On a graphics screen, a column would be one pixel.

Screen_Attributes returns information about the attributes (*e.g.*, boldface, reverse video, blinking, etc.) that the screen supports. The information is returned in an implementation-defined manner (*i.e.*, you get to invent your own representation again).

```
void Set_Cursor(int row, int column);
void Set_Row(int row);
void Set_Column(int column);
void Set_Attr(private attributes);
int Get_Row(void);
int Get_Column(void);
private Get_Attr(void);
void Put_Char(char c);
void Put_String(char *str);
void Beep(void);
```

Set_Cursor sets the cursor to the specified row and column. It is assumed that the optimal (*i.e.*, least cost) command sequence will be selected.

Set_Row sets the cursor to the specified row, without affecting the column. Instead of a separate routine, this could be multiplexed onto **Set_Cursor**, say by one of the following:

```
Set_Cursor(row, -1);
Set_Cursor(row, Get_Column());
```

Set_Column sets the cursor to the specified column without affecting the row. The functionality provided by this routine could also be multiplexed onto **Set_Cursor**.

Set_Attr sets the current attributes to those specified.

Get_Row returns the row that the cursor is on.

Get_Column returns the column that the cursor is on.

Get_Attr returns the current attributes.

Put_Char outputs the supplied character to the screen, updating the cursor position. The character is always displayed: it is never part or all of a command sequence.

Put_String outputs the supplied string and leaves the cursor after the string. It otherwise works as per **Put_Char**. These strings are always displayed, even if they appear to contain screen commands. Commands may be sent to the screen only by means of the supplied procedures.

Beep rings the screen's bell or flashes the screen.

```
CLEOL(void);
Clear_Line(void);
CLEOS(void);
Clear_Screen(void);
```

CLEOL sends the command sequence that optimally clears from the current cursor position to the end of the current line. The cursor does not move.

Clear_Line sends the command sequence that optimally clears the entire current line, and leaves the cursor in column 0 (you are assuming a zero-origin on all of these numbers, aren't you?).

CLEOS sends the command sequence that optimally clears from the current cursor position to the end of the screen. Lines after the current one are completely cleared. The cursor does not move.

Clear_Screen sends the command sequence that optimally clears the entire screen, and leaves the cursor at the upper-left corner (row 0, column 0).

```
void Insert_String(char *str);
void Delete_Chars(int count);
void Insert_Lines(int count);
void Delete_Lines(int count);
void Scroll_Lines(int from, int to, int count);
```

These commands are available on advanced displays only (by the definition of an advanced display from Chapter 2), and each is assumed to send the optimal command sequences to effect its purpose.

Insert_String takes a string, determines the optimal command sequence required to insert it, and inserts it starting at the current cursor location. Line wrap is not performed: excess characters are dropped off the right edge of the screen. This routine could have been defined to accept a count instead of a string and to insert blanks. However, it is easier to optimize command sequences by having the string to be inserted available.

Delete_Chars accepts a count and deletes that many columns. Line wrap is not performed. Blank columns are inserted from the right edge of the screen.

Insert_Lines accepts a count and inserts that number of blank lines, starting with the line that the cursor is on (thus, you can insert lines at the very top of the screen).

Delete_Lines accepts a count and deletes that number of lines, starting with the line that the cursor is on (thus, the line at the very top of the screen can be deleted). Blank lines are scrolled in from the bottom.

Scroll_Lines accepts a *from* line, a *to* line, and a *count.* The lines starting with the *from* line, and up to but not including the *to* line, are scrolled by *count* lines (positive scrolls the lines up and negative scrolls the lines down).

```
private Screen_Timings(private goal);
```

Screen_Timings accepts a description of a goal, and returns timing information on the various choices of screen routines that could be used to achieve that result. The description takes into account the current screen status. The information is to help the redisplay code select the best screen routine, not to help the screen routines optimize their own performance (such optimizations are assumed to be done anyway). As with the other private data types, the information is returned in an implementation-defined manner (*i.e.*, you get to invent your own representation). Note that the two private data types in this procedure's definition (that for *goal* and the procedure itself) refer to *different* data types with different representations.

Considerations

This section describes various considerations that go into the redisplay algorithm. In other words, these are the ways in which the algorithm gets complicated. While none of these ways are particularly difficult to implement in themselves, collectively they would clutter the redisplay algorithms presented later. Hence, you should keep these topics in mind when reviewing the algorithms.

The topics in this section are only vaguely related to each other and are in no particular order.

Status Line

In general, each buffer will have some lines of status information. In addition, there may be general editor status information. Finally, there may be lines of separators between windows. (One hopes that on "small" screens (*i.e.*, those with less than, say, fifty lines), the numbers for these are "one," "none," and "none: use the buffer status line as a window separator" in order to devote as many lines as possible to showing the text being edited.)

In any event, this "framework" information must be retained and displayed. The user-oriented command routines and redisplay must work together to provide this infrastructure.

Here are some sample types of per-buffer status information:

- the file name
- the buffer name (may be the same as the file name)
- the buffer status: unmodified, modified, read-only
- the current modes
- the point position in characters and/or buffer length

- the point position as a percentage
- the location of the top of the window as a percentage (or "top", "bot", or "all" as appropriate)
- the point column
- the current attribute
- the current line and number of lines

Of course, any one editor implementation will only show some of this information at a time. This list is not definitive.

Here are some sample types of editor status information:

- the name and version number of the editor
- copyright information
- the current date and time
- the current system "load average," or other system information

Again, any one implementation may only show some of this information, and this list is not definitive.

End of the Buffer

There are two cases that must be handled.

First, if the entire buffer fits in the window, you will run out of buffer before you run out of window. The caveat here is to ensure that this case is properly detected and that the entire buffer is shown, with the start of the buffer at the upper-left corner.

Second, if the entire buffer does not fit in the window but the end of the buffer appears, the end should be close to but not at the bottom of the window.

Those portions of the window that follow the end of the buffer can be left blank or marked in some fashion. As a rule, Emacs-type editors leave that part of the window blank.

Horizontal Scrolling

A window has a finite width. Some lines will not fit within that width. There are two popular ways of handling such a situation: horizontal scrolling and line wrap. Ideally, your editor should offer the user a choice between them. This section will describe the first.

When performing horizontal scrolling, a line longer than the window width will spill off the edge: the part of the line that does not fit thus will not be visible to the user. As the user types, the text being displayed will adjust so that the text around the point is always visible. In addition, the user should be provided with commands to move the window left or right (with a few characters of overlap). In addition, the status line should contain indicators that show whether text is currently lost off of either the left or the right sides (use separate indicators).

Line Wrap

When performing line wrap, the window never moves left or right at all. Instead, the text that would have been clipped off of the right edge of the window is wrapped to the next line. If the line is sufficiently long, it may wrap two or more times.

In this type of display, no window motion commands are required. In addition, the status indicators are also not required, although you may wish to mark the wrapped lines.

Line wrap introduces a new problem that must be handled properly: that of single lines that, when wrapped, occupy the entire window. Although rare, such lines do show up from time to time in non-text files.

When horizontal scrolling and line wrap are compared, neither comes out a clear winner and both offer valuable features, hence the assertion that your implementation should support both line wrap and horizontal scrolling.

The advantages to horizontal scrolling are that it is easy to implement, and can be processed quickly.

One of the disadvantages is that it requires a fast display. Consider the case when the user has a 160-column line displayed in an 80-column window. On the average, the window will have to be shifted twice per line of typing. Another disadvantage is that clipped text appears to have been deleted. It can be rather disconcerting to the user to have this text vanish and reappear.

One of the advantages of line wrap is that all of the text is always visible. In addition, when editing a very long line, the entire window shows the immediate context. In contrast, when editing the end of a long line when using horizontal scrolling, most or all of the remainder of the window will be blank, having been scrolled off the left edge.

The main disadvantages to line wrap are the additional complexity in the redisplay required to handle the line wrap, the very poor presentation when lines are only slightly wider than the window, and the disconcerting multi-line "shifting" that occurs when a user is inserting or deleting near the start of a wrapped line.

Word Wrap

Once you have line wrap, the next logical step is to break the line on a word boundary instead of a character boundary. You then offer "word wrap," a feature found in almost every word processor available today. Typically, a word processor will store each paragraph of text as a single line and simply perform word wrap upon it. Ruler lines are used to adjust the margins and change the type of justification.

This is a very nice feature to offer. It does have some pitfalls for the unwary implementor, however:

- Your redisplay now has to handle look-ahead.
- You are going to have to decide where to break the lines (white space only, include dashes, include other punctuation?).
- Your users are going to want ruler lines, and so you must provide all of that infrastructure.
- You will have to track where the word wrap actually occurs because the user thinks (and hence the line-manipulating commands operate) in terms of the lines as displayed.

If you do implement word wrap, you may as well go the whole way and support flushing right, centering, and justification of text during display.

Tabs

Tab characters can be handled in two ways. The first way is to not handle them at all. Instead, convert them to spaces upon entry. In this case, the redisplay code never sees those characters and hence doesn't need to deal with them.

The second - and by far the most common method - is to treat the tab as a "cursor control command" that in effect says "think of me as n blanks, where n is the number of units to the next tab stop." Thus, when the redisplay code encounters a tab, it computes n, then pretends that it is displaying n consecutive blanks (or a single blank of width n). N can be computed in one of three ways.

First, tab stops can be set every c columns (or characters). In a zero-origin numbering system, tabs set every c columns are set at columns 0, C, 2*C, 3*C, ... For example, when c is 8, tabs are in columns 0, 8, 16, 24, ... The *C* language expression to compute n is:

```
n = c - x % c;
```

where x is the current column.

The second way to set tab stops is to allow them to be set at arbitrary column positions. This way is often used in ruler lines in simple word processors. In this case, you must decide on a representation such as a bit array or an array of the columns where tabs are set.

The third way to set tab stops is to allow them to be set at arbitrary positions, where the positions are measured in units such as inches, millimeters, etc. This way is most useful on graphics screens.

So far, only "traditional" tabs have been described. These might be termed "left" tabs because the left edge of the text is placed at the tab stop. Other types of tabs have become popular (again) with the advent of word processors:

- **Right tabs** adjust the position of the text to the left of the tab stop so that its right edge is at the tab stop. Typically, all text back to the previous tab stop or the start of line is adjusted.

- **Decimal tabs** search for a decimal point (comma in Europe) and place that character at the tab stop. Again, typically, all text back to the previous tab stop or the start of line is adjusted. These tabs act as right tabs if the text does not contain a decimal point.

- **Centering tabs** center the preceding text between the current and previous tab stop.

Again, other variations are possible. Note that only the (traditional) left tabs can be implemented without some sort of look ahead.

Control Characters

Control characters are those that are not printing characters, a space, a newline, or a tab. ("Printing characters" means just that: if your system supported "extended" or "enhanced" character sets, then those characters may not count as control characters.) In addition, a word processor may store some information "in band." That information would be either interpreted or skipped on redisplay.

However, even in a word processor or on a system with an extended character set, there should be a way to view (and edit) a "pure" binary file. In order to view such a file, there must be a standard representation for non-printing characters.

One representation is to show such characters in octal ("\###") or hexadecimal ("\x##").

However, the most common representation – and possibly the most useful one – is to show such characters in caret notation (for a complete list of the caret notation, see Appendix E). The easiest way to define this notation is with a code excerpt:

```
void Caret(char c)
        {
        if (c == NL) {
                ...handle newlines...
                return;
```

```
        }
if (c == TAB) {
        ...handle tabs...
        return;
        }
if (c & 0x80) {
        Put_Char('~');
        c &= 0x7f;
        }
if (c < SP || c > '~') {
        Put_Char('^');
        c ^= '@'
        }
Put_Char(c);
}
```

When handling these multiple-character characters, your implementation must be consistent. For example, be sure that your cursor-positioning code takes the extra characters into account. Your implementation must also properly handle the case where such a character spans a line boundary. It doesn't matter which choice is made here (*i.e.*, the choice is between splitting the character at the boundary and moving the whole character to the next line), only that your implementation handle it consistently and correctly.

Proportionally Spaced Text

Once you have tabs and control characters down, displaying text in a proportionally spaced font is not too difficult. The main variation is that you no longer assume that all printing characters are the same width. Instead, you look up the width of each one as you display it. Actually, you can even support *kerning* by looking up each consecutive pair of characters to decide how to handle them.

The main "gotcha" in supporting proportionally spaced text is that one character no longer always exactly overwrites another on the screen. Thus, if you change an "m" to an "i", you have to figure out what to do with the extra width. Fortunately most displays that handle proportionally spaced text (mainly graphics displays) offer a high-performance primitive to scroll a region of the screen.

Attributes, Fonts, and Scripts

The next level of generality is the support of attributes, fonts, and scripts. Attributes include such modifiers as boldface, italics, underscoring, and superscripting. Fonts include the different typefaces such as Times Roman and Helvetica. Scripts include language families such as European and Japanese.

With these, your support can be as complex as you wish. Especially when it comes to scripts, your time and energy are going to give out long before you can provide support for all languages.

However, each one is fairly simple to handle. The first step is to store the attribute, font, and script information somewhere (see Chapter 5). The second step is to interpret that information.

Breaking Out Between Lines

As was mentioned in the procedure interface definitions, the redisplay process does not have to run to completion before editing resumes. Instead, it can get to a convenient spot and check for any user input. If input has arrived, the redisplay can be aborted ("broken out of") and the input processed. It is important to keep in mind that the purpose of redisplay is to provide feedback to the user. If the user has already typed something, there is no immediate need for the feedback. Hence, redisplay can be broken out of and then restarted after the user's input has been processed.

In order to keep the amount of state information to a minimum, it may make sense to not abort instantly, but instead to finish a current chunk of redisplay (say, a line) before checking for input. At the minimum, you must keep track of how far along you had proceeded, so that you don't wind up redisplaying your redisplayed text.

The presence of between-line breakout can affect how your redisplay is done. For example, if resources are tight, it may make sense to start by redisplaying the line that the point is on, then to go on to the other lines as you have time. In that way, the information that is most important to the user is the first to get updated.

Lest there be any doubt: between-line breakout is a very important feature and should only be left out of the very simplest implementations or those implementations that can complete even the most complex redisplay in under 100 msec.

Multiple Windows

Supporting multiple windows implies that the screen is divided into sections, with each section showing a possibly different buffer or part of the buffer. There are several ways that multiple windows can be supported:

- Don't support them. Instead, rely on the (presumed) operating system ability to run multiple instances of the editor. This is not desirable because the different instances may not be able to communicate quickly with each other. For example, you may not be able to "cut" from one window and "paste" into another.

- Support horizontal windows only. Horizontal windows occupy the entire width of the screen. This is a good and popular choice. It is not too difficult to implement, yet it provides a large chunk of the required functionality.

- Support both vertical and horizontal windows (tiled). (Better)
- Support arbitrary overlapped windows. (Best, and supported by many windowing packages)

The main thing to keep in mind when implementing multiple windows is that, when two or more windows contain the same text, changes made to one should be immediately reflected in the other.

If you do support multiple windows, you can implement status and prompt lines as buffers in themselves and simply fit them in as additional windows to be displayed. In this way, you no longer have to consider them as special cases.

Redisplay Itself

The basic role of redisplay is to ensure that all changes to the sub-editor are promptly reflected on the screen. Two major approaches are used by implementors to performing redisplay.

```
                -----------------
                | user commands |
                -----------------
                  /           \
                 /             \
                v               v
     --------------         -------------
     | sub-editor |         | redisplay |
     --------------         -------------
```

`First Approach`

The first approach is for the routines which are invoked by the user to tell the redisplay code exactly what they did (*e.g.*, "I deleted 5 characters from here"). This approach is not a very clean one and it is prone to error, as the same information must be given twice (once to the sub-editor and once to redisplay), and hence an implementation must handle the situation where the two sets of instructions are not consistent (*e.g.* the application tells the sub-editor to delete a line but tells redisplay to insert a line). This is an especially important consideration because we would like to encourage novice users to write their own commands. The extra effort of getting the redisplay correct might discourage such efforts.

```
                      -----------------
                      | user commands |
                      -----------------
                        /
                       /
                      v
           --------------         -------------
           | sub-editor |  <----> | redisplay |
           --------------         -------------

                       Second Approach
```

The second – and preferred – approach is to have the redisplay code communicate with the sub-editor to track the changes. This approach also has two methods of operation.

The first method (which might be called "sub-editor-driven") is to have the sub-editor calls communicate directly with the redisplay. For example, **Insert_Char** would make a call to display saying, "I inserted this character at this place." The second method (which might be called "redisplay-driven") is to have the redisplay operate on its own and ask the sub-editor for information.

The sub-editor-driven method appears to be simple to implement, but upon closer examination turns out to be quite complex. This complexity arises for several reasons.

First, the desirable operations for a sub-editor to offer (as shown in the sub-editor procedure interface definitions) do not match well to the available operations on displays. Hence, the redisplay code will have to perform this conversion. An example would be deleting a line. The code to perform the delete might be:

```
void Delete_Line(void)
        {
        mark_name beg;

        Find_First_In_Backward(NEWLINE);/* to start of line */
        if (Mark_Create(&beg) != OK) return;

        Find_First_In_Forward(NEWLINE); /* to end of line */
        Point_Move(1);                  /* skip over newline */

        Copy_Region(kills, beg);        /* save in kill buffer */
        Delete_Region(beg);             /* gone */

        Mark_Delete(beg);
        }
```

The sub-editor operation is "delete a region" and the region just happens to contain a line. *Somebody* has to examine the region to determine that it contains a line and that a "delete line" call to the display *might* be the correct one to use.

Second, the redisplay code will have to filter the sub-editor operations (and subsequent directives) that happen outside the window.

Third, every change made in the buffer does not necessarily imply a change in the display. For example, if the buffer contains the text:

```
Here is a line.
Here is a line.
Here is a line.
Here is a line.
```

and the first line is deleted, the following lines do not in fact change. That particular case may be rare, but the following happens fairly often:

```
Here is line 1.
Here is line 2.
Here is line 3.
Here is line 4.
```

In this case, the "Here is line " strings should not be redisplayed.

Fourth, the change might be no change at all. For example, the "lower case region" command applied to the text:

```
Most people believe the Unicorn to be a mythical animal.
```

might have in its inner loop:

```
Replace_Char(tolower(Get_Char()));
Point_Move(1);
```

This would have the effect of telling redisplay 56 times that a character had changed, when in fact only two of those characters were changed. One might argue that the **Replace_Char** routine could check to see whether the new character was in fact different before informing redisplay, however:

- You haven't gained anything, just changed who is doing the checking.
- The inner loop could have been written:

```
c = Get_Char();
Delete(1);
Insert_Char(tolower(c));
```

The most telling reason for not using the sub-editor-driven method, however, is more fundamental. The sub-editor's responsibility is to handle the buffer, not redisplay. It is the redisplay's responsibility to handle the redisplay function.

The algorithms presented in the remainder of this chapter illustrate the basic algorithms. They do not handle all possible error cases, nor do they handle many of the options listed above, such as variable width characters, line wrap, between-line breakout, and others.

The Framer

The framer is that part of the redisplay code that decides what part of the buffer will appear in the window. The redisplay code maintains two marks, one at the top of the window and the other at the bottom. The algorithm is fairly simple. Here it is:

```
int num_lines_window;   /* the number of lines in the window */
int point_pct;          /* the preferred percentage */

void Framer(void)
        {
        mark_name saved;
        location new_start_loc;
        int cnt;

                /* remember where we started */
        if (Mark_Create(&saved) != OK) {
                Fatal("can't create mark for redisplay");
                }

        Find_First_In_Backward(NEWLINE);
                /* count at most one window's worth of lines */
        for (cnt = 0; cnt < num_lines_window; cnt++) {

                        /* stop at the start of the window */
                if (Is_Point_At_Mark(top_of_window)) break;

                        /* stop at the start of the buffer */
                if (Compare_Locations(Buffer_Start, Point_Get) >= 0)
                        break;

                        /* record where a fresh screen would start,
                        just in case we need it */
                if (cnt == point_pct * num_lines_window)
                        new_start_loc = Point_Get();
```

```
                Point_Move(-1);
                Find_First_In_Backward(NEWLINE);
                }

                /* has the window moved? */
        if (cnt >= num_lines_window)
                Mark_Set(top_of_screen, new_start_loc);

        Point_To_Mark(saved);
        Mark_Delete(saved);
        }
```

In essence, the algorithm followed by this routine is: "so long as the point would still wind up in the window, leave the start of window unchanged. If the point would not wind up in the window, place it at the preferred percentage."

This version of the algorithm assumes that a buffer line will always occupy exactly one window line and that all buffer lines are the same height.

The Basic Algorithm

The basic redisplay algorithm is as follows:

```
int num_lines_window;   /* the number of lines in the window */
int num_chars_window;   /* the number of characters in the
                        window (its width) */
char window[MAX_ROWS][MAX_COLS];        /* window contents */

void Redisplay(void)
        {
        mark_name saved;
        int row;
        int col;
        int i;
        int point_row;
        int point_col;
        char c;

                /* remember where we started */
        if (Mark_Create(&saved) != OK) {
                Fatal("can't create mark for redisplay");
                }
```

```
                Framer();
                Point_To_Mark(top_of_window);

                        /* loop over the whole window */
                for (row = 0; row < num_lines_window; row++) {
                        for (col = 1; col < num_chars_window; col++) {

/* save the coordinates of the point so that we can put the cursor
there later */

                                if (Is_Point_At_Mark(saved)) {
                                        point_row = row;
                                        point_col = col;
                                        }

                                c = Get_Char();
                                if (c == NL) {  /* at a newline? */

/* check whether the rest of the window line is blank.  if it is not, clear it */
                                        for (i = col; i < num_chars_window; i++) {
                                                if (window[row][i] != SP) {
                                                        Set_Cursor(row, i);
                                                        CLEOL();
                                                        memset(&window[row][i], SP,
                                                                num_chars_window - i);
                                                        }
                                                }
                                        }

/* no newline, so has there been a change in the sub-editor? */

                        else if (window[row][col] != c) {
                                Set_Cursor(row, col);
                                Put_Char(c);
                                window[row][col] = c;
                                }
                        Point_Move(1);
                        }
                }

/* clean up */
```

```
Mark_To_Point(bottom_of_window);
Set_Cursor(pointrow, pointcol);

Point_To_Mark(saved);
Mark_Delete(saved);
}
```

The preceeding code shows your basic, garden variety redisplay algorithm. It will work on any screen that supports cursor positioning (the **CLEOL** call can be simulated by sending Space characters). It will work quite well on communications channels running at 4800 bps or over. Its only memory requirements are an array large enough to hold the window (typically 1920 characters). There are no special redisplay "hooks" in the sub-editor management code.

This algorithm is sufficient (and nearly optimal) in those cases where CPU and memory are plentiful and the screen does not perform insert/delete line or character operations. If memory is tight, the algorithm can be modified to only retain a complete copy of the current line. If you must be prepared to emulate the **CLEOL** operation, it may be worthwhile to record the last non-blank column in each screen line. Doing so minimizes the number of Space characters that must be sent.

Sub-Editor Interaction

The basic algorithm can be sped up tremendously if some redisplay-specific hooks are placed into the sub-editor. There are a number of different ways that the hooks can be introduced. All of these methods track the changes made to the buffer in one way or another.

The first way is to keep a separate modification flag that tells whether any changes were made to the buffer since the last redisplay. If no changes were made, then redisplay will consist of either a simple cursor motion or a complete screen regeneration.

The seond way, and much more useful, is to keep the modification flag on a per-window-line basis. A general interface to accomplish this that works with all sub-editor implementation schemes is to define a third type of mark, called a *window mark*. This mark has a flag associated with it. There is one window mark for each line in the window. Just after a redisplay has been completed, all the flags for all window marks are clear. Each time the sub-editor changes any of the buffers' contents, it sets the flag on the window mark that is located before and closest to the change. The redisplay code can examine the flags. Only window lines that have their corresponding window marks set need to be examined closely during redisplay.

Note that window marks need not be located at the start of a buffer line. If lines are being wrapped, one will be at each wrap point. If horizontal scrolling is being performed, one may be at the start of the buffer line and another at the right edge of the window. In this way, changes made to the right of the window won't cause the redisplay code to examine unchanged text.

This interaction is easy to define and implement in the sub-editor. It is inexpensive to implement as the marks have to be examined for updating anyway. It is also highly effective at reducing CPU overhead, as most commands change only a single line. And, although redisplay has to examine the flags for every line, most of the time only one or two will show changes..

A third way is to associate a unique identifier with each window mark instead of a flag. This identifier would be changed by the sub-editor whenever the associated text changes (*i.e.*, instead of setting a flag it changes the unique identifier). Typically, the identifier will be a 32-bit integer. Whenever an identifier is required, the current value of the integer is used and the integer is incremented.

The only problem that can arise with using unique identifiers is if a unique identifier is not in fact unique. This problem can arise if all 2^32 unique identifiers are consumed before all lines in the window have changed.

Some sub-editors that use the linked-line scheme use the addresses of line structures as the unique identifiers. While doing so is space efficient, the sub-editor must ensure that if a line is freed, the address is not re-used until all windows have been completely redisplayed.

Finaly, there is one more flag that can help redisplay a great deal. This flag is only useful if the point is located at the end of a buffer line. The flag would say whether any buffer modification other than "insert one or more characters" has been performed. If the flag says not, all that redisplay needs to do is to output those characters. As this situation is very common, it can save a significant amount of computation.

The Advanced Algorithm

The advanced-redisplay algorithm has two improvements over the basic algorithm. First, it provides a way of efficiently taking advantage of the insert/delete line and character functions which are supplied with many screens. Second, it provides a low CPU overhead way of performing a redisplay on basic displays.

The basic idea used by this algorithm is to assign a unique identifier to each window line. (See the preceding section.) When the redisplay encounters a modified line (the unique identifiers don't match), it performs a pattern match on the unique identifiers for the remainder of the window. It then uses the information derived from that match to determine the best sequence of insert/delete line commands to issue to the screen.

In more detail, this algorithm loops over the window lines, checking each saved unique identifier against the current identifier returned by the sub-editor. If they match, no work needs to be done and the algorithm proceeds to the next line. If they don't match, it can be for one of three reasons.

The first reason could be that an additional line or lines were inserted between the two window lines. This condition is detected by comparing the window-line unique identifier against the rest of the unique identifiers returned by the sub-editor and finding a match. (Remember that the window-line unique identifiers are the unique identifiers returned by the sub-editor one redisplay iteration ago.) The insertion case is where we once had lines:

```
AB
```

and now have:

```
ACB
```

We determine how many lines are in "C" (because we know how far down we had to go to find a match) and tell the screen to insert that many lines. (If there is information after this window on the screen, you will first have to delete that many lines from the end of the window.)

The second reason could be that a line (or lines) was deleted. This condition is detected by comparing the unique identifier returned by the sub-editor for the next line against the unique identifiers of the rest of the window lines and finding a match. The deletion case is where we once had lines:

```
ABC
```

and now have:

```
AC
```

We determine how many lines are in "B" (because we know how far down we had to go to find a match) and tell the screen to delete that many lines. (If there is information after this window on the screen, you will eventually have to insert that many lines at the end of the window.)

The third reason could be that the line was changed. This condition is detected by comparing the unique identifiers of the following window lines against the unique identifiers returned by the sub-editor. This case is either where we once had lines:

```
ABC
```

and now have:

```
ADC
```

or:

```
ADE
```

In other words, neither the insertion condition nor deletion condition was met. Knowing now that a line has been changed, the next step is to determine exactly how the line has changed.

The algorithm starts by comparing the buffer line against the window line and determining how many leading characters are in common. (If the whole line is common, no changes need to be made to the screen and the algorithm stops.) For example, if the window line is:

```
abcdef
```

and the buffer line is:

```
abcxef
```

the two have three characters in common from the start.

The next step is to repeat the comparison, but work backwards starting from the end. The example strings have two characters in common from the end.

The third step is to compare the line lengths. If the two lines are the same length, only the changed part in the middle needs to be updated on the screen. In the example strings, the lengths are the same (six6). This optimization can be done even on a basic display.

If the two lines are not the same length (for example, the buffer line is "abcxyzef"), the characters in the window line that are replaced by characters in the buffer line can be rewritten (in the example, the "x" replaces the "d"), then the requisite number of characters can either be inserted or deleted and the remainder of the changes written (insert two characters, "yz"). If there is no common text at the end of the line and the buffer line is shorter than the window line, a **CLEOL** call can be used instead of deleting characters.

Line wrap can pose a problem. The window and buffer lines may have no end text in common, and yet an insert or delete character operation might be the appropriate one. For example, consider the case where the window width is six characters, the window line is "abcdef", and the buffer line is "abcxdef". Here, the buffer line will ultimately become two window lines, "abcxde" and "f". This case is detected by having no common portion at the end and noticing that the line wraps. A more complicated matching process can detect the situation and appropriate action can be taken.

This entire section considered only the (admittedly very common) case where line and character insertions and deletions were only made in one place. It is very reasonable and appropriate to use more general pattern-matching techniques to properly optimize multiple insertions and deletions (Miller 1987).

Redisplay for Memory-Mapped Displays

Redisplay for memory-mapped displays boils down to one of three cases. Each case is relatively simple.

First is the case where both reading from and writing to the screen causes flicker. The solution is to use the basic redisplay algorithm.

Second is the case where reading does not cause flicker but writing does. The solution is to use the basic redisplay scheme, but change it to use the actual window memory for storing the window array.

In the third case, neither reading nor writing causes flicker. On each redisplay cycle, merely copy the buffer text into window memory, not forgetting to process new lines, etc., as needed.

Questions to Probe Your Understanding

Define a set of editor procedures to handle vertical windows. (Easy) Extend that set to handle overlapping windows. (Medium)

Implement the procedures that you just defined. (Hard)

Define a representations for the private data types mentioned here (function keys, attributes, command times). (one is Easy, all together are Medium)

Identify the places where left-to-right, top-to-bottom biases are built into the interface definitions. English and European languages have this bias. (Easy)

Rework the interface definitions to remove this bias and to be able to handle all eight (yes, eight) combinations of directions. (Medium)

Outlining is popular these days. "Outlining" is the ability to selectively skip over parts of the text during redisplay. For example, one level might only display the chapter and section titles. Another level might include all titles and the first sentence of each paragraph. Identify how adding outlining would affect redisplay. (Medium)

Identify how the editor's redisplay algorithm changes when it makes use of the UNIX *curses* library. (Hard)

How do the presence of ligatures and contextual forms used by non-Roman languages affect cursor motion? (Medium)

What modifications to the redisplay algorithm are required to handle ligatures or contextual forms used by non-Roman languages? (Hard)

Chapter 8: User-Oriented Commands: The Command Loop

He left it dead, and with its head
He went galumphing back.

The previous two chapters described a way of dividing an implementation into parts and covered the internal sub-editor and redisplay. This chapter describes the last part: the user-oriented commands. This last part is what gives the editor its "feel." It determines the overall command structure (the syntax) and what each of the commands does (the semantics).

Command structure is a large enough topic to be divided into two chapters. This chapter describes *how* to implement the command structure. The next chapter covers command set design issues. It thus describes *what* commands should be implemented.

The Core Loop: Read, Evaluate, Print

The command loop is built around a basic core. This core reads in commands, evaluates (or executes) them, and prints the results.

Reading commands is the process of accepting user input and determining what operations the user wishes to perform.

Evaluating commands is the process of carrying out the user's wishes. In general, this is done by executing a series of sub-editor calls.

Printing the results is the redisplay.

The core loop looks like this:

```
char c;

while (1) {
        c = Key_Get();
        if (Evaluate(c)) break;
        Redisplay();
        }
```

This loop accepts user input (a single character), evaluates it, exits (if the user has requested to quit the editor, causing **Evaluate** to return True), and invokes **Redisplay**. This, like all program examples in this chapter, is a simplified version of just one of the many ways you can implement these functions. They are meant as examples, not as limits.

The Evaluate Procedure

```
FLAG Evaluate(char c)
        {
        FLAG is_exit = FALSE;
        FLAG is_arg = FALSE;
        int arg = 1;

        while (!(*commands[c])(&is_arg, &arg, &is_exit, c)) {
                c = Key_Get();
                Redisplay();
                }
        return(is_exit);
        }
```

This is the core of the **Evaluate** routine. The **is_exit** flag records whether the command is one to exit the editor. The **is_arg** flag records whether the user has specified a repeat-count argument. The **arg** variable records the repeat-count argument.

This routine – and the editor implementation – is built around a set of command dispatch tables. Each table is an array of pointers to procedures, indexed by command characters. Thus, the element:

```
commands['a']
```

would specify the procedure to handle the command designated by the "a" character. These procedures all have the same interface. This interface is:

```
FLAG Command_Procedure(FLAG *is_argptr, int *argptr,
        FLAG *is_exitptr, char c);
```

The first three arguments are *pointers* to the three state variables. They are pointers instead of the values so that the command procedures can alter their values. The fourth argument is the character that is used to invoke the procedure. The procedure returns True if the command has completed, or False if the command is incomplete.

The reasons for selecting this interface will be made clear through four sample command procedures: "move by character," "insert character," "second-level dispatch," and "accept an argument."

Move by a Character

This procedure moves forward by *arg* characters if *arg* is positive or backward by *arg* characters if *arg* is negative. It looks like this:

```
FLAG Move_by_Character(FLAG *is_argptr, int *argptr,
         FLAG *is_exitptr, char c)
        {
        Point_Move(*argptr);
        return(TRUE);
        }
```

Insert a Character

This procedure inserts *arg* copies of the character used to invoke it. If *arg* is negative, its absolute value is used. It looks like this:

```
FLAG Insert_A_Character(FLAG *is_argptr, int *argptr,
         FLAG *is_exitptr, char c)
        {
        int arg = *argptr;

        if (arg < 0) arg = -arg;
        while (arg-- > 0) Insert_Char(c);
        return(TRUE);
        }
```

Second-Level Dispatch

This procedure doesn't implement a command itself. Rather, it accepts a second character and uses that to select a command from a second dispatch table. In this case, the "^X" character will be used as the dispatch.

```
FLAG Ctrl_X_Dispatch(FLAG *is_argptr, int *argptr,
         FLAG *is_exitptr, char c)
        {
        char c;

        c = Delayed_Display(CTRL_X_DELAY, "^X ");
        return(*ctrl_x_commands[c])(is_argptr, argptr,
                is_exitptr, c));
        }
```

The **Delayed_Display** routine waits for a character and returns it. If more than a specified amount of time passes with no input, the prompt string is displayed.

Note that this routine passes the arguments and exit status to and from the second-level command routine.

Accept an Argument

Again, this procedure doesn't implement a command itself. Rather, it performs one step of accepting a numeric argument. For the purposes of this example, we will assume that all digit characters specify an argument and do not insert themselves.

```
FLAG Argument(FLAG *is_argptr, int *argptr,
         FLAG *is_exitptr, char c)
        {
        if (!*is_argptr) {      /* no arg yet */
                *is_argptr = TRUE;
                *argptr = 0;
                }
        *argptr = *argptr * 10 + c - '0';
        return(FALSE);
        }
```

This routine is the first one that does not completely execute the command. Rather, it modifies the state information that is passed to the command procedure itself.

(Note: this routine does *not* implement the Emacs "universal argument" command, but is a simplified version for the purposes of this example only. It actually performes *vi*-style argument handling.)

Philosophy

The loop as described puts few (theoretical) restrictions on the command syntax. Each character, in its raw form, is mapped to a procedure which is in turn evaluated. State information is passed to and from this procedure, which can either update the state information, perform an operation, or both. Arbitrary syntax and semantics can be implemented with this base.

In theory, a syntax of commands being words (*e.g.*, "delete," "move," etc.) could be implemented in this structure by having either a large number of dispatch tables (and thus implementing a symbol-state table architecture) or a procedure which parses the syntax of the command via conditional statements. If you really want to do one of these, you will want to invent your own – different – internal structure.

A Minimalist Command Set Design

Consider the thought that every character that is typed at the keyboard causes a procedure to be executed. The first conclusion that results is that it is silly to type "insert x" or anything like that when you want "x" to be inserted. As this is a very common operation, it makes more sense to bind the key "x" to an "InsertX" function (or, more probably, the **Insert_A_Character** procedure just defined).

This architecture binds all of the straight, printing ASCII characters to commands that insert the character. The remaining things that can be entered from most keyboards are the control characters, the delete key, and the break key. These could be bound to functions that implement a complex syntax, but why bother? It is not too difficult for users to learn even a large number of key bindings, so let us bind the control keys directly to useful functions. For example, ˆF could be "move forward a character," ˆD could be "delete the following character," and so forth. Note that the "break" key does not have an ASCII value and is therefore difficult to use without writing operating system-specific code.

Thirty-three functions (the 32 control characters plus the Delete character) are not enough for even the commonly used functions. Thus, some of the keys should be bound to functions which temporarily rebind the dispatch table. For each of these rebinding functions, 128 new functions are made available (there is no reason for the printing characters in those second-level tables to be bound to "self insert").

Thus, even though we began with a structure for the command loop that did not impose any constraints on the syntax of commands (and thus was as general as possible), we arrived at a specific syntax for commands. This syntax is to bind the printing characters to "self insert," bind the control characters to a mixture of useful functions and second-level dispatch tables, and to have three or four alternate dispatch tables (enough to supply many hundreds of commands). Thus, commands are rarely more than two keystrokes long. The price that is paid for this brevity is a possibly longer time learning to use the editor effectively.

Note that most of the increased time spent learning the editor is *not* from the brevity of the commands, but because there are more commands to learn. Given a "conventional" editor of some other command set design (*e.g.*, insert/replace modes or command lines) and an equivalent subset of this "minimalist" editor, learning times will probably be comparable when the same number of commands from each are covered (assuming sensible command assignments in both cases).

Errors

There are two main types of errors: internal and external. Internal errors are those that occur in the editor itself. Examples are a subscript being out of range and division by zero. External errors are those that are caused by the user. Examples of these are an attempt to delete off the end of the buffer. There are also "non-error" errors, such as a normal exit condition. Errors can be detected both from within the editor and from outside the editor (for example, by the operating system).

Internal Errors

Internal errors will be considered first. These errors cause an immediate exit to the operating system with no questions asked and no delays tolerated. They will be internally generated by such things as arithmetic overflows and bad subscripts. (While the editor might catch and process some of these, it will not in general process them all. This section only discusses the non-processed ones.) These errors are unpredictable and the state of the editor should remain intact.

The user should also be able to signal such an error to abort out of the editor. He or she might want to do this signaling because of a problem with the editor itself (*e.g.*, infinite loop) or because he or she wants to do something else (*e.g.*, suspend this process and do another task). This signaling is usually done with the help of the operating system. In any case, the precise state of the editor should be retained so that it can be resumed exactly where it left off. Most operating systems have some facility for doing this; they differ principally in the freedom of action that they allow before losing the state. This freedom ranges from nothing to doing arbitrarily many other things.

At the user's discretion, the editor should be restartable either from exactly where it left off or at a safe restart point. This point is ordinarily a portion of the editor which recovers the buffers and other current state information and then resumes the command loop. Note that in many implementations, the editor must perform actions both on the process suspension and when it resumes. These actions must handle saving and restoring the state, restoring and saving the display modes, and taking note of any changes in the environment, such as a window resizing.

External Errors

External errors are principally user errors. The action ordinarily taken is the display of an error message and a return to command level. The implementation of this level of recovery is built into the procedures which implement the commands.

There is a variation of external errors which are generated manually by the user. Typically, these involve backing out of an undesired state (*e.g.*, the unwanted invoking of a dispatch table rebinding or aborting an undesired argument). The ˆG character has often been used for this purpose. In this case, the procedures will know that this character has been typed and will implement the back-out protocol.

Exiting

Finally, provisions to exit the editor must be made. These provisions often take the form of a flag variable such as the *is_exit* variable described earlier.

Note that various other uses might be multiplexed onto this flag, signifying varying levels of "exiting." For example, one level could be used by buffer switching in order to rebind the dispatch tables (see the section on modes later in this chapter). Alternatively, the different functions could use multiple flag variables.

Ordinary exiting involves several types of processing. The editor might ask the user what to do with buffers that have been modified but not written out. If, as is ordinarily assumed, the state of the editor is preserved across invocations, the state must be saved. If not, it must be sure that all memory is deallocated. Finally, the user's environment should be restored as it was found. This implies such varied things as cleaning up the stack, closing files, deallocating unneeded storage, and resetting terminal parameters.

Arguments

Arguments are specified by the user to modify the behavior of a function. The Emacs argument mechanism will be described as an example of three diverse ways in which arguments are obtained.

There are three standard argument types. First are numeric (prefix) arguments. These are invoked by a string of functions (which are in turn invoked by characters typed before the "actual" command character) and are an example of using the key/function binding to implement a more complicated syntax. Next are string (suffix) arguments. When obtaining a string argument, the editor is invoked recursively on an argument buffer, and upon return from the recursive invocation the contents of that buffer are given to the requesting procedure. Last are positional arguments. These are the internal variables of the editor.

Numeric (Prefix) Arguments

Prefix arguments are entered before the command whose behavior the arguments are modifying, thus, their syntax does not depend upon the command. The interpretation of prefix arguments can vary from command to command. Emac type editors limit these arguments to numeric values.

Ordinarily, commands will have an internal variable available to them named something like *arg*, and it will have a value of one. Prefix arguments allow the user to change that value to any other positive or negative integer. It is useful to provide a mechanism for command procedures to determine whether an argument has been given at all. This mechanism allows the procedures to handle the default case where no arguments are supplied differently than the case where an argument is supplied.

Each command uses arguments for different, but related, purposes.

The first purpose is to specify a repeat count for a command. Thus, specifying an argument of "12" to the "forward character" command would cause the command to move forward 12 characters.

The second purpose is to tell a command to use a specific value. For example, it doesn't make sense to say "move to the end of the buffer" 12 times. Instead, that command might interpret its argument as a line number and move to the specified line of the buffer. In this case, the "default" value would be the (end of the) last line.

An Emacs-type text editor uses the ˆU character as the "universal argument" function. It can be used in either of two ways. ˆU *command* means to supply an argument of "4" to command. Adding another ˆU means to multiply the current argument by four. Thus, ˆU ˆU ˆU *command* means to supply an argument of 64 to the command. The factor of 4 was selected because 5 is too large (1, 5, 25, 125 goes up too fast) and, while 3 might have better spacing (1, 3, 9, 27, 81, 243), the powers of 4 are known by all people who are likely to be around computers. In addition, on a 24 x 80 display, 64 is about the number of characters per line and 16 is 2/3 of the screen height.

The other use is to specify a value exactly. ˆU *number command* means to supply an argument of *number* to the command. For example, ˆU - 1 4 7 *command* means to supply an argument of -147 to the command. The ˆU in this case serves as an "escape" to logically rebind the digit and "-" keys. If you want to supply an argument to the commands normally invoked by the digit and '-' characters, you use the quote command, located on ˆQ.

On some terminals, there are two sets of numeric keys. One set is across the top row and always sends the ASCII code for the corresponding digit character. Another set may form a numeric pad and its keys can be configured to send either the ASCII codes for the digit characters or different codes. In this case, these "other numbers" can be bound directly to functions that set up the implied arguments and the initial ˆU is not needed.

String (Suffix) Arguments

Numeric arguments are made available in the same way to all commands. Suffix arguments, however, must be explicitly requested by the commands that use them. A command may also request multiple suffix arguments. Most suffix arguments are for strings, not numbers.

The program notifies the user of the string argument by displaying a prompt. This prompt indicates the type of argument that is requested. The user responds by entering the value up through and including a terminating character. The command then proceeds to execute, using the value in whatever way is appropriate.

The following points should be taken into consideration regarding string arguments.

First, the prompt should clearly state what is being asked for – for example, "Name of the file to be read."

Second, the key or key sequence used to terminate the end of the string should be able to vary and should be indicated in the prompt – for example, "Name of the file to be read (Return): " or "String to search for (ESC): " There should be a way to cleanly abort out of the prompt and its requesting command. This should be the same command used for the "abort" command (*e.g.*, ˆG).

Third, in order to facilitate the abort process, the command should *first* ask for all user input, and only *then* perform any actions. This organization means that any abort will result in no effect rather than leave inconsistent state information.

Each command that requests a string prompt should provide a default value for the prompt. This value should be used if the user enters a null response. The value should be the program's best guess about what value the user would most likely want to enter. If no other guess is available, the last value entered should be used.

Here are some examples of string arguments:

- Search string: Ask for a string and search for the next occurrence of it in the buffer. If the user enters a null string, use the same string that was last entered.
- Write file: Ask for a string and, using it as a file name, write the contents of the buffer to the specified file. If the user enters a null string, use the current file name associated with the buffer.
- Change buffer: Ask for a string and switch to the buffer whose name is the user's response. If the user enters a null string, use the buffer that was last the current one (*i.e.*, the one that the user was in before the one that the user is in now). Note that this default may *not* be the one whose name was last entered.

Here are some example prompts:

- Name of the file to write to (Return, default /home/fin/test):
- String to search forward for (ESC):
- Name of the buffer to switch to (ˆM, default chapter5):

While these examples were requesting a character string, this need not always be the case. For example, to enter numeric values, the requesting procedure merely has to convert the read-in character string to a numeric value. An example of such a command would be a "go to line" command.

One way to implement the routine that accepts string arguments is to use a variation of the **Get_Line** routine defined in the Introduction. However, a better way to implement this routine is to create an argument buffer in a new window, display a prompt, and call the editor recursively with that as the current buffer. By following this scheme, the full power of the editor is available to correct typing mistakes or otherwise make the entry process easier. It has the additional advantage of not creating a new "mode": the user is free to continue editing while responding to the prompt.

Further, the full power of the editor can be brought to bear on a problem. For example, suppose that someone sends you a mail message that says "the answer is in file X." While reading the mail message, you give the "find file" command. This command prompts you to enter a file name. You switch buffers (from the prompt buffer to the mail message buffer), copy the file name, switch back, and paste it into the prompt, then type the prompter terminator. *Voila*! A fully integrated, modeless environment.

Finally, the prompts need not be "lifeless" and "passive." A passive prompt just accumulates the input until complete, then passes it back as a block. It has no interaction. A "lively" and "active" prompt offers interaction with the user. For example:

- **Searching**: The search can be *incremental*, with the search proceeding as the user types.

- **File names**: The program can offer *file name completion*, where the user can enter a prefix, press a key, and the program fills in as much of the file name as possible. A different key might display a list of all file names that match what has already been typed.

Positional Arguments

Positional arguments are not directly specifiable by the user. They are the editor's internal state variables. Such variables include both those required by the editor (*e.g.*, the length of the buffer, the locations of the point and the mark, etc.) and those which have a specialized purpose (*e.g.*, the current value of the right-hand margin, the tab spacing, etc.).

Often these values are used in unusual ways. For example, the horizontal position (column) of the point can often be a more pleasant way of specifying a value than entering a number. The user can indicate that "this is where I want the right margin to be" instead of having to count characters to get a number.

A specialized positional argument is the *region*. This is the range of text delimited by the point and the mark. By convention, it does not matter whether the point or the mark is placed earlier in the buffer.

Selection Arguments

The use of graphical input devices opens up new ways of issuing commands and specifying arguments. For example, the cursor can be moved by a graphical input device as well as the more traditional point-motion commands. In addition, a region can be specified by a "click and drag" operation (or whatever sequence is used by the operating system).

Rebinding

Binding is the act of connecting a name and a meaning, *rebinding* the act of changing the binding. In the case of editors, there are two different levels that binding (and rebinding) can occur on.

The first is at the key level. Binding in this case mean attaching an operation to a key. These bindings are often implemented by means of a dispatch table.

The second is at the function level. Binding in this case means attaching a procedure to an operation. Again, these bindings are often implemented by means of a dispatch table.

For example, the alphabetic keys may be bound to the "insert" operation. This operation, in turn, can be bound to a variety of procedures:

- The basic "insert a character" procedure.
- The basic procedure, but one that saves a copy of the buffer every so often.
- The basic procedure, but one that performs word wrap (by inserting hard newlines, not in redisplay). This is often called something like "fill mode."
- A different basic procedure, say one that performs replacement (overwrite) instead of insertion.

Implementations can perform at one of two levels of rebinding: static and dynamic. Static rebinding is when the new procedure is known about at the time that the editor is invoked. All implementations can perform this level of rebinding. Dynamic rebinding is possible when the new procedure can be defined after the editor is invoked. Unless otherwise stated, this discussion assumes dynamic rebinding.

To a first approximation, editors that are written in compiled languages (*e.g.*, C and Pascal) can only perform static rebinding, and editors that are written in interpreted languages (*e.g.*, Lisp) can also perform dynamic rebinding. Dynamic linking, however, allows compiled editors to include new procedures at run time, and so this distinction is not always a proper one to make. Dynamic bindings are also possible when a compiled language is used to implement an interpreted language, which in turn implements at least the user command portion of the editor.

Rebinding Keys

The process of key rebinding is relatively simple and is done essentially the same way in all implementations. A set of dispatch tables is used to map keys (represented by their ASCII values) to their respective functions.

In languages such as C and Lisp, the table can contain the pointer to the procedures themselves. In less powerful languages such as Fortran and Pascal, the dispatch table branches to a different part of the same routine that contains the table. There, the procedure call is made. In languages that supply it, a case statement can be used instead of the n-way branch.

All of these command procedures have the same formal parameters, and so they can all be invoked with the same calling sequence. Thus, the C and Lisp direct invocations can work properly. Note also that simple commands do not have to have a separate procedure assigned to them, but the code to execute them can be placed in-line in place of a call (where the case-statement equivalent is used). Making this substitution loses some potential flexibility.

Rebinding Functions

Dynamic rebinding is ordinarily a language-supplied feature and so it will not be discussed in depth. Two comments will, however, be made on how to simulate it.

If the underlying operating system has dynamic linking (*e.g.*, Multics, OS/2, and some new UNIX systems), a procedure may be rebound at run time. Dynamic linking is a way of linking procedures together in which the actual link is not made until the procedure is about to be executed. At that time, the procedure is located in the file system and brought into memory. The link may either be left alone, in which case the next call will have the procedure re-located (a relatively expensive process) or it may be *snapped.* Snapping a link is the process of converting the general call instruction (which is kept in a special, writeable part of the program) into a call instruction to the appropriate address. If a link is snapped, it must be explicitly unsnapped before any rebinding is done.

If the operating system does not support dynamic linking, you might choose to simulate it manually. Such a process is complex, and some thought will have to be given to the desirability of rebinding functions. The process is tantamount to explicit overlaying.

This all has a straightforward bearing on rebinding functions. Rebinding a function involves changing the definition of the procedure that is invoked by referencing it. What has been discussed are ways of changing such a procedure definition. Note that if the code to execute a function is inserted in-line in the basic editor, it cannot be rebound by any of these methods.

If dynamic linking is not available and is not feasible to simulate, there is still one way out. This way will only provide static rebinding. Instead of just using one dispatch table which indicates a procedure to be called directly, use two. Use use the first table to map from keys to the operation to be performed (*e.g.*, ˆF is mapped to "moving forward one character") and the second table to map from the operation to be performed to a procedure that will perform it (*e.g.*, "moving forward one character" is mapped to the **Forward_Char** procedure).

Modes

A *mode* is a collection of command rebindings. Modes can be invoked implicitly, explicitly, or automatically.

An *implicitly* invoked mode is one that is not visible to the user. Implicit modes are used to support large, infrequently used commands. For example, suppose that you had an editor command that played the game Adventure. You probably wouldn't want the code for that command to be occupying resources whenever you were using your editor for editing. However, you might still want to make your "adventure" command available at all times. In this case, you would use an implicit mode. The "adventure" command would then take these steps:

- Load the modules that implement the command.
- Rebind the key that invoked the "adventure" command to run the new code.
- Run the code the first time.

From now on, whenever the user gives the "adventure" command, the editor will directly execute that code.

An *explicitly* invoked mode is one that the user asks to use. Examples of such modes are "auto fill" mode, "auto save" mode, and alternate command sets. The common element is that the user gives a command, knowing that that command itself has no function other than to persistently alter the key bindings.

An *automatically* invoked mode is one which the implementation determines is appropriate to invoke, based on a command given by the user that "appeared" to do something else.

One example of an automatically invoked mode is a language mode (for example, a "C" mode). This mode will automatically be invoked whenever the user edits a C source file (by convention, one whose name ends in ".c" or ".h"). Such a mode might do the following:

- Rebind the internal variable that identifies which characters are legal in tokens (*i.e.*, variable names) to also include the "_" character, which can occur within C names. This change would make the **Forward_Word** function treat a C variable name as a word.

- Similarly rebind the sentence and paragraph operations to operate on statements and language blocks.

- Rebind the ";" key to be an "electric" semicolon so that typing a ";" to finish one statement would cause the editor to determine and insert the appropriate indentation.

- Similarly rebind the Tab, Return, and Line Feed keys.

- Replace the "fill" or "reformat" paragraph command with one that "prettyprints" the current language block.

And so forth. Another example of an automatically invoked mode is the specialized mode that the editor places you in when executing such commands as "help," "read mail," and "view a directory." In these commands, the user is effectively placed in a specialized application that shares as much as possible with the regular editor commands, but does have its own extra commands. For example, in the "view a directory" application, the "d" key might delete a file, the "r" key might rename a file, and so forth. However, the "buffer" or "window" switch command should still be available so that the user can perform other editing while the special application is active.

Modes and Dynamic Rebinding

The function rebindings that are commonly done by an editor are known in advance and so they can be done by any implementation (see the preceding section for a discussion of the difficulties involved in function rebinding). Fully dynamic rebinding (the new definition of the procedure is not known until run time) is desirable for several reasons:

- Debugging is greatly eased if the trial-and-error cycle time is reduced by not having to compile and link the whole editor each time. Instead, only one function has to be recompiled and linked.

- Space savings are achieved if unneeded modes and autoloaded single functions are not brought into memory until called.

- If the editor is implemented in an interpreted language, users can develop their own functions relatively easily. Such "sideline" development is advantageous because it allows many people to develop useful programs. Thus, the editor can be specialized in many more ways than any reasonable support group could ever implement on its own. Implementation in an interpreted language also encourages tailoring the editor to a user's own taste, enhancing his or her productivity.

Implementing Modes

Modes are defined on a per-buffer basis and so an implementation must provide for changing these bindings as the current buffer is switched. The general technique for doing this is to have a set of default bindings for the editor, a set of current bindings for each buffer, and a set of procedures that can be invoked to change the former into the latter. When a buffer switch is made, the current bindings are used to dispatch all commands.

Whenever a change to the mode list is made – especially one that removes a mode – the editor must initialize the current bindings to the default bindings, then invoke each mode procedure in turn to make its changes.

Changing Your Mind

This section discusses the methods used to help users who want to change their minds about an editing command.

Command Set Design

By far the most effective step that you can take is to design the command set to minimize both state variables and changes of perspective. Such proper design is far more effective than any other tool. However, as these topics are covered in the next chapter and in Chapter 1, they won't be discussed here.

Kill Ring

The most basic way in which a user can change his or her mind is to delete something, then say "oops, I didn't want to delete that." After all, if the user inserted extra text, deleting it is easy and straightforward. However, retyping accidentally deleted text is in general neither easy nor straightforward. Hence, an important feature to provide is the ability to save that text for the user. This feature can be added as a single-level save (referred to as the *kill buffer*) or a multiple-level save (the *kill ring*).

As an extra benefit, once the deleted text is saved, that feature can be used for "cut and paste" operations. In addition, the saved text can be tied into the system "clipboard" or similar facility.

An Emacs-type editor records multiple text-deletions in a "kill ring" (for historical reasons, commands that save the deleted text were called "kill" commands). Some small but fixed number of the successive deletions are stored together. The "yank" or "paste" command retrieves the last such deletion (inserting it at the point). Alternate "yank" commands are available that cycle through the kill ring, re-deleting the last-yanked text and replacing it with the next item. (A ring is better than a single kill buffer because it can store multiple deletions. It is superior to a stack of separate buffers because of the ease with which various "undeletions" can be tried out.)

It is easy to implement such a kill ring. A buffer is designated as the one to hold the deleted text and the commands that perform the deletion simply copy the text that they are about to delete to that buffer before performing the actual deletion. There are two fine points to the implementation.

First, successive deletion commands should add to the current deletion, not create a new one. Thus, the Emacs commands:

```
^U ^[ d
```

which deletes the next four words, should have exactly the same effect as:

```
^[ d ^[ d ^[ d ^[ d
```

i.e., four successive "delete word" commands. In both cases, all four words should be part of the same deletion.

Second, your implementation must take care to delete properly. Deleting *following* items (characters, words, sentences, etc.) should add to the *end* of the deleted text. Deleting *previous* items should add to the *beginning* of the deleted text. Not coincidentally, all deletion commands in the command set of Emacs-type editors are of the form "delete following..." or "delete previous..." with the exception of the "delete current line" command. This command must take the text from the point to the end of the line and add it to the end of the deleted text, and take the text from the point to the beginning of the line and add it to the start.

Undo

The "kill ring" approach requires explicit support from the user commands and provides considerable power, yet there are many ways in which it does not make it easy for a user to change his or her mind. An "undo" facility is the most general way that you help a user when he or she changes his or her mind.

In principle, an undo facility provides a mechanism for reversing changes made by the user. These effects can be as simple as moving the point or inserting or deleting a character, or as complex as a file write or global replace.

Note, however, that you may still want to provide the kill ring, as it offers both one intuitive (if limited) type of undo, as well as operations that undo cannot perform. For example, you cannot implement "cut and paste" with undo, except for the limited case where you only want to paste the text exactly where you cut it from!

Unlike the kill ring, which requires explicit support in the user commands, the best place to provide support for undo is in the sub-editor interface. Note that this is in the interface, not the sub-editor itself, although in general the undo facility will work closely with the sub-editor.

The support works like this: each sub-editor procedure that makes any change to a state variable or the buffer first makes a note of what is to be changed, then records the pre-change value, then finally makes the change. For example, the **Point_Set** routine would record "the point is about to change," and the old (current) location of the point. It would then change the point's location. Note that some procedures (such as the ones to delete a block of text) must record arbitrarily large amounts of state information.

This type of recording allows you to back up as far as you like. Since, in essence, each change to the state information is recorded, all earlier states are recoverable by reversing each state change (in reverse order, of course).

There is more to implementing undo than just recording state changes, but the additional items are more icing than cake.

First, most undo commands operate on a user-command, not sub-editor call, basis. Thus you must also record when a new user command is given. Thus, each undo then undoes consecutive changes until it reaches such a command marker. In this way, even a complex global replace can be undone. Note that in general you will wish to have repeated undo commands undo successively-earlier other commands.

Second, it probably makes sense to undo an entire consecutive set of newly typed characters as a single command.

Third, the resources available to retain the undo information may be limited. This design minimizes that problem, as the "excess" undo state can simply drop off the end. The part that is retained will still be consistent.

Fourth, the operating system may not support the undoing of file-level operations or other commands. In some cases, you can simulate such undoing (say, by making a backup file), but in general you will have to live with some limits to undo. (Undoing a print operation after the printing is complete is quite difficult.)

Fifth, state information is kept in places other than in the sub-editor. These other places must also be incorporated into the undo facility.

An Undo Heresy

Is undo a nice feature to offer? Yes. Is it vital to an editor? Probably not. Will adding it make a poorly designed editor into a good one? No. Will it make such an editor acceptable? Maybe. As was said earlier, the best way to help the user is with a good command set design: it will minimize the need or desire for an undo.

While undo is a general-purpose facility that has good applications, it is not clear that a text editor is one of them. The "good design" approach (using the Emacs command set as an example) and the undo approach will now be compared in their approach to moving around in text and deleting text.

Moving around in text is simply solving the problem "I am at X and I want to be at Y." The good design solution involves translating this difference into a sequence of commands to move the point from X to Y. If a mistake is made in the process of implementing the solution, the problem is merely restated to "I am at X' and I want to be at Y" and it is re-solved. The undo solution differs by detecting the error (*i.e.*, deviation from the intended solution), saying "undo" to put you back on the original path, and proceeding. Ordinarily this difference between the two solutions is not very great.

If the user has accidentally moved a large distance (say, to the start of the buffer), it becomes a little more difficult for the user to recover his or her earlier position. Emacs-type editors resolve this issue by having the large-movement commands set the mark to where you were. Thus, an interchange point and mark sequence will recover from the error. Keep in mind that almost all of the time, the user does not care where the mark happens to be.

The two approaches are all but identical in the text deletion case. On the one hand, accidentally-deleted text is recovered with a "yank" command, and on the other hand, with an "undo" command.

In conclusion, undo is a nice extra feature, but is no substitute for a good design.

Redo

Redo is the mechanism for undoing an undo. Conceptually, the record of undos is:

1. most-recent command changes
2. next-most-recent command changes
3. next-next-most-recent command changes
4. ...

The first invocation of the "undo" command undoes #1. The next invocation undoes #2, and so forth. Redo redoes the most recent undo, with repeated "redo" commands moving back up the undo stack. Let's look at an implementation:

```
FLAG Undo(FLAG *is_argptr, int *argptr, FLAG *is_exitptr, char c)
        {
        if (undo_ptr == NULL) undo_ptr = last_command_undo;
        Undo_Command(undo_ptr);
        undo_ptr = Previous(undo_ptr);
        return(TRUE);
        }
```

```
FLAG Redo(FLAG *is_argptr, int *argptr, FLAG *is_exitptr, char c)
        {
        if (undo_ptr == NULL)
                Error("No undo to redo");
        else {
                Redo_Command(undo_ptr);
                undo_ptr = Next(undo_ptr);
                }
        return(TRUE);
        }
```

and, in the main command loop:

```
if (last_cmd != Undo && last_cmd != Redo) undo_ptr = NULL;
```

The **Undo** procedure checks and, if it hasn't been called "recently," starts at the latest command. It undoes the command, then sets a pointer to point to the undo information for the previous command.

The **Redo** procedure checks and, if **Undo** hasn't been called "recently," can't run, as there is no undo to redo. Otherwise, it redoes the command and sets a pointer to point to the undo information for the next command.

The main command loop determines whether the **Undo** or **Redo** procedures have been called "recently." Basically, if the last command wasn't undo or redo, it resets the undo pointer to null.

This implementation allows arbitrary undoing and redoing in any combination, so long as the commands are given sequentially. Bear in mind that with the "change in state" recording of undo information, it is only legal to apply those changes in the correct order.

This implementation ignored the arguments to the undo and redo commands. Feel free to assign reasonable interpretations to the arguments in your implementations.

Macros

In a sense, macros allow a user to give commands by specifying them implicitly instead of explicitly. Macros fall into three general levels.

Again

The "again" facility allows a user to say "do what I just did again." For it to be most useful (*i.e.*, easier to type than retyping the command), it should be assigned to a short key sequence (*i.e.*, one shifted key).

The "repeat count" argument to the "again" command should be used instead of the previous "repeat count" argument to the command being repeated.

Keystroke Recording

This facility allows the user to say "start recording," then give a series of commands (observing their effects as they are typed), then say "stop recording." Later, the entire sequence of keystrokes can be replayed with a "play recording" command. A "repeat count" argument to the "play recording" command will cause the recording to be replayed the specified number of times. Note that commands within the recording can have repeat count arguments of their own.

Macro Languages

Finally, the editor should provide the user full access to the editor's macro language (if any). This language will in general provide a full programming language, thus allowing the user to specify an arbitrary set of editing operations as well as a way of naming these procedures for later invocation and key rebinding.

Redisplay Interaction

The introduction of keystroke recording and macro languages only servers to underscore the separation between the redisplay code and the rest of the editor described in the previous chapter. The playback of recorded keystrokes will almost certainly complete with no intervening redisplay. Thus, if any of the code based its actions on the current window contents, it would almost certainly execute incorrectly.

Questions to Probe Your Understanding

Expand the main command loop to include other types of input such as mouse operations. (Easy)

Generalize the main command loop and related code to use a general event-driven mechanism. (Medium)

Write routines to move by a word and to delete a sentence, but be sure to consider all of the punctuation and white space aspects of the problem. (Medium)

What is a good memory management scheme to use for holding the undo information? (Medium)

What fundamental support is required in the editor to best implement the keystroke recorder? (Easy)

Chapter 9: Command Set Design

> "And hast thou slain the Jabberwock?
>   Come to my arms, my beamish boy!

Previous chapters have discussed the external constraints on implementations and the division of the editor into more manageable pieces. You now understand how to build an editor. But which editor should you build? This chapter discusses various issues involved with designing the editor's command set.

Your editor implementation will (you hope) be used by many people. Regardless of the expertise of these people, there are some design principles that should be followed. A good design incorporates these qualities:

- responsiveness
- consistency
- permissiveness
- progress
- simplicity
- uniformity
- extensibility

Each of these qualities will be discussed in turn.

Responsiveness

Responsiveness means that each action taken by the user is handled and confirmed immediately. In other words, the application responds well to the user. This good response allows skilled users to work fast.

A different name for responsiveness is *visible effect*, which is to say that every action taken by the user has a visible effect on the display. This effect might be simple cursor motion, a change in the text being shown, or a message. Even if only an internal state variable is changed, that state variable should have an indicator on the display. (Ringing the display's bell is considered to be a "visible" effect for the purposes of this section.) By following this principle, the user is never in doubt whether the application is keeping up with his or her typing or has received a command: there is consequently never a need to issue a command because of doubt or uncertainty.

On some displays, the desire for a visible effect can conflict with other design goals such as minimizing display flicker. One solution to this problem is to delay the display of visible effect until the user stops typing for a few seconds. In this way, as long as a user is typing, he or she is presumed to know what he or she is doing and so the feedback is less important. However, if the user should stop typing, the application would quickly show the current state.

An important ramification of responsiveness is good error-checking. User input should be checked as soon as logically possible after it has been entered. If the input passes the check, some sort of confirmation is given (*e.g.*, a message or beep). If the input fails the check, an error indicator is displayed. As a general rule, no incorrect input should be accepted, unless it is infeasible to perform the check. Different checks will naturally be performed at different times. For example, if the user is being asked to enter a number in a specified range, the individual characters can be checked as they are typed to ensure that they are digits. The number as a whole cannot be range-checked until the user has indicated that it is complete (say by pressing the Return key).

Consistency

Consistency means that all parts of the application work the same way. This topic is covered in more detail in the section on "modes."

Permissiveness

Permissiveness means that the user is in control of the application and not vice versa. While this might sound tautological at first, it is a principle that is often honored only in the breach. Think of the applications that you have used that lead you through a step-by-step process, with only limited choices at each step. Often, these applications do not allow you to review or change earlier decisions short of aborting the whole application and starting over from scratch.

Writing a permissive application is both easier and harder than writing a non-permissive version of the same one. It is harder because the implementation has to be able to handle any request at any time. (An "event - message - object" design model makes it easy to handle such unpredictable requests.) If some process is multi-step, the application must have interlocks to prohibit processing the steps out of order (and of course these interlocks should have their state variables displayed). It is easier because you as a designer do not have to anticipate all possible paths through the application and decide in advance which ones are reasonable.

Progress

Progress means that each command should meet some part of the user's goals. An example of a command that does not make progess would be a command to "show the current line." This command does not contribute to making any of the changes that the user (presumably) desires: it just wastes the user's time and effort. It is the elimination of such commands that makes screen-oriented text editors such an improvement over the older ones.

In general, there should be no commands for the user to give that merely tell the application to do something that it has enough information to figure out on its own. For example, if the user moves to the end of the buffer, the application should display that part of the buffer. (Actually, as stated this principle is a bit harsh. From time to time, the application will not be able to guess correctly and it is acceptable to have a way for the user to take control. For example, sometimes the user wants to position the window to include two areas of particular interest. The application in general cannot detect this case. But if that happens often, there is a design problem.)

Simplicity

Simplicity goes under the sobriquet of "keep simple things simple." Complicated things can be complicated (or simple if that works out), but simple things should never be complicated.

This principle means that the basic editing operations (insert, delete, move) should be as conceptually simple as possible. For example, inserting a character is a conceptually simple operation. The simplest way of expressing that operation is to just type that character. Having input/edit (or "input/overtype" or "insert/replace") modes is an example of making a simple thing complicated. With the input/edit mode, inserting a character becomes "am I in insert mode? No. Then type the 'go into insert mode' command, type the character, and maybe type the 'leave insert mode' command." Hardly simple.

This principle is closely related to efficiency. It is natural to think that the command set that requires the fewest user operations is the best one to use. Unfortunately, that natural thought does not remain valid when taken to extremes. On an extreme basis, the set of editing operations could be Huffman encoded into a command set. While the resulting command set would be optimally efficient, it would probably not be usable. For example, the command to insert the string "the " might be ^X 7. On the other hand, simple things tend to be efficient if for no other reason than that they don't have the baggage of being complicated. Ideally, the most-often used commands should be the shortest.

Uniformity

Uniformity also goes under the names "regularity," "predictability," and "orthogonality." Basically, a command set is uniform if, when a user knows some part of it, he or she can predict the unknown parts. Another way of looking at it is that the command set fits into a pattern.

This principle is important in that the user is freed from learning each command separately. Instead, the user learns some of the commands and a set of rules for generating the rest. For example, here are the basic Emacs character commands:

^F move forward character

^B move backward character

^D delete the following character

^H delete the preceding character

(Keep in mind that ^H is also the Back Space key.) Here are the word commands:

^[**F** move forward word

^[**B** move backward word

^[**D** delete the following word

^[**^H** delete the preceding word

A user learns these commands by learning this basic command set:

F move forward ...

B move backward ...

D delete the following ...

^H delete the preceding ...

and these rules:

- control-shifted means "character"
- ^[-prefixed means "word"

It is rare that a command set will ever be completely uniform. However, it is important to take advantage of uniformity where possible.

Extensibility

Extensibility means the ability to accommodate changes. This principle has a number of aspects. First, changes can be accommodated by designing in "holes" or "gaps" where users can install commands of their own. Second, the command set can be made rich ("large") as the larger number of commands provides more places for commands to be placed. Third, a uniform command set helps extensibility. For example, if the command set has a set of "sentence" operations (move by, delete by), these can be converted as a set into "statement" operations for use in programming languages.

Modes

This chapter uses the word "mode" in a different way from the command-set-oriented use of "mode" from earlier chapters. It is unfortunate that the same word is used by the industry in different ways.

What are "modes," and why should you care about them? Simply put, a design has a mode when an end user has to do an "unnecessary" action in order to do the desired action. You should care about modes because having modes can make a program harder to use. As a programmer, you are used to modes and deal with them constantly without conscious thought. Your end users, however, may be confused by the presence of modes. Their thinking might go something like "I pressed the 'f' key (at command level) and it showed me an 'f', so why does pressing the 'f' key here move me forward by a screen?"

The definition just presented is a little abstract, so an example is in order. A piano is a device that has (almost) no modes. If you want a piano to make a sound, you just press a key. Each key is independent: it produces the same note regardless of which other keys were pressed before.

A piano really does have modes. You select the modes with the foot pedals. It is easy to learn to overlap the key presses with the foot pedal ups and downs, so the modes, although present, do not interfere with playing the piano. Moreover, the modes have a great deal of overlap with the basic keys. In a manner of speaking, the piano's "command set" is quite orthogonal.

A typical scientific calculator has many modes. An example of such a mode is the degrees/radians selection. If you are in degrees mode and want to compute the sine of an angle in radians, you must first switch to radians mode and then compute the sine. Switching to radians is not part of your calculation. Rather, it is something that you have to do in order to perform your calculation. Hence, the calculator has a mode. A similar – but less useful – mode in a text editor would be an insert/overwrite (or replace) mode.

These examples are at the extremes of the range. The piano is an almost modeless device, while a calculator has many modes.

What does all this have to do with designing an editor's command set? These things:

- you should have as few modes as possible;
- modes should be aligned with the activity;
- if you must have a mode, make it visible.

Adding keys (or a mouse) can reduce the number of modes. For example, having both "sine in degrees" and "sine in radians" keys would eliminate the degrees/radians mode. However, there is an upper limit to the number of keys that you can put on a user input device, be it a calculator or a keyboard.

Rearranging the modes so that they coincide with natural breaks in the activity also reduces confusion. For example, it is not too unreasonable to have a "text editor mode" and a "spread sheet" mode, where the two modes correspond to completely different applications. Switching modes is less confusing because the end user is mentally changing gears at the same time. On the other hand, with modern computers' ability to rapidly switch between different application programs, it can be very valuable to have the different applications present the same interface to the end user: doing anything else is often not in the end-users' best interests. Of course, if you did provide additional functions within the text editor, the differences in the modes whould be minimized.

Returning to the example of an insert/edit mode for an editor, we can see how that mode is not aligned with a change in activity. To the user, "editing" is a single activity that comprises both deletion and insertion. Changing between insert and edit modes is thus a mode change with no accompanying activity change.

Some text editors offer an insert/replace mode which affects how newly typed text affects the existing text. In insert mode, newly typed text is inserted. In replace mode, each newly typed character usually replaces an existing character. However, in many cases users do not want to replace characters: they want to replace words, sentences, or other higher-level objects. In these cases, simple replacement is not sufficient since it is unlikely that the new text is exactly the same length as the old. The correct effect can be readily achieved when insert mode is combined with an operation that defines a region or selection that identifies the old text.

Modes should be made visible. For example, in older calculators, the degrees/radians mode was hidden. New calculators have an indicator on the display that shows the current mode. Although the mode is still there, the indicator reminds the operator that the mode exists and can even guide the operator's next input.

The best way to make the modes visible is to show all state information on the display. In this way, it is *possible* for a knowledgeable end user to predict accurately the effect of the next command by examining the current display. Of course, not all end users are "knowledgeable," but how could a *non*-knowledgeable user ever succeed if a knowledgeable one cannot?

Note how the reasons for making modes visible dovetail with the reasons mentioned earlier for commands having visible effects.

Use of Language

DO NOT PRODUCE OUTPUT IN ALL UPPER CASE. UPPER CASE IS MORE DIFFICULT TO READ THAN lower case. Proper capitalization and punctuation are also important. Would you rather read:

```
ENTER CITY STATE AND ZIP CODE:
```

or:

```
Enter city, state, and zip code:
```

Use full words and phrases: do not abbreviate. Displays are large enough and output fast enough so that abbreviations are no longer required.

Prompts should have the form *verb object* (at least in English). A prompt of:

```
Username:
```

doesn't tell the user what to do. However, a prompt of:

```
Enter your username:
```

or even:

```
Please enter your username:
```

is reasonably unambiguous. In the first case, a user is apt to feel confused and unsure of what to do next. (What about a username? Should he or she go get one? Whose username?) In the second case, that confusion vanishes.

Error messages should state *whether* the operation was performed, *why* something went wrong, and *what* to do instead. Instead of:

```
File write error.
```

or (gasp!):

```
SIO-FI-ERR-12
```

this:

```
The file was not written because the disk was full.  Clear space
on the existing disk and try again or write the file to a
different disk.
```

Longer, but much better.

The application should do what computers do best: arithmetic, checking, recording. Users should do what they do best: direct the application to solve a problem. Don't make the user count things or keep track of what was done in the past.

Be generous in what you accept. If both Delete and Back Space are used for erasing a character, accept both. Unless there is a good reason otherwise, don't distinguish between upper and lower-case input. In general, if there is a possible way to unambiguously determine what the user wants, accept it.

Guideline Summary

This section presents a brief summary of the guidelines already presented.

Overall

- Responsiveness: reflect all input immediately to the display.
- Consistency: the same input should always have the same result.
- Permissiveness: the user controls the application.
- Progress: the user input should achieve the user's goal, not be for the application's benefit.
- Simplicity: keep simple things simple.
- Uniformity: make the commands easy to learn and to predict.
- Extensibility: plan for growth and change.

Modes

- Avoid modes where possible.
- Where modes cannot be avoided, align mode changes with activity changes.
- Remind the user what mode(s) are in effect.

Use of Language

- Use mixed case, proper capitalization, and punctuation.
- Use full words and phrases: do not abbreviate.
- Prompts should tell the user what to do.
- Use full error messages.
- Have the program do what computers do best.
- Be generous in what you accept.

Structure Editors

One idea that keeps recurring is that of a "structure editor." In general, a structure editor limits editing to valid transformations on the object being edited. They are often used as programming language editors. In those cases, there may be a command to "insert an 'if' statement." The user then sees something like this:

```
if <#> then <>
else <>
```

(This example does not use the C language. In general, people don't do structure editors for the C language.) The point is positioned at the "#" character and the user is then allowed to make syntactically valid transformations to continue programming. These editors are often found as the subject of research papers. For reasons that will be described, it is fortunate that they are not often found anywhere else.

Of course, the user is in trouble if, for example, he or she decides to negate the condition and make the "else" clause into the "then" clause. If the user is lucky, there will be an editor command to do this operation. If not, the user may have to "cut" the else part and "paste" it into the then. Or worse, the user may be forced to delete the else part and retype it at the then.

It may well be possible to create a structure editor that is also a good editor design. However, in all my research I have never seen one. There are two reasons why creating such an editor is difficult:

First, while the structure-oriented operations may be well suited to the process of *writing* a program, they are not well suited to the process of *editing* one. The distinction is a subtle but important one. The examples (usually shown in the papers) all show how easy it is to write programs this way. After all, it is a nice typing aid to be able to insert many characters of language statement with a short command. However, most of the work involved in programming is in editing programs that are already written. Editing operations are often ugly and involve intermediate states that are not valid language syntax. It is just in those areas that the structure-oriented operations start getting in the way.

Second, there is no carryover from one part of the editing task to another. Sure, it may be easier to write the program, *but the task of editing text strings and comments to the program has not been addressed by the programming-language editing commands.* The user still needs a full-feature editor to handle the strings constants, the comments, and other documentation that are an integral part of any programming project. By adding the structure editor, either a completely separate editor or a complicated new mode has been introduced and consistency has been lost. (I will completely skip over the question of how to handle the programmer that is editing more than one programming language. I will point out that I have worked on projects where I have been editing programs written in more than five languages at the same time.)

Note that the arguments just presented address the concept of a structure editor, not any one editor in particular.

Programing Assistance

Even if the structure editor approach is not the best, there are still techniques that can be used to help write and edit programs. If you like, it could be said that these techniques are adapted from structure editors. However, the origins of these techniques are lost in the mists of history and no one knows which was developed first: structure editors or the adaptations to general editors.

Typing aids: The first technique is to have commands that serve as typing aids. These aids would insert statements or statement parts by typing just a few characters (presumably fewer than the statements or parts themselves!). In this way, users gain the "express typing" benefits of structure editors.

Language modes: Further, a language mode can tailor the effects of commands to suit the characteristics of the language. For example, the commands that move by or manipulate words would be adjusted to use language tokens; those that use sentences would be adjusted to use language statements; and those that use paragraphs would be adjusted to use a statement block or procedure. In addition, commands that perform indentation can also be modified to handle statement nesting.

However, these alterations change how commands work and so some predictability is lost. In addition, while the alterations would presumably only be made for buffers that hold programs, these programs include comments. Hence, the commands need to "know" whether they are operating in a comment and thus whether to use the altered behavior. Even so, not all users are happy with such changes. Hence, there should be a way for users to turn them off. (I, for example, prefer to disable all language modes when editing.)

Syntax checking: Structure editors offer good syntax checking. Most even prevent you from creating a syntactically incorrect program. While possible to implement, syntax checking is not clearly appropriate for an editor, given that a better alternative may be available. This better alternative is for users to be able to invoke the compiler from within the editor, and to have the editor be able to parse the compiler's error and warning messages.

Simple syntax checking is a feature that looks useful on the surface, but turns out not to be very useful in practice, as good programers tend to make relatively few syntax errors. Syntax checking can catch errors like:

```
ovid Foo()
        {
        return (a -+ b;
        }
```

This example has a misspelled keyword, a missing close parenthesis, and an illegal operator combination. Syntax checking can catch the last two of these, but not the first: at the syntax level, there is no way to tell whether "ovid" is a misspelled keyword or a programmer-defined type.

Semantic checking can catch the "ovid" problem, as well as missing declarations, mismatched types, and other such problems. With programs spread across multiple files, there is simply no way that an editor would be able to assemble the information required to perform the correct analysis. It is up to the language compiler to perform that function.

Compiler invocation: And so we bring up the best way for an editor to help in program development. That is for the user to quickly and easily be able to invoke the compiler and work with the results. The commands might be "compile this file," "move to the place indicated by the next error message," and so forth. The lesser features such as syntax checking would only be used on systems where invoking the compiler is expensive (in user time, not computer power) to do from within the editor.

Command Behavior

This section describes some of the considerations involved with designing some of the commands. As with other parts of the book, the purpose of this section is not to say merely "do it this way," but to review why different approaches should be considered.

Does Down Move the Point or the Text?

Let's say that the point is somewhere in the middle of a large buffer, large enough that neither the top or bottom is on the display. The user gives the "move down a line" command (say, by pressing the down-arrow key). What happens?

First, the point (and cursor) could both move down one line. The user is thinking "I want to move down" and lo, the point moves down.

A variant on this choice is to move the point down one line, but to move the text of the buffer *up* in the window. This variant has the unfortunate property that the cursor is always kept in the center of the window.

There is a another choice. The cursor could stay in the same place, and the *text of the buffer* move down. In effect, this moves the point *up* one line.

A moment's thought shows that both choices are indeed valid interpretations of "down." In fact, both have been implemented many times. All modern editors now use the first interpretation. In fact, you might be wondering why anyone would select the second interpretation.

The descriptions just given do not reveal why the two interpretations arose. For that information, we have to step from the world of text editing into the world of computer graphics.

Consider the following picture:

```
 o              |\              The quick
---             | \             red fox
 |              | /             jumps over
/ \             |/              the lazy

user            display buffer
```

In this view, the user sees the text on a display. Now let us redraw this picture more abstractly:

```
 o              ------  The quick
---             |ed f|  red fox
 |              |umps|  jumps over
/ \             ------  the lazy

user            window  buffer
```

In this view, the user is "looking through" a window onto the text. The user sees only that part of the text that can be seen through the window. Now the question "when the user gives the 'move down one line' command, does the command move the window or the text?" makes sense.

Scrolling vs. Paging

Closely related to the previous point is whether to scroll or page the screen. Again, there are two choices, and again, we are considering the case where the user is giving "move down one line" commands. In all cases, the *point* is moving down one line at a time. The question relates to how the *cursor* moves on the screen.

First, the cursor can move down one line at a time until it reaches the bottom of the screen (possibly with a line or so of "guard zone"). Once there, the whole screen moves up one "page" and the cursor is re-centered.

Second, the cursor could stay in the same place on the screen and the text could move up by one line.

The second method offers the advantage that the maximum amount of surrounding text is always visible. However, it offers the much more severe disadvantage that "just moving around" appears to be constantly changing the text. That is quite distracting to users. It also ties what the user sees with where the user is making changes. Thus, if the display has a 24-line screen and the preferred row is line 10, the user is out of luck if he or she wants to make changes while looking at text that is more than 10 lines before the point or 14 lines after it.

A third method would be to have the cursor move down to a guard zone, then scroll the screen instead of paging it. This method offers better continuity than does just paging. It is especially nice if you repeatedly give the "move down line" command. Personally, I find it exasperating because your context gradually reduces to the size of the guard zone, then *stays there*. With paging, the context is automatically restored to about one-half of the window. When using editors that select this method, I have to give the "recenter window" command much more often than I like to.

Page Breaks

This is more of an issue with word processors than text editors, but it is worth mentioning anyway. The problem arises when the program is displaying the buffer in its "printed form," including page breaks. It is very tempting for the implementor to always position the page break at the top of the display. However, it is important that users be able to see the text just before and after the page break at the same time.

How Many Ways Can You Move by a Word?

When writing commands that operate on words, the first question that arises is "what is a word?" We need a definition that is both possible and easy to implement. We will approach a definition in a series of refinements.

The first step is to consider all of the characters between white space to be one word. With this definition, the sequence:

```
This is a very-strange test sentence, isn't it?
```

would be considered to be eight words. While a good start, it is not sufficient, as the following sequence would be considered only two words:

```
here--a phrase
```

But this sequence would be considered four words:

```
here -- a phrase
```

Thus, the next step is to define a word as a string of letters and digits. With this definition, the three examples would be considered to have ten, three, and three words, respectively. This definition has the advantage that for something to be considered a word, there should be something "word-like" about it (*i.e.*, the characters "–" are not considered to be a word. In addition, the presence or absence of extra spaces around the "–" does not change the number of words: a good sign that we are on the right track.

Our refinement can stop right here and be considered acceptable. There are some changes that we can make, but these changes are not uniformly considered improvements.

The first change is to add some characters to the "word" characters in language modes. For example, when writing C programs, the underscore character ("_") is legal within a token. By adding this character to the letters and digits, a "move by word" command will now properly move by tokens.

The second change is to add other characters, such as dash ("-") and quote (" ' "). But these are added in a special way: they must be surrounded by letters or digits in order to be considered as part of a word. This change allows the "very-strange" and "isn't" parts of the sequence to be considered as single words.

However, suppose that you had a very-long-hyphenated-phrase. It probably makes sense to consider this phrase as four separate words. In particular, it is better to err on the side of dividing one "word" into two rather than combining two "words" into one. For example, in our very-long-hyphenated-phrase, it would be difficult to change the "hyphenated" to "dashed" if word motion commands considered the whole thing as one word.

Moving by Words

As with many of the other topics here, there are two popular ways of moving forward by words. Oddly, though, there is only one popular way of moving backwards.

One way to move forward is to move to the end of the word. For example, if we had the text:

```
one two three four
     ^
```

and the cursor was at the "w" (which means that the point is between the "t" and the "w"), and a "move forward word" command were given, this move would leave the point here:

```
one two three four
       ^
```

i.e., just before the white space after the word. The other method would move the point to the start of the next word:

```
one two three four
        ^
```

When moving backwards, both methods leave the point at the start of the word:

```
one two three four
    ^
```

The difference may become slightly more clear if we look at the code that might be used to implement these commands. Assuming that the constant *WORDSTRING* contains the characters that comprise a word, the code to move backward a word is this:

```
Find_First_In_Backward(WORDSTRING);
Find_First_Not_In_Backward(WORDSTRING);
```

The code to move forward to the end of the word is the same code, with "Backward" changed to "Forward":

```
Find_First_In_Forward(WORDSTRING);
Find_First_Not_In_Forward(WORDSTRING);
```

Finally, the code to move forward to the start of the next word is this:

```
Find_First_In_Forward(WORDSTRING);
Find_First_Not_In_Forward(WORDSTRING);
Find_First_In_Forward(WORDSTRING);
```

The first method, to leave the point at the end of the current word, has the property that it is symmetric with respect to backward motion. The second method, to leave the point at the start of the following word, lacks that symmetry. On the other hand, it makes it easier to move to the beginning of the following word.

The choice is up to you. I strongly prefer the first method.

Deleting by Words

This question is "okay, so I have decided what happens when I *move* by words. What should happen when I *delete* by words?"

The first answer, which works very well, is simply that you should delete whatever the corresponding move command would move over. Thus, if the motion command were to move to the start of the next word, the deletion command should delete that same text.

However, consider this case:

```
This is some text.
                   ^


--------------------


And, three lines later, more text.
```

The "move forward word" command would move to just before the "A". However, is it really desirable that the "delete forward word" command delete the lines in between, including the row of dashes? Well, yes, if you want to be consistent (and predictable). This is one of the reasons why this style of word motion is not the best one to use.

The second answer would be to be "intelligent." This view is used by Apple Computer (see Apple 1987). In it, you would change exactly what was deleted based on circumstances. For example, with the text:

```
Here is some text.
           ^
```

a "delete forward word" command would delete the word "some" and the following space, thus leaving this:

```
Here is text.
           ^
```

instead of this:

```
Here is  text.
           ^
```

This definition has the advantage that deleting a word deletes the "supporting structure" for the word as well and thus makes the text as if the word was never there. It also means that "sloppy" users don't leave stray extra spaces around. It has the disadvantage that if you wanted to replace one word with a new one – a very common operation – you now have to re-type the space. This definition also loses predictability in that only white space is so treated: commas, periods, and other punctuation marks are not. So, to replace "old" with new in the following text:

```
This is an old word.
                 ^
```

you give the "delete forward word" command then type "new ", but to do the same replacement with the text:

```
This is an old, word.
                 ^
```

you give the "delete forward word" command, then simply type "new". Now explain that to a user quickly and painlessly. (Note that Apple Human Interface Guidelines do not provide for word-level operations other than selection.)

In all fairness to Apple – and I believe that their guidelines are excellent – their guidelines are build around a keyboard/mouse combination for user input. This book assumes that only a keyboard is available. Changing such a basic assumption will result in changing some of its conclusions: that is why you should make your choices based on a full understanding of the options and assumptions that apply in your situation.

These examples have all operated on full words. The Apple guidelines do not have guidelines for how to handle deleting parts of words because the guidelines only support whole words as objects. However, you are free to invent your own semantics for handling partial words in an "intelligent" manner.

Where Do Sentences and Paragraphs End?

I will start with the cheery statement that there is no way to correctly determine the ends of all possible valid English sentences by analyzing syntax alone. Why not?

Consider these text sequences:

```
This is a sentence.
The value 3.14159 is close to the value of pi.
The value 3. is close to the value of pi.
Dr. Martin is a medical doctor.
I hate typing long words and prefer to abbrev. them.
```

We all know that a period is used to end a sentence. The second example shows that periods can occur within a sentence. The third shows that periods can end a token and yet not end a sentence. The fourth shows that the token can be a word. The last shows that you can't just work off a list of known abbreviations. When considering these examples, remember that we are applying semantic knowledge to the statements, something that is beyond the ability of most computer programs. From the program's point of view, the sequences might as well be:

```
Xxxx xx x xxxxxxxx.
Xxx xxxxx 0.00000 xx xxxxx xx xxx xxxxx xx xx.
Xxx xxxxx 0. xx xxxxx xx xxx xxxxx xx xx.
Xx. Xxxxxx xx x xxxxxxx xxxxxx.
X xxxx xxxxxx xxxx xxxxx xxx xxxxxx xx xxxxxx. xxxx.
```

All that said, what is a way out? The definition that I have found that works best was worked out by trial-and-error and refined over a period of years is this:

```
Find_First_In_Forward(".?!:");
Point_Move(1);
```

```
Find_First_Not_In_Forward("\"')]}");
if (xiswhite(Get_Char())) <you are at a sentence end>;
```

This definition looks for any of the sentence-ending characters (colons are considered to end sentences here). It then skips over any of the characters that tend to follow sentence ends. Finally, it checks for white space.

This definition has the unfortunate property that the last three examples are considered to be two sentences each. On the other hand, it has the advantage that it works fairly well on non-contrived examples. And, as with words, it is better to count one sentence as two than to treat two sentences as one.

Note that only one white-space character (any character of Space, Tab, newline, or whatever else you wish to include) is required to end a sentence. Depending upon which style manual you follow, either one or two spaces should be included after the end of a sentence. However, even if your style manual requires two spaces, your users may not use that manual or may simply forget to type the extra space. Hence, you should not penalize them for the omission.

I know of one implementation that makes life difficult for its users. Its end-of-sentence definition requires two spaces. Yet, when you refill a paragraph, it removes the second space. Thus, you can move by sentences until you first refill the paragraph, after which the entire paragraph is treated as one sentence...

Paragraph ends are much easier than sentence ends. If you are using word wrap, each paragraph will be one line long. Thus, a newline character will end a paragraph. Otherwise, a newline followed by any white-space character (Space, Tab, newline, etc.) will mark the end of the paragraph.

If you are fortunate (or unfortunate, it depends on your outlook) enough to use a text formatter, you will want to include your formatter-command characters as paragraph-break characters. For example, if you are using the nroff formatter, you will want a paragraph break here:

```
This is some text.
.LP
This is some more text.
```

So just look for a newline followed by either white space or a dot.

Moving and deleting by sentences and paragraphs involves all of the same problems as moving and deleting by words. See the earlier discussion.

How to Search

This topic supplies lots of choices, all of them good. The question is more one of how much work you want to put into your implementation than which is the correct approach.

The first choice is between buffered searching and incremental searching. *Buffered searching* means that your implementation prompts for a search string, waits for the user to enter the complete string, then performs the search. It works and is easy to implement, but not as good as incremental search.

Incremental search means that the implementation searches as the user types. Here is an example:

user types **what is done**

ˆS start incremental search

a find the first "a"

b (1) find the first "ab"

c find the first "abc"

ˆH erase the "c" from the string; go back to where you were after (1)

d find the first "abd"

ˆS (2) search again for "abd": you are now at the second one

ˆS search again, you are now at the third one

ˆH get rid of the last ˆS; go back to where you were after (2)

You get the idea. The commands available can be as powerful as you like. This is clearly a much nicer way to search than buffered searching. Just as clearly, it is more work to implement.

The next question is what the search string should be. The most simple case is that the editor should search for the string exactly as typed. Thus, this string:

```
Some text.
```

would match only the string "Some text." in the buffer. While simple, it is not necessarily useful.

The first alternate way to search would be to simply ignore upper and lower case. Thus, the string would also match "some text." and "SOME TEXT." and "SoMe TeXt."

Another way to search would be to have lower-case characters in the search string match either upper or lower case in the buffer, but an upper-case character in the search string match only upper-case characters in the buffer. The search string would then match "SOME TEXT." and "SoMe TeXt." but not "some text." This way is more useful than one might think, because you can enter "ROM" in order to find "ROM" but not the "rom" in "from", yet you can still find both "the" and "The" with one search string.

Another search variation is whole-word match. Thus, one could search for "the" without finding "then". In addition, it would be handy to be able to allow varying amounts of white space to match. Thus, our "Some text." string would match

```
Some
text.
```

and

```
Some      text.
```

You can get as complex as you like with all of this, up to wild cards and UNIX-style regular expressions. Just don't forget to include a way to quote any special characters so that the user can search for them exactly.

Commands to Handle Typos

There are two very common forms of typographic errors for which special commands can be helpful.

Capitalization Commands

One typographic error is the incorrect upper/lower case of a character, part of word, or word. Two different forms of commands can be defined to handle this task.

The first form operates as a "move forward word" command, but forcing all characters moved over to UPPER, lower, or Capitalized case (the latter is first character in upper case, and all others in lower case). Note that three separate commands are required.

The second form is a "rotate case" command. The definition that I use acts differently depending upon whether you are within a word or between words. In the former case, it affects all characters between the point and the end of the word. In the latter case, it affects the entire previous word. In either case, it examines the current state of the word and rotates it among word -> Word -> WORD -> word, etc. The point is not moved. This definition turns out to be very handy.

Twiddling

Another typographic error is interchanging two characters. For example, "teh" instead of "the". There are three forms of the command to fix this.

The first form interchanges the two preceding characters. Thus:

```
abcd
  ^
```

becomes

```
bacd
 ^
```

The second form interchanges the two surrounding characters. Thus our original becomes:

```
acbd
 ^
```

Neither of the first two forms moves the point. The third form moves the point and our original becomes

```
acbd
  ^
```

The advantage to this form is that repeated executions of the command serve to "drag" a character along.

Questions to Probe Your Understanding

Consider how your favorite editor's command set and implementation meet the design guidelines. (Medium)

Define a command that you would like to see in an editor. (Easy)

What are some different ways to handle moving down a line (hint: consider how to handle variable-width characters)? (Easy)

How do the word, sentence, and paragraph problems change when languages other than English are considered? (Medium)

Define a complete command set. (Hard)

Define a good structure-editor command set. (Hard)

Chapter 10: Emacs-Type Editors

Oh frabjous day! Callooh! Callay!"
He chortled in his joy.

With a thesis subtitle of "A Cookbook for an Emacs" and this book's subtitle of "Emacs for Modern Times," it is a safe bet that I would get around to discussing Emacs-type editors at some point. And so I have.

First, I will admit to being biased. I have used Emacs-type editors exclusively for going on fifteen years now (except for a brief stint in durance vile on a Macintosh). In addition, I have implemented or worked on implementations of a half-dozen Emacs-type editors. This chapter will describe what it is about an Emacs-type editor that makes it special.

"What Do You Mean, 'Emacs-type?' "

This chapter discusses "Emacs-type" editors and not a particular "Emacs" editor because, due to its very nature, there isn't one definitive "Emacs" editor. The closest that you can come to such a thing is either the original Emacs-on-TECO or Gnu-Emacs.

The Command Set

The specifics of the Emacs command set will always be shifting and changing. Even details on how the basic commands operate will change. However, the broad outlines of the basic commands are pretty constant, which is to say that an editor that does not implement them is not considered to be an Emacs-type editor. Appendix C lists one Emacs command set and a set of changes to it.

By and large, the Emacs command set follows the design guidelines presented in the previous chapter. Those places where it falls short are usually either physical constraints (there are only a limited number of keys available), design compromises (to achieve A one must trade off B), or the ever-present "historical reasons" (it was a good idea at the time and, while we know better now, it is too ingrained to change). All things considered, the Emacs command set meets the design guidelines better than any other editor that I am aware of.

Some editors never get beyond providing the command set. These editors are neither extensible nor provide an extended environment. These limitations may be due to the implementation environment or merely the amount of effort that is available to devote to the project. There is a place for such editors (they are listed in Appendix B as "command set"). After all, an implementor should be able to decide to create a small editor without feeling obligated to spend years on the project. Given that this decision has been made, it is better to use the Emacs command set than any other (at least in my opinion).

The Extended Environment

Emacs-type editors have been and continue to be used for many things besides editing text. Here are some examples.

Emulation: If you or another user likes a different editor's command set for some reason, you can just emulate it in an Emacs-type editor. Thus, you can still have all of the power of an Emacs and your favorite command set as well.

Electronic mail: A text editor can be the primary interface to a mail system. Messages can be composed by editing a buffer and can be sent with a command. Mail can be read and managed by reading it into a buffer and having commands to perform such operations as "move to the next message" and "summarize all messages." Having the full power of an editor available makes it easy to un-delete an accidentally deleted message or to copy part of the text of a message into one's reply. In addition, you have only minimal additional learning in order to use the system very effectively.

Command shell: A text editor can be the primary interface to the operating system. Command lines can be edited with the full power of the editor before being evaluated. The past record of interaction can be kept and parts of it examined or re-used in new command lines. If the operating system does not have support for advanced terminals, a display editor can offer its interface for use by other programs. Other programs would then take advantage of the terminal independence of the editor. Alternatively, other programs would insert their output into a buffer and the editor would become an entire terminal-management system. This function has been somewhat superseded by a window system. But again, why learn two systems when you only need to learn one?

Compilation: A text editor can work closely with a compiler to speed turnaround when developing software.

Debugging: A text editor can be used by a debugger. Multiple buffers and multiple windows can be used to examine (perhaps multiple) source files, interact with the debugger, and see the output/input of the program as it runs. In addition, a debugger might use a window or two to do such things as constantly show the values of selected variables.

File interface: A text editor can be an interface to a complicated file. For example, an indexed sequential file can be updated by providing editor commands to read and write entries (adding or deleting them can be managed as well). The full power of the editor is available for editing the contents of the entry.

File system interface: A text editor can provide a smooth interface to the file system. A directory can be read by the editor and "edited" by the user. Files can be deleted or otherwise changed in a smooth manner by merely moving to the file name and giving a command (*e.g.*, "delete").

Binary files: A text editor can be used to examine and – when absolutely necessary – modify binary files. It can thus replace various patching programs.

Again, all of these functions are currently performed by one or more Emacs-type editors. The main advantage to building them off of the editor is that, even in the absence of such features, most users spend the bulk of their time using the text editor (or word processor, etc.). By extending that environment, only minimal learning is required to use those features. Users are thus free to get work done instead of having to spend their time reading manuals. This extended environment is one of the hallmarks of an Emacs-type editor.

Extensibility

The final hallmark of an Emacs-type editor is extensibility. Emacs was born because of the extensibility of its predecessor, TECO. TECO was extensible and many of its users took advantage of that extensibility to write their own command or change the existing ones. Eventually, one person (Richard Stallman) took a large number of those extension packages and created a "standard" package of Editor MACroS: EMACS. With extensibility in its heritage and even in its name, an Emacs-type editor is expected to be extensible.

Extensibility means, quite simply, that end users can change any of the features of the editor. There should be *no* feature that a sufficiently dedicated end user cannot change. This implies that the full source code of the implementation is available to all end users (and that is why GNU-Emacs is distributed under its "CopyLeft" policy).

Now, not every end user will want to recode redisplay. However, the principle remains. Most end users will only want to tweak a few parameters and maybe "fix" a command or two. That's great and you should encourage such changes by making it easy to make them.

Now that the importance of extensibility has been explained, it is easy to see why the Emacs command set cannot be standardized: each user will want to change it. In a way, the Emacs command set can be compared to a ball of mud. You can add more to it, or take some away, and you will still have a ball of mud. (Actually, this property is true of most extensible systems.) That is also why any comparison between an implementation of an Emacs-type editor and any other editor is pointless: the Emacs-type editor can be changed (and may already have been) to overcome any shortcoming.

Questions to Probe Your Understanding

Identify those ways in which the Emacs command set does not meet the design guidelines listed in the previous chapter. (Easy to Medium)

Identify other editors or similar programs that offer extended environments. (Easy)

Identify other editors or similar programs that are extensible. (Easy)

What other programs would you like to be extensible that aren't? (Easy)

Epilogue

'Twas brillig, and the slithy toves
  Did gyre and gimble in the wabe:
All mimsy were the borogoves,
  And the mome raths outgrabe.

And so concludes the verse "Jabberwocky" by Lewis Carroll. A piece of nonsense verse embedded in a nonsense work, it resembles the "real world" about as much as a computer program does. Which is perhaps why Lewis Carroll is so popular among computer programmers.

And, before you ask, the chapter structure of this book was worked out well before the fit of the verse was noticed. Right.

Questions to Probe Your Understanding

What parts of "Jabberwocky" fit particularly well with the chapters that they lead off? Particularly poorly? (Easy – or is it?)

Appendix A: A Five-Minute Introduction to C

This Appendix provides a brief introduction to that part of the C programming language as used in the examples presented in this book. This appendix is not a reference manual, nor does it describe all of C. Rather, it assumes that you are familiar with other programming languages in general and just need a nudge or two to follow the examples presented in this book. If you want to learn more about the C language, consult ANSI (1990) or Kernighan (1978). If you are interested in C, you should also look into the C++ language.

The best way to think of the C programming language is to consider it as providing the programmer with a mechanism for allocating and naming memory, control structure, and an extension mechanism. Many people consider it barely a high-level language. Perhaps for that very reason, it is ideally suited for systems and utilities such as text editors.

The *declaration mechanism* provides the programmer a way to allocate and name memory. Data types are oriented around what is best suited to the hardware.

The *language statements* provide a control structure mechanism. All of the usual control structures are available. In addition, a full suite of arithmetic and bit operators is available, again focusing around what is best suited to the hardware.

The *procedure definition and call mechanism* provide the extension mechanism. Many "standard" C operations such as string copy and input/output are implemented in terms of this mechanism.

Comments are enclosed in "/* ... */".

Case Conventions

The names in this book follow the convention that *UPPERCASE* names are pre-defined constants, *MixedCase* names are procedures, and *lowercase* names are variables. Again, these are conventions, not requirements.

Data Types and Declarations

All variables and procedures used are declared. Declarations are of the following form:

type variable;

The language supports the following data types:

char The variable holds one character. Characters are typically 8 bits wide. They may either be signed (range -128 to +127) or unsigned (range 0 to 255) at the implementation's discretion.

int The variable holds an integer of a size convenient for the hardware. This size is typically 16 or 32 bits.

float The variable holds a floating point number.

FILE * The variable holds a file descriptor.

struct *name* **{ <list of declarations> };** The declarations are combined into a single, larger data type named *name*.

type **(****name***)();** The *name* is the address of a procedure that returns a value of type *type*.

void No value. Used to indicate that a procedure does not return anything or accepts no arguments.

A declaration of the form

type *

means that the variable holds the address of an object of the specified type. The variable is called a *pointer* to the specified type. A declaration of the form

type name[*constant*]

means that the variable holds an array of objects of the specified data type. The array is *constant* object long. The form

type name[*constant1*][*constant2*]

is used for a two-dimensional array.

The following data types are not part of the language, but represent types used in examples. An implementation would define these in terms of existing-language data types.

FLAG The variable holds a True or False value. (In C a 0 value is considered to be False and any non-zero value is considered to be True.)

status The variable holds a success or failure status value. This value may include warning or error information.

location The variable holds a value that represents a point or mark location within a buffer.

time The variable holds a value that represents the time date and time.

private You get to define this.

Constants

Integers are written as themselves (*e.g.*, "56" means the value fifty-six).

Hexadecimal constants are written in the form "0x##", where the ##s are hexadecimal digits.

Character strings are enclosed in double quotes (" "). A NUL-terminator (a byte of decimal value 0) is automatically appended to the string by the compiler. Character strings are considered to have the type "array of char."

Character constants are enclosed in single quotes ("). They are automatically converted to integers whose value is that of the character. For example:

"a" is an array of two characters, consisting of the characters 'a' and NUL (values 97 and 0 decimal, assuming ASCII).

'a' is an integer whose value is 97, assuming ASCII.

while:

"abc" is an array of four characters, consisting of the characters 'a', 'b', 'c', and NUL (values 97, 98, 99, and 0 decimal, assuming ASCII). The form 'abc' is officialy undefined (some compilers might consider this as an integer whose value is 97 * 65536 + 98 * 256 + 99, but don't count on it).

The character '\b' refers to the ASCII Back Space character (8 decimal).

Note: the "" / " syntax is used in all code excerpts. However, the normal English typographical conventions of using "" are followed in the body of the text.

Pre-defined Constants

NEWLINE The character string that represents a system-specific newline, written "\n".

NL The character that represents a newline, 10 decimal.

SP The space character, 32 decimal.

TAB The horizontal tab character, 9 decimal.

NUL The nul character, 0 decimal. Character strings are terminated by this character.

NULL The null pointer: no data object can be at this address.

BUFFERNAMEMAX The size of the longest possible buffer name plus 1 for the trailing NUL. Possibly 33.

FILENAMEMAX The size of the longest possible file name plus 1 for the trailing NUL. Typically 1,025.

Procedure Structure

Procedures have the following structure:

```
type Name(<arguments>)
                {
                <local variables>

                <statements>
                }
```

The procedure is named *Name* and returns data of type *type* (*type* can be a structure or pointer as well as a basic type). The argument list contains a list of declarations or the keyword **void** if the procedure takes no arguments. The local variables are then declared (and may be initialized at each procedure invocation). Last are the procedure statements.

Statements

The statements are the usual ones. A semi-colon (";") terminates a statement. Comments start with "/*" and end with "*/". Statements can be grouped with "{" and "}" characters, so the sequence

```
{
<statement 1>
<statement 1>
        ...
<statement n>
}
```

is equivalent to one statement. White space and columns are not significant.

if (*condition*) *then-statement*

if (*condition*) *then-statement*
else *else-statement*

for (*initializer*; *end-test*; *increment*)

statements

`execute the initializer, then the end test, then repeat the statements, the increment, and end test until the end test becomes True`

break; `exit a loop immediately`

continue; `skip the rest of the loop body, but don't exit the loop`

while (*end-test*) *statements*

`repeat the end test and statements until the end test becomes True`

for (;;) *statements*

`repeat the statements forever: a break, continue, or return statement is used to exit the loop`

switch (*expression*) **{**

case `LABEL1:`
statements
break;
case `LABEL2:`
statements
break;
...
default:
statements
break;
}

`execute the statements after the label whose value matches the expression, or the statements after "default" if present and no label matches`

return(_expression_**);**

`return a value from a procedure`

Operators

The (possibly unusual) language operators are these:

= assignment; not test for equality

== test for equality; not assignment

!= test for not equal

! logical negation: !FALSE becomes TRUE

a+b returns the sum of a and b

a-b returns the difference of a and b

a*b returns the product of a and b

a/b returns a divided by b

a%b returns a modulo b

a&b returns the bitwise and of a and b

a|b returns the bitwise or of a and b

a^b returns the bitwise exclusive or of a and b

The construct "a @= b" where "@" is any of the operators "+" through "^" does the same as "a = a @ b", except that "a" is only evaluated once.

Operators that return True / False results return 1 for True and 0 for False. However, when a True / False value is required (say, in an if-condition), any non-zero value means True and zero means False.

&v returns the address of v

s.m selects member m of the structure s (s is of type "struct")

p->m selects member m of the structure pointed to by p (p is of type "struct *")

++v increment the value in v and return the new value

v++ increment the value in v and return the pre-increment value

--v decrement the value in v and return the new value

v-- decrement the value in v and return the pre-decrement value

The construct (*type*)*expression* (called a "cast") converts the value returned by the expression to the specified type.

Standard Library Functions Used in This Book

fclose(<fileptr>) closes a file opened earlier with fopen.

fgets(<buffer>, <length>, <fileptr>) reads one line from a file opened earlier with fopen.

fopen(<name>, <mode>) opens a file for reading. A <mode> of "r" means "read-only."

free(<ptr>) frees memory previously allocated by malloc.

isprint(<key>) returns True if <key> is a printing character or False if not.

malloc(<size>) allocates a block of memory at least <size> characters long.

memmove(<to>, <from>, <length>) moves <length> characters from <from> to <to>, working properly if the areas overlap.

memset(<start>, <char>, <length>) sets <length> characters starting from <start> to the character <char>.

printf(<format>) or *printf(<format>, <arg>)* copies the characters from the format string to the screen "as is" until a '%' character is encountered. The sequence "%s" means to take the next argument and send it as a string to the screen. The sequence "%c" means to take the next argument and send it as a single character to the screen. (The routine does a lot more, but the examples in this book don't use the extra functionality.)

strcpy(<to>, <from>) copy the from string to the to string.

strlen(<sting>) returns the number of characters in the string, not counting the terminating NUL.

Non-Standard Library Functions Used in This Book

Fatal(<message>) handles a fatal error.

xiswhite(<c>) returns True if the supplies character is a white-space character (space or tab) or False if not.

xstrcpy(<to>, <from>) works like the C *strcpy* routine to copy one string to another, but is defined to work properly if the strings overlap.

Appendix B: Emacs Implementations

This appendix has not been converted. To see the current Emacs Implementations list, click here. Its URL for citation purposes is:

http://www.finseth.com/~fin/emacs.html

Appendix C: The Emacs Command Set

First, there is no such thing as an "official" Emacs command set. One of the main reasons why Emacs-type text editors exist is that each user is free to change the commands to suit his or her own tastes. Another reason is that there are many different implementations of Emacs editors and each will have a slightly different command set. You should consult the documentation that comes with your implementation (or the documentation that you write for your own implementation) for specifics.

That said, this appendix will list most of the default command set that comes with GNU-Emacs. It will also list my own alterations to these defaults.

Emacs-type text editors implement the "one-dimensional array of bytes" editing model. Line breaks are represented as single Newline characters, regardless of the representation used by the operating system.

Notation

The first Emacs took advantage of a particular keyboard's features. This keyboard allowed you to type all of the normal characters. In addition, both **Control** and **Meta** keys were available. These keys acted as full Shift keys, allowing the user to specify Control- and/or Meta-shifts to any key. For example, one could type:

notation key combinations

5 5

% Shift-5

C-5 Control-5

C-% Control-Shift-5

M-5 Meta-5

M-% Meta-Shift-5

M-C-5 Meta-Control-5

M-C-% Meta-Control-Shift-5

The last two could also be written C-M-5 and C-M-%. Most current keyboards can only send the usual ASCII characters. That limitation removes the possibility of typing the multiply shifted characters and hence the need for the unusual notation. The command sets presented here will thus use the familiar caret notation, a copy of which is listed in Appendix E. There are still some systems that use extended keyboards like these. The documentation for those systems will use the original notation.

The Escape key (usually labelled "Esc") which sends the escape character is used as a prefix to make up for the missing Meta shift key. By convention, those keyboards that have an extra shift key (perhaps labelled Meta or Alt) can specify meta commands by setting bit 2ˆ7 in characters sent from the keyboard.

Emacs commands are, by convention, always upper/lower case-independent. Thus, ˆX B and ˆX b both refer to the same command. The only exception is the "self-insert" command, where B inserts a "B" and b inserts a "b". Only the uppercase versions will be listed.

The terms "S-exp" and "defun," while familiar to Lisp programmers, are probably not familiar to others. "S-exp" refers to a parenthesized list and "defun" refers to a function definition. Most Emacs language modes remap these Lisp-specific commands to comparable commands for other programming languages.

Default GNU-Emacs Command List

ˆ@ Place the mark at the point.

ˆA Move to the beginning of the current line.

ˆB Move backward one character.

ˆC Prefix for mode-specific commands.

ˆD Delete the following character.

ˆE Move to the end of the current line.

ˆF Move forward one character.

ˆG Abort execution and return to the edit loop.

ˆH Help

ˆI Tab to indentation in a language-specific manner.

^J Insert a line break and indent in a language-specific manner.

^K Delete the text to the end of the current line; if at the end of the line, delete the line break. With an argument, delete that many complete lines.

^L Rebuild the display from scratch, centering the point.

^M Insert a line break, leaving the point after the break.

^N Move down one line, staying in as nearly the same column as possible. If at the end of the buffer, grow the buffer.

^O Insert a line break, leaving the point before the break.

^P Move up one line, staying in as nearly the same column as possible.

^Q Quote: Insert the following character as typed.

^R Reverse search (see ^S).

^S Search incrementally for a string after the point:

- **^R** Search for the previous occurrence.
- **^S** Search for the next (2nd, 3rd...) occurrence.
- **^?** "Untypes" the last character (including ^R, ^S, etc.)
- **^G** Abort search: return to starting place.
- ^[Complete the search.
- **^Q** Quote a search command.
- **ctrl-XX** Complete the search and execute the XX command.
- **other** Add the character to the search string.

^T Interchange the characters on each side of the point, leaving the point after the second one.

^U Universal argument. There are two forms:

- *^U ^U .... <cmd>* does <cmd> 4, 16, 64, 256, ... times depending upon the number of ^Us (each ^U is a multiplier by 4).
- *^U <integer> <cmd>* does <cmd> <integer> times. (*e.g.*, ^U 3 5 ^F means to ^F 35 times).

^V Move the bottom of the current screen to the top of the screen: *i.e.*, move down one screen.

^W Delete the text between the point and the mark ("cut").

^X Prefix for the ^X commands listed below.

^Y Copy the top item from the kill ring to the point; place the mark at the beginning of the block and the point at the end ("paste").

^Z Suspend the program's execution and return to whatever invoked the editor.

^[Prefix for the Meta commands listed below.

^\ Undefined.

^] Aborts a recursive edit.

^^ Undefined.

^_ Undo.

SP ... ~ Insert themselves.

^? Delete the preceding character.

BS Same as ^H, also BACK SPACE.

TAB Same as ^I.

LF Same as ^J, also LINE FEED.

CR Same as ^M, also CARRIAGE RETURN or RETURN.

ESC Same as ^[, also ESCAPE.

DEL Same as ^?, also DELETE or RUBOUT (obsolete).

Help Commands:

^H ^C Describe the copying policy.

^H ^D Describe the distribution policy.

^H ^H Help on help.

^H ^N View Emacs news.

^H ^W Describe the (lack of) warranty.

^H ? Help on help.

^H A Apropos (which commands deal with ...?).

^H B Describe the current key bindings.

^H C Briefly describe a key.

^H D Describe a function.

^H F Describe a function.

^H I Invoke the "info" subsystem.

^H K Describe a key.

^H L Describe problems with the (current) version ("lossage").

^H M Describe a mode.

^H N View Emacs news.

^H S Describe syntax.

^H T Tutorial.

^H V Describe a variable.

^H W Where is ...?

Control-X (^X) Commands:

^X ^@ Flush mouse queue.

^X ^A Add mode abbreviation.

^X ^B Display a list of all buffers and associated information.

^X ^C Exit editor.

^X ^D Display the current directory.

^X ^E Evaluate the last S-exp.

^X ^F Ask for the name of a file and read it into a new buffer whose name is derived from the file name.

^X ^G Cancel ^X prefix.

^X ^H Remove mode abbreviation.

^X ^I Indent rigidly

^X ^L Convert the region to lower case.

^X ^N Set the goal column.

^X ^O Delete the blank lines around the point.

^X ^P Place the point and the mark around the current page.

^X ^Q Toggle the "read only" marker.

^X ^R As ^X ^F, but mark the file "read only."

^X ^S Save the current buffer if it has been modified.

^X ^T Transpose lines.

^X ^U Convert the region to upper case.

^X ^V Find alternate file.

^X ^W Ask for the name of a file and write the buffer to that file.

^X ^X Exchange the point and mark.

^X ^[Repeat a complex command.

^X $ Set selective display.

^X ' Expand an abbreviation.

^X (Start collecting a keyboard macro.

^X) End collecting a keyboard macro.

^X + Add global abbreviation.

^X - Remove global abbreviation.

^X . Set the fill prefix.

^X / Point to register.

^X 0 Delete window.

^X 1 Delete other windows.

^X 2 Split window vertically (one above the other).

^X 3 Split window vertically (one above the other), but stay in the first.

^X 4 Prefix for the Control-X 4 commands listed below.

^X 5 Split window horizontally (one beside the other).

^X ; Set the comment column.

^X < Scroll the window left.

^X = Display the current cursor position.

^X > Scroll the window right.

^X A Append the region to a buffer.

^X B Switch to a buffer.

^X D Edit directory ("DIRED").

^X E Invoke the last keyboard macro.

^X F Set the fill column to the horizontal position.

^X G Insert a register.

^X H Place the point and the mark around the entire buffer.

^X I Insert a file.

^X J Register to point.

^X K Kill a buffer.

^X L Count the number of lines in the page.

^X M Send electronic mail.

^X N Narrow the editing bounds to the region.

^X O Switch to the other window.

^X P Narrow the editing bounds to the page.

^X Q Keyboard macro query.

ˆ**X R** Copy a rectangle to a register.

ˆ**X S** Save some buffers.

ˆ**X U** Advertised undo.

ˆ**X W** Widen the editing bounds.

ˆ**X X** Copy to a register.

ˆ**X** [Move backward one page.

ˆ**X**] Move forward one page.

ˆ**X** ˆ Grow the current window.

ˆ**X ‘** Move to the location implied by the next error message.

ˆ**X {** Shrink a window horizontally.

ˆ**X }** Grow a window horizontally.

ˆ**X** ˆ**?** Delete to the beginning of the current sentence.

Control-X 4 Commands:

ˆ**X 4** ˆ**F** Find file other window.

ˆ**X 4 .** Find tag other window.

ˆ**X 4 A** Add change log entry other window.

ˆ**X 4 B** Switch buffer other window.

ˆ**X 4 D** DIRED other window.

ˆ**X 4 F** Find file other window.

ˆ**X 4 M** Mail other window.

Meta (ˆ[) Commands:

ˆ[ˆ**@** Place the point and the mark around the S-exp.

ˆ[ˆ**A** Move to the beginning of the current defun.

ˆ[ˆ**B** Move backward one S-exp.

ˆ[ˆ**C** Exit recursive edit.

^[**^D** Move down one level of list structure.

^[**^E** Move to the end of the current defun.

^[**^F** Move forward one S-exp.

^[**^G** Format ("grind") the current S-exp.

^[**^H** Place the point and the mark around the S-exp.

^[**^I** Complete a Lisp symbol.

^[**^J** Indent a new comment line.

^[**^K** Delete the following S-exp.

^[**^N** Move forward one list.

^[**^O** Split line: move the rest of this line vertically down.

^[**^P** Move backward one list.

^[**^Q** Indent an S-exp.

^[**^S** Incremental search forward using regular expressions.

^[**^T** Transpose the adjoining S-exp.

^[**^V** Scroll the other window.

^[**^W** Append the next delete to the top item on the kill ring.

^[**^X** Evaluate a defun.

^[^[Evaluate an expression.

^[^\ Indent the region.

^[**SP** Insert exactly one space.

^[**!** Ask for and execute a shell command.

^[**$** Invoke the spelling checker on a word.

^[**%** Query replace: ask for an old string and a new string. At each occurrence of the old string, display it and ask for a command:

  ^L Redisplay the screen.

^R Invoke the editor recursively.

^W Delete the old string and invoke the editor recursively.

^[Exit.

SP Replace the old with the new and continue.

, Replace and wait for confirmation.

. Replace and exit.

! Replace the rest without asking.

^ Return to previous old string (jump to mark).

^? Don't replace but continue.

^[**'** Set abbreviation prefix mark.

^[**(** Insert paired parentheses.

^[**)** Move past the closing parenthesis.

^[**,** Tags loop continue.

^[**-** Make the argument negative.

^[**.** Find a tag.

^[**/** Abbreviation expand.

^[**0 .. 9** Use digits as argument (similar to ^U).

^[**;** Indent for comment.

^[**<** Move to the beginning of the current buffer.

^[**=** Count the lines within the region.

^[**>** Move to the end of the current buffer.

^[**@** Place the point and mark around the current word.

^[**A** Move to the beginning of the current sentence.

^[**B** Move backward one word.

^[**C** Capitalize the following word.

^[**D** Delete the following word.

^[**E** Move to the end of the current sentence.

^[**F** Move forward one word.

^[**G** Fill text in the region.

^[**H** Place the point and the mark around the current paragraph.

^[**I** Tab to tab stop.

^[**J** Indent new comment line.

^[**K** Delete the remainder of the current sentence.

^[**L** Convert the following word to lower case.

^[**M** Move back to indentation.

^[**Q** Fill the current paragraph. ^U ^[Q means to justify the paragraph.

^[**R** Move to window line.

^[**T** Transpose the adjoining words.

^[**U** Convert the following word to upper case.

^[**V** Move the top of the current screen to the bottom of the screen: *i.e.*, move up one screen.

^[**W** Copy the region to the kill buffer.

^[**X** Execute extended command.

^[**Y** After ^Y: Delete the just-yanked text and yank the previously killed text.

^[**Z** Zap to character.

^[[Move to the beginning of the current paragraph.

^[\ Delete the spaces and tabs around the point.

^[ ] Move to the end of the current paragraph.

^[^ Delete the indentation on the current line.

^[| Execute a shell command on the region ("pipe the region through a shell command").

^[~ Clear the buffer modified flag.

^[**^?** Delete the previous word.

The Author's Command Set

This section lists the changes the author makes to the just-presented default command set.

ˆC Rotate case of word: foo -> Foo -> FOO -> foo ...

ˆH Delete the previous character.

ˆI Just insert a TAB: no special indentation.

ˆJ Insert a line break and indent the new line the same as the current one.

ˆK Delete lines as usual, but don't treat an argument specially.

ˆM Just insert a line break: no special actions.

ˆN Move to the next line: don't extend the buffer.

ˆT Always transpose the two preceding characters.

ˆV A slightly different move down screen.

ˆZ Move up screen.

ˆ\ Delete all white space (indentation) on the current line.

ˆ] Invoke keyboard macro.

ˆ_ Suspend the program's execution and return to whatever invoked the editor.

ˆ? Delete the previous character with no special actions.

ˆX ˆE Execute one shell command.

ˆX ˆI Insert a file.

ˆX ˆM Invoke a subshell.

ˆX ˆN Null (to prevent typing by accident).

ˆX ˆP Null (to prevent typing by accident).

ˆX ˆR Re-read file.

ˆX ˆT Return to top-level editing (exit all recursive editors).

^X \ Get rid of all "^H_" strings in the buffer (used to make UNIX man pages more readable after doing "^X ^E man *title*").

^X C Invoke compiler.

^X ^H Help.

^X N Null (to prevent typing by accident).

^X P Null (to prevent typing by accident).

^X R Read electronic mail.

^[**^H** Delete previous word.

^[**^R** Query replace.

^[^[Null (to prevent typing by accident).

^[**SP** Set the mark.

^[< Move to the beginning of the buffer. With an argument, move to the specified line number.

^[= Display the number of lines in the buffer and the number of the line that the point is on.

^[> Move to the end of the buffer. With an argument, move to the specified line number.

^[**G** Ask for a line number and go to the specified line.

^[**I** Null (to prevent typing by accident).

^[**J** Null (to prevent typing by accident).

^[**M** Null (to prevent typing by accident).

^[**N** Null (to prevent typing by accident).

^[**O** Null (to prevent typing by accident).

^[**P** Null (to prevent typing by accident).

^[**R** Replace string (as ^[^R, but don't ask).

^[**S** Center the current line.

^[**Z** Null (to prevent typing by accident).

^[\ Delete all surrounding spaces, tabs, and line breaks and re-insert one space.

Appendix D: The TECO Command Set

This appendix presents a summary of the MIT TECO editor's command set. Should you actually find an ITS or TOPS-20 system and wish to run TECO on it, this appendix will be useful but not completely replace the full manual.

TECO implements the "one-dimensional array of bytes" editing model. Line breaks are stored as single newline characters. Large files are divided into *pages*. Pages within a file must be edited in order (*i.e.*, all editing on the first page is done, then the second page is read in and edited, etc.). The only way to go backwards is to finish editing the file and to start over. On the other hand, on most systems, only very large files (over 64 Kilobytes) are split into pages.

Commands are single or double characters. Upper/lower case is ignored. The basic forms of commands are "C", "nC" and "n,mC". The first form executes command "C" with the default arguments, the second supplies one argument of "n", and the third supplies two arguments of "n" and "m".

Some commands take string arguments. The string consists of all characters after the command up to the terminator character. This character, called *altmode*, is the Escape character, and is written as "$" (for historical reasons, of course).

Commands are accumulated into a *command string*, until a double altmode terminator is entered, at which time the commands are executed. One of the altmodes may serve to delimit a string. For example:

```
5C$$
```

moves forward five characters, and the command string:

```
Ithen$$
```

inserts the string "then", and the string:

```
5CIthen$8R$$
```

moves forward five characters, then inserts the string "then", then moves backward eight characters.

TECO also supports the concept of *Q-registers*. These are variables that can either hold arbitrarily large strings of text or numbers. Each Q-register can either hold a string or a number and may change back and forth as desired. However, it may only hold one or the other at any given time. Q-registers are named by single characters. When holding strings, Q-registers act just like buffers: you can switch to, insert into, delete from, move around in, display, and search in them. Q-register names are either

- one alphanumeric character preceded by zero, one, or two periods (names with two periods are reserved for system variables), a "variable name" of the form $name$ (these are dollar signs, not altmodes), ‘
- a subscripting expression such as :Q(index),
- a * (for certain commands, it causes them to return their data instead of storing it),
- an expression in parentheses (for certain commands), or
- up to 3 periods followed by a ˆR or ˆˆ and any ASCII character.

TECO was first written for PDP-10s running ITS (the Incompatible Timesharing System) and its command set incorporates some knowledge of that system's file name syntax. Briefly, an ITS file name has these four parts:

```
DEV;DIR:PRTONE PRTTWO
```

Each part can be up to 6 characters long and is stored in one 36-bit word. Within that word, each character is squeezed into a 6-bit character set, so lower-case characters are folded into their upper-case equivalents. The operating system and applications programs maintain a default value for each of the four parts. For example, if a user specifies a name of "foo", the default device, directory, and second part of the file name are automatically filled in. Multiple versions of a file are kept by setting the second part of the file name to a number. Successive versions are maintained by incrementing the number. A second part of "<" refers to the earliest version around. A second part of ">" refers to the latest version around (for reading) or one past the latest version around (for writing).

General notation:

ˆX Denotes the specified control character (see Appendix E for a listing of all ASCII characters).

$ Denotes the altmode character unless otherwise specified.

| Denotes a choice (either the form on the left or the form on the right are acceptable).

m | **n** | **arg** Denote integer arguments.

cmd An arbitrary command string.

dir Denotes a directory.

file Denotes a file name.

k Denotes either "m,n" or "n": either a text range of characters m through n or n successive lines.

string Denotes a string argument.

: | **@** Modify the operation of certain commands.

Commands

n^@ Argument: if n > 0, same as ".,.+n". If n <= 0, the same as ".+n,."

m,n^@ Returns the value n - m.

m,n:^@ Returns the value n,m.

^A Logical xor operator.

^B Used for cleaning up after failed searches.

^C When typed from the console, it terminates the command string and starts execution.

**^Fstring$** Inserts its string argument, after deleting the last string found or inserted. Same as "FKDIstring$".

^G Executes immediately. Erase's the command string as typed so far. Also aborts current command if one is executing.

^H Inserts a Back Space character.

^I Inserts a Tab character.

^J Flushes any pending values.

^Kstring$ Executes string as a system command.

^M Flushes any pending values.

n^N Sets FS LINES$ to n.

:^N Toggle the FS TTMODE$ flag.

:n^N Does n^N:^N.

^Ofilename$ Bigprints filename on the device open for output.

^Pcmd0$cmd1$cmd2$ ASCII sort command. Assuming the point is at the start of a record, cmd0 should move the point to the start of the key, cmd1 should move the point to the end of the key, and cmd2 should move the point to the end of the record (= start of the next record). If FS ^P CASE $ is non-zero, then this command ignores case.

^Q Quotes the next character.

^R Invokes Real time (Emacs-like) mode.

n^S If n > 0, sleeps for n 30ths of a second. Otherwise, sleeps until system uptime is >= -n.

n:^S Sleeps for at most n 30ths of a second, returning immediately when input becomes available.

^U Displays the current directory in a user supplied manner. Executes immediately if it is the first character in a command string.

^V Pops the next position off of the ring buffer of positions. Successive ^Vs move through the 8-item ring.

:^V Returns the value on the top of the ring.

n^V Pushes n onto the ring buffer unless n is the same as the value on the top of the ring.

n:^V Pushes n onto the ring buffer.

^W Returns to the top level.

^X Its value is m, the first argument to m,nMq. It is only valid inside macros.

^Y Its value is n, the second argument to m,nMq. It is only valid inside macros.

^Z Suspends the editing process.

$ (^[) Terminates text argument; two successive altmodes terminates a command string.

^ Exits from the innermost macro invocation.

^]x Specially processes character x.

^]^X Reads and returns the string argument which follows kMq. It is only valid inside macros.

^^x Returns the ASCII value of "x".

SP Same as "+", except that just a space is not an argument.

!label! Defines a label or a comment. It is a comment if no command attempts to go there.

arg"x then-cmd ' Conditional. It checks the arg according to condition x. Executes the command string then-cmd if the condition is true. Arg is discarded. Conditionals are:

- **B** is arg the ASCII value of a delimiter (..D)?
- **C** is arg the ASCII value of a non-delimiter?
- **D** is arg the ASCII value of a digit?
- **E** is arg == 0?
- **G** is arg > 0?
- **L** is arg < 0?
- **N** is arg != 0?
- **U** is arg the ASCII value of an upper-case character?
- **:x** Reverses the meaning of condition x

arg"x then-cmd '"# else-cmd' Conditional with else.

Logical or operator.

%q Increments the numeric contents of q and returns the result.

& Logical and operator.

' Terminates a conditional.

(|) Specify precedence for argument operators.

***** Multiplication operator (no precedence).

+ Addition operator.

, Separates numeric arguments.

- Subtraction operator.

. (by itself) Specifies the position (number) of the point. See also the special .. Q-register names listed below.

/ Division operator (no precedence).

0-9, . Digits. XXX is interpreted in base FS IBASE$ (usually 10); XXX. is interpreted in base FS I.BASE$ (usually 8).

: Modifies the action of certain commands.

n; Does nothing if n < 0. Otherwise, it passes control to the character after the next >. In other words, it is used to terminate iteration. If n is null, it uses the value of the last search.

:; Like ;, but reverses the condition.

@@; Like ;, but exit if arg == 0.

:@@; Like ;, but exit if arg != 0.

n<cmd> Does command cmd n times, or indefinitely if n is null.

:<cmd> Begins errset. If an error occurs inside <>, execution will resume after the >.

k= Types k.

k:= Types k, omits CR/LF.

k@= Types k in the echo area.

k@:= Types k in the echo area, omitting CR/LF.

? If first character after an error message, displays the last several command characters. Otherwise, enters trace mode.

:? Leaves trace mode.

@ Modifies the action of certain commands.

A Appends the next page of the input file to the buffer.

n:A Appends the next n lines (up to a page marker) from the input file to the buffer.

@A Appends all of the input file to the buffer and closes the input file.

mA Returns the ASCII value of the character m characters to the right of the point.

0A Returns the ASCII value of the character at the point.

B Argument; equivalent to 0 (*i.e.*, the beginning of the buffer). However, its value is modified by the virtual buffer boundaries.

nC Moves forwards n characters.

n:C Same as moving, but returns -1 if the move succeeds or 0 if the move would fail.

mD Deletes forward m characters.

-mD Deletes backward m characters (there is no equivalent to R).

E... See E-Commands listed below.

F... See F-Commands listed below.

Gq Inserts the contents of Q-register q into the buffer. If q has a number, its string representation is inserted. FS INSLEN$ is set to the length of the inserted text.

m,nGq Inserts the range m to n from Q-register q.

:Gq Returns a copy of the string in Q-register q.

n:Gq Returns the value of the character at position n in Q-register q.

H Argument: wHole buffer: equivalent to "B,Z".

Istring$ Inserts the string at the point.

@Ixstringx Inserts the string delimited by the "x" characters at the point (lets you insert a string that contains an altmode).

nI Inserts the character with ASCII value n.

m,nI Inserts m copies of character n.

:Iq Inserts the Q-register q into the buffer.

n:Iq Inserts the character with ASCII value n into Q-register q.

m,n:Iq Inserts m copies of character n into Q-register q.

:Iqstring$ Inserts the string into Q-register q, replacing any prior contents.

@n:Iq Inserts the character whose ASCII value is n into Q-register q, replacing any prior contents.

@m,n:Iq Same as n:Iq, but inserts m copies.

@:Iqxstringx Same as :Iqstring$, except that string is delimited by the characters x.

nJ Sets the point to the specified position (BJ or just J is move to start; ZJ is move to end).

n:J Does the set and returns -1 if successful and 0 if not.

m,nK Kills (deletes and saves the deleted text) the characters in the range; moves the point there.

nK Kills what L would move over.

n:K Kills what :L would move over.

@K Like K, but only LFs preceded by CRs are recognized.

mL Moves to start of mth line after the point.

0L Moves to the start of the current line.

m,nL Same as m+n-.J, used by some other commands.

m:L Moves to the end of the m-1th line.

0:L Moves to the end of the *previous* line.

@L Like L, but only LFs preceded by CRs are recognized.

m,nMqstring$ Executes the contents of Q-register q as TECO commands. If the Q-register contains a number, it executes the corresponding ^R mode command.

:M Tail-recursive form of M. Like M then ^\, but the current function is removed from the stack before the new one is called.

@M Fools the called macro into thinking it was called from ^R mode.

**nNstring$** Same as nSstring$, but it does P and continues the search if the end of the buffer is reached.

Olabel$ Goes to the specified label. Generates an error if the label is not found.

:Olabel$ Returns if the label is not found.

@Olabel$ Allows the label to be abbreviated.

nP Writes out the buffer and a ^L (page mark), kills the buffer, and reads one page from the input file. All of this is repeated n times.

m,nP Writes out the specified range of the buffer, but does not kill it or append input.

nPW Writes out the buffer and a ^L (page mark), no killing or reading. All of this is repeated n times.

@P As P, except that the low-order bit of each word written should be preserved and not cleared. Used for writing binary files.

Qq Returns the value in Q-register q as a number. If the Q-register holds text, this returns the pointer to that text.

nR Moves backwards n characters (same as -nC).

n:R Same as moving, but returns -1 if the move succeeds or 0 if the move would fail.

nSstring$ Searches forward for the nth occurrence of the string and places the point after the string. If the argument is null, the last non-null argument is used. Special characters in search strings:

- **^B** Matches any delimiter (see ..D).
- **^O** Divides string into alternate patterns. Thus, Sfoo^Obar$ will find the first of ”foo” or ”bar”.
- **^Qx** Quotes x.
- **^X** Matches any character.
- **^Nx** Matches any character except for x, where x is any character.
- **^N^B** Matches any non-delimiter.
- **^N^X** Matches nothing.

Note that Sfoo^O$ will always succeed and will move the point over the next three characters if and only if they are ”foo”. -2-(:Sfoo^O$) will do that and return non-zero if they were ”foo”.

**-nSstring$** Same as nSstring$, but searches backwards and leaves the point before the string.

n@Sxstringx Same as nSstring$, but the string is delimited by the character specified by the ”x”s. If the argument is null, searches for the null string.

**n:Sstring$** Same as nSstring$, but returns the value -1 if successful or 0 if not. If a ^O is used within -n is returned if the nth subpart is found.

kT Types out the text in the range (n lines or m,n characters).

@T Types out in the echo area.

nUq Inserts number n into Q-register q, returns no value.

m,nUq Inserts number n into Q-register q, returns m.

kV Displays what T would type. Puts ”/\” where point is and does ”–MORE–” processing.

@V Does standard buffer redisplay.

kVW Does V, then waits for one character and returns its ASCII code as a value.

W Flushes current value except when part of PW or VW.

kXq Inserts text range k into Q-register q, replacing any prior contents.

k@Xq Same as kXq, but appends to q.

Y Kills the buffer and reads one page from the input file into the buffer.

@Y Kills the buffer and reads the rest of the input file into the buffer.

Z Argument: specifies the length of the buffer in characters.

[q Push the text or number from Q-register q onto the Q-register push down list.

\ Moves past number, returns its value.

n Inserts a printed representation of character n (in base ..E).

m,n Is like n\, but pads with spaces to m columns.

: Returns the representation of n as a string instead of inserting it.

]q Pop the text or number from the Q-register push down list into Q-register q.

ˆ Replaced by @.

**n_string$** Same as nSstring$, but it does Y if the end of the buffer is reached.

ˆ**?** Erases the last character of command string.

E-Commands (most file commands are here)

E^Udir$ Displays the specified directory in a user-defined manner.

E?file$ Tries to open file. Returns 0 if successful or an error code if not.

EC Closes the input file.

EDfile$ Deletes the specified file.

:ED Deletes the currently open file.

**EEfile$** Same as the sequence infinityPEFfile$EC.

EFfile$ Closes the output file and changes its name to file.

EG Inserts various information into the buffer on successive lines: the date as YYMMDD, the time as HHMMSS, the current username, the default file names, the real names of the files open for input and output, the date in text form, a 3-digit value (day of week, day of week of 1st of this year, leap year status), and the phase of the moon. There are better ways of getting most of this information.

EI Opens a file "_TECO_ OUTPUT" for writing on the default device.

:EI Same as EI, but uses the current default file name.

@EI Same as EI, but sets the default device to DSK:

EJfile$ Restores the complete state from the file, which must have been saved with @EJ.

@EJfile$ Saves TECO's complete state to the file.

EL Types out a listing of the default directory.

EM Inserts a listing of the default directory into the buffer.

ENoldnew Renames file old to file new.

**EPfile$** Does ERfile$, then bigprints the file name twice on the device open for writing.

EQfromto Creates a link from the file "from" to the file "to".

ERfile$ Opens a file for input.

ETfile$ Sets the default file name to the specified file name.

EWdir$ Same as EI, but with the specified directory.

:EWdir file$ Same as EW, but with the specified file name.

EYdir$ Types out a listing of the specified directory.

EZdir$ Inserts a listing of the specified directory into the buffer.

E[Push the input channel.

E Push the output channel.

E] Pop the input channel.

E^ Pop the output channel.

E_oldnew Copies file old to file new.

:E_oldnew Copies file old to file new, preserving the old file's date.

F-Commands

m,nF^@ Returns m and n in numerical order, such that the new m will be > n.

nF^@ Returns 2 arguments that specify the range from the point to the location n lines away.

F^A Runs every character in the buffer through a dispatch table.

nF^Bstring$ Searches in string for the character whose ASCII value is N.

@F^Bstring$ Searches the buffer forward for a character not in string.

-@F^Bstring$ Searches the buffer backward for a character not in string.

m,n@F^Bstring$ Searches the buffer in the range for a character not in string.

F^Estring$ Overwrites the next length-of-string characters with string. Same as deleting and inserting, but the gap does not need to move.

F^K Reads a string argument from within a macro.

m,nF^Sq Searches Q-register q for a word that contains n, starting at m.

F^X Its value is k, all arguments to m,nMq. It is only valid inside macros.

F^Y Its value is the number of arguments to m,nMq. It is only valid inside macros.

string:F^^ Determines whether string is a short Q-register name.

argF"x Same as regular conditional, but passes the arg to the then or else command string.

F$ (dollar) Returns FS CASE$ and inserts in the buffer the case shift and lock characters, if any. If FS CASE$ is non-zero, all characters are converted to uppercase (if > 0) or lower case (if < 0) on input. The case-shift character causes the next character to be read in the other case. The case-lock character temporarily complements the preferred case. On output, if FS CASE$ is odd, characters in the non-standard case will be preceded by case-shifts. If even, no translation is done.

nF$string$ Sets FS CASE$ to n and sets the case shift and lock characters to the first two characters in string.

F(Is like (, except that F(returns its arguments, making it easy to use a value twice without using a Q-register.

F) Is like), except that F) returns its arguments exactly, discarding the data saved by (.

F*string$ Reads and discards a string argument.

F+ Clears the screen.

F6string$ Returns string with the first six characters packed into a word (this TECO is running on a 36 bit machine).

nF6 Expands n into an ASCII string and inserts it into the buffer.

n:F6 Expands n into a string.

F;tag$ Throws to the tag. This is a ”long jump.”

F<!tag!cmds> Catches a throw and executes the commands.

:F<!tag!cmds> Is an errset and a catch at the same time.

F=qstring$ Compares the Q-register q to string. Returns 0 if ==, positive if q is > string, or negative if q is < string. If value is not zero, the value's absolute value is 1 + location of the difference.

@F=qxstringx Compares the Q-register q to string delimited by x.

m,nF=string$ Compares the buffer in the range m to n to string.

m,n@Fxstringx Compares the buffer in the range m to n to string delimited by x.

F? Same as 30F?

0F? Same as 30F?

nF? Mbox control. Argument is a bit string, bits:

- **bit 2^0** close the gap
- **bit 2^1** run garbage collect
- **bit 2^2** clear the jump cache
- **bit 2^3** flush unused core
- **bit 2^4** close the gap if it is >5000

m,nFA Justifies text within the range.

nFA Justifies n lines of text.

@FA Fills without justification.

**kFBstring$** Same as Sstring$ in the domain defined by k. If k is of the form m,n and m > n, search backwards. ":" and "@" modifiers work.

kFC Converts text range k to lower case.

k@FC Converts text range k to upper case.

n:FC Returns the upper-case version of the character whose ASCII value is n.

nFD Returns the range ".,x", where "x" is the position just after the nth level down in parenthesis after the point.

-nFD Goes backward.

FE Inserts a list of all TECO error messages into the buffer.

nFE Inserts only the message whose error code is n.

@FEstring$ Returns the code of the error whose message is string.

FG Process an error.

@FG Process an error and throw away type ahead.

FI Reads one character and returns its ASCII value.

:FI As FI, but don't flush the character (it will be re-used).

@FI As FI, but returns the value in the 9-bit TV character set.

@:FI As @FI, but don't flush the character (it will be re-used).

FJ Insert the command line used to invoke TECO into the buffer.

FK Returns the value - FS INSLEN$, *i.e.*, length of the last string inserted or found by a search or FW. FK is always < 0 except for a backwards search or FW.

nFL Returns the range ".,x", where "x" is the position just after the nth list after the point.

-nFL Goes backward.

n@FL As nFW, but does S-expressions.

nFLD Same as nFLK.

nFLK Kills what nFL implies.

nFLL Does the move implied by the nFL.

nFLR Same as nFLL.

nFLXq Combines nFL with Xq.

m,nFM Attempts to move the point so that the cursor will appear at column n, m lines below where you started.

FN Is the same as "[..n:I..N". It is needed to eliminate the possibility of a ^G within the string.

FOqname$ Performs a binary search of a table of fixed-length entries. It is intended for symbol tables. Q-register q contains the table and "name" is what should be searched for. The first word of the table contains the number of words for each entry in the table.

objectFP Returns a number describing object. Values:

- **-4** A number, none of the below.
- **-3** A number that could be in pure string space.
- **-2** A number that could be in impure string space.
- **-1** A dead buffer.
- **0** A living buffer.
- **1** A Q-vector.
- **100** A pure string.
- **101** An impure string.

FQq Its value is the number of characters in Q-register q or -1 if the Q-register holds a number.

FR Updates the display.

FSname$ Returns the value of the specified variable (listed below).

FTstring$ Types its string argument.

:FTstring$ Types its string argument at the top of the screen.

@FTstring$ Types its string argument in the echo area.

@:FTstring$ Types its string argument in the echo area but only if no input is available.

nFU Returns the range ".,x", where "x" is the position just after the nth level up in parenthesis after the point.

-nFU Goes backward.

FVstring$ Displays its string argument.

:FVstring$ Displays its string argument, then clears the rest of the screen.

nFW Returns the range ".,x", where "x" is the position just after the nth word after the point.

-nFW Goes backward.

n:FW As nFW, but only does n-1 words.

n@FW As nFW, but does Lisp atoms and not words.

nFWD Same as nFWK.

nFWK Kills what nFW implies.

nFWL Does the move implied by the nFW.

nFWR Same as nFWL.

nFWXq Combines nFW with Xq.

kFXq Same as X and K combined: kXqkK.

FY Inserts all that remains of the input file before the point.

nFY Inserts at most n characters.

FZfile string$ Creates and starts a non-exec fork.

FZ$ Resumes the inferior fork.

F[flag$ Pushes the value of FS flag on the Q-register PDL.

nF[flag$ Pushes and sets the flag to the new value.

nF[^R CMACRO$ Pushes the definition of the character whose number is n.

m,nF[^R CMACRO$ Pushes and sets.

F_ Mostly the same as _, but keeps working regardless of the setting of FS _DISABLE$.

F]flag$ Pops the value of FS flag from the Q-register PDL.

nF]^R CMACRO$ Pops the definition of the character whose number is n.

F~ Like F=, but both strings are compared as if converted to upper case.

Special Q-registers, names are of the form "..x"

..0 ^P puts its three arguments into these.

..1

..2

..A Holds the string to represent the cursor (default is "/\").

..B Holds the macro to display the user buffer.

..D Holds the delimiter dispatch table, which tells several commands (FW, FL, "B, "C and search ^B) how to treat ASCII characters.

..E Holds the output radix for = and \.

..F Holds the ^R secretary macro. Can be used for auto save.

..G Holds the user-specified directory display macro.

..H Is the "suppress-display" flag.

..I Holds the value of . at the start of the command.

..J Holds user-specified label for –MORE– processing.

..K Holds deleted text.

..L Executes when TECO first starts.

..N Macro that to be executed when another macro exits.

..O The current buffer.

..P Holds the user-defined error-handler macro.

..Q Holds the symbol table used to define TECO variables.

..Z Safety backup copy of ..O.

FS Variables

Names can be up to six characters long. Spaces in names are ignored. Only as much of a name as is required to make it unique is required, although programs should include the entire name. Saying FSname$ returns the value of the flag. Saying nFSname$ or m,nFSname$ sets the value. If a flag can be set and you want to use the flag as the second operand of an arithmetic operator (*e.g.*, .+FSname$C), enclose the FS in parentheses (.+(FSname$)C).

These names can never include control characters. The "^" in some of the names is a leading caret. However, the combination usually relates to the implied control character.

% BOTTOM Size of the bottom margin as a percentage of the number of lines being displayed.

% CENTER Where TECO should prefer to put the cursor.

% END Size of the area at the bottom of the screen, such that TECO should never choose to put the cursor there.

% OPLSP (Read only) Non-zero if the input is coming from a Lisp job

% TOP Size of the top margin (analogous to %BOTTOM).

% TOCID (Read only) Non-zero if the terminal can insert and delete characters.

% TOFCI (Read only) Non-zero if the terminal can generate 9-bit characters.

% TOHDX (Read only) Non-zero if the terminal is half-duplex.

% TOLID (Read only) Non-zero if the terminal can insert and delete lines.

% TOLWR (Read only) Non-zero if the terminal can generate lower case characters.

% TOMOR (Read only) Non-zero if the use wants –MORE– processing.

% TOOVR (Read only) Non-zero if the terminal can overprint.

% TOROL (Read only) Non-zero if the user has selected scroll mode.

% TOSAI (Read only) Non-zero if the terminal can print the SAIL character set.

***RSET** Initially 0. If set to non-zero, trace information is not cleared automatically.

.CLRMOD Normally -1. If negative, screen is normally cleared automatically. If 0, automatic screen clears are not done (used for debugging). If positive, the screen is never cleared.

.KILMOD Normally -1. If 0, FS BKILL$ doesn't actually do the kill.

.TYI BACK Backs up the point FS .TYI PT$ by one step. After backing up n steps, you can use FS .TYI NXT$ to re-get those n input characters.

.TYI NXT Extracts one character from the ring buffer of past input characters.

.TYI PT Pointer into the ring buffer that contains the last 60 input characters.

:EJ PAGE Is the number of the lowest page used by :EJ'd shared pure files.

ADLINE Is the line size used by the FA command.

ALTCOUNT Is the number of $$s that TECO has seen at interrupt level.

BACK ARGS (Read only) Returns the arguments to a macro in a different stack frame (*i.e.*, one of the macros that was called that eventually called you). Returns 0, 1, or 2 values in the same ways that F^X does. If the argument to this is 0 or positive, it returns the arguments for the specified frame number (0 is outermost). If negative, returns the arguments for the relative frame number (-1 is your caller).

BACK DEPTH (Read only) Returns the number of stack frames, not counting you.

BACK PC Returns the PC of the stack frame that is specified in the same way as FS BACK ARGS$. m,nFS BACK PC$ sets the PC to m.

BACK QP PTR (Read only) Specifies where a ^\ will return to. Arguments are as for FS BACK ARGS$.

BACK RETURN (Write only) Returns from the specified stack frame. Arguments are as for FS BACK ARGS$.

-1 FS BACK RETURN$ is equivalent to ^\. BACK STRING (Read only) Returns a pointer to the string or buffer being executed. Arguments are as for FS BACK ARGS$.

BACKTRACE Returns a copy of the program being run by the stack frame. Arguments are as for FS BACK ARGS$.

BBIND is useless, but F[B BIND$ and F]B BIND$ are useful for pushing to and popping from a temporary buffer.

BCONS (as in n FS BCONS$) returns a new buffer n characters long. It is initially filled with 0's (NULs).

BCREATE is like FS BCONS$ U..0. In other words, the buffer is selected instead of returned.

BKILL Kills the specified buffer.

BOTHCASE Initially 0. If == 0, case is significant during searches. If > 0, case is ignored. If < 0, case of special characters (@[\]^_ and ‘{|}~^?) is also ignored.

BOUNDARIES Reads or sets the virtual buffer boundaries.

BS NO LF If non-zero, suppresses the LF that follows any backward motion or rubbing out in ^R mode on printing terminals.

CASE Like F$, but neither gets nor sets the case-shift or case-lock characters.

CLK INTERVAL Is the interval between real time clock ticks in 1/60 seconds. Only active during user input.

CLK MACRO Is the real-time interrupt handler macro. If the macro types anything out, it must not leave ..H set.

CTL MTA If negative, it suppresses the ^R mode definitions for all control-meta characters. This makes it easy to edit TECO commands.

DATA SWITCHES (Read only) The contents of the PDP-10 console switches.

DATE (Read only) The current date and time as a number in file-date format. It can be fed to FS FD CONVERT$ or FS IF CDATE$.

D DEVICE Is the default device name.

DD FAST (Read only) Is non-zero if the current device is fast (*i.e.*, local disk).

D FILE Is the default file name.

D FN1 Is the default file name first part.

D FN2 Is the default file name second part.

D FORCE Setting this to non-zero forces a complete redisplay of everything except the mode line. It is used for putting up temporary displays.

D SNAME Is the default sname.

D VERSION Is the default versions number, a reflection of FD D FN2$. If the latter is numeric, reading this value returns the corresponding number. If it is ”>” or ”<”, this value is 0 or -2, respectively. If it is not numeric, this returns -1. if FD D FN2$ is numeric, setting this value sets the file name. Otherwise, the setting is ignored.

D WAIT When set to non-zero, causes the display to pause slightly between lines of output.

ECHO ACTIVE When set to non-zero, indicates that output has been written to the echo area, so the echo area needs to be cleaned up.

ECHO CHAR When a ^R mode character is being executed, this value holds the character that caused the invocation.

ECHO DISPLAY (Write only) As for FS ECHO OUT$, but outputs in display mode.

ECHO ERRORS When set to non-zero, error messages are printed in the echo area.

ECHO FLUSH When set to non-zero, automatic clearing of the echo area in ^R mode is enabled.

ECHO LINES Then number of lines at the screen bottom that can be used for command echoing.

ECHO OUT (Write only) Used for outputs to the echo area. If it has a numeric argument, the argument is the ASCII code of a character to echo. With a string argument, the string is echoed.

ERR Same as FS ERROR$ if read. If written to, creates an error with the specified error code.

ERRFLAG When negative, signals to redisplay that the first -n lines of the display contain an error message and should not be overwritten.

ERROR The error code of the most recent error.

ERR THROW (Write only) Return to the innermost error catcher.

EXIT (Write only) Does a .break 16.

FDCONVERT With a numeric argument, converts it from an ITS file date to a string of the form ”dd/mm/yy hh:mm:ss” and inserts the string into the buffer. The form n:FS FDCONVERT$ returns the string. With no argument, reads the string from the buffer and converts it to numeric form.

FILE PAD Is the character used to pad the last word of files. Usually 3 (^C).

FLUSHED Is set to non-zero if a –MORE– has been flushed, and thus output is being discarded.

FNAM SYNTAX Controls what happens when only one file name is present. If this is 0, the file name is used as part two. If > 0, the file name is used as part one. If < 0, the file name is used as part one and ">" is used for part two.

GAP LENGTH (Read only) The length of the gap.

GAP LOCATION (Read only) The buffer position of the gap.

HEIGHT (Read only) The number of lines on the screen.

HELP CHAR Contains the character used for the help character. Normally, ^_. If set to -1, help is not recognized (*e.g.*, useful for ^Q).

HELP MAC Is the macro to execute when the help character is typed.

H POSITION (Read only) Returns the column that the point is in.

HSNAME The user's home directory.

I&D CHR When set to non-zero, TECO tries to use the terminal's insert and delete character functions.

I&D LINE When set to non-zero, TECO tries to use the terminal's insert and delete line functions.

IBASE The input radix for numbers not ended by ".". Initially 8 + 2.

I.BASE The input radix for numbers ended by ".". Initially 8.

IF ACCESS (Write only) Sets the access pointer for the input file.

IF CDATE The creation data for the input file.

IF DEVICE (Read only). The device for the input file.

IF DUMP The dumped bit for the input file.

I FILE (Read only) The name of the input file.

IF FN1 (Read only) The first name of the input file.

IF FN2 (Read only) The second name of the input file.

IF LENGTH (Read only) The length of the input file.

IF REAP The reap bit for the input file.

IF SNAME (Read only) The sname of the input file.

IF VERSION (Read only) The version number of the input file or FS IF FN2$.

IMAGE OUT Outputs its argument in super-image mode (no translations at all).

IN COUNT Is an old name for FS TYI COUNT$

INSLEN Is the length of the last string inserted with "I", "G", or "\", or found with "S" or "FW". It will be negative after a backward search.

JNAME (Read only) Returns the jobname.

JRN EXECUTE (Write only) Opens a journal file for playing back. The form :FS JRN EXECUTE$ closes the file. The default file names are used.

JRN IN (Read only) Is non-zero when a journal file is being replayed.

JRN INHIBIT When set to non-zero, input is taken from the terminal even though a journal file is being replayed. This is how FS JRN MACRO$ can work.

JRN INTERVAL Specifies how often a journal file being recorded is updated on disk. The interval is in units of commands.

JRN MACRO This macro is called when a journal file is being replayed and TECO encounters a colon or ˆG in the file. The character is passed as an argument. In the case of a ˆG, the macro should execute a ˆR and then quit by doing -1 FS QUIT$. In the case of a colon, this macro should read more characters from the file by doing FS JRN READ$ and acting upon them.

JRN OPEN (Write only) Opens a journal file for writing (recording). The default file names are used. The form :FS JRN OPEN$ closes the file.

JRN OUT (Read only) Is non-zero when a journal file is being recorded.

JRN READ (Read only) Reads a character from the journal file being replayed. If there is no such file, it returns a random value.

JRN WRITE (Write only) Outputs its argument, either a character or a string, to the journal file being written. If there is no such file, it does nothing.

LAST PAGE (Read only) Set to -1 when a file is opened and set to 0 when the last character has been read.

LINES Is the number of lines used by a standard buffer redisplay. 0 means to use the whole screen.

LISPT When set to non-zero, it means that text is supposed to be passed between TECO and its superior.

LISTEN Returns non-zero if input is available to be read by FI. If it is given an argument and no input is available, the argument is typed out.

MACHINE (Read only) Returns the name of the machine that TECO is running on.

MODE CHANGE When set to non-zero, the FS MODE MACRO$ needs to be run eventually.

MODE MACRO The macro to update Q-register ..J and the mode line.

MODIFIED When set to non-zero, the buffer has been changed since last read or written.

MP DISPLAY (Write only) Outputs text to the main program display.

MSNAME The name of the working directory.

NOOP ALTMODE When set to a negative number, an altmode is considered a no-op. When set to 0, an altmode is considered an error. When set to a positive number, altmode ends execution as ^_ does.

NOQUIT Gives the user control over ^G.

OF ACCESS (Write only) Sets the access pointer for the output file.

OF CDATE The creation data for the output file.

O FILE (Read only) The name of the output file.

OF LENGTH (Read only) The length of the output file.

OF VERSION (Read only) The version number of the output file or FS OF FN2$.

OLD FLUSHED Saves the value of FS FLUSHED$ when that is set to zero upon returning to ^R.

OLD MODE Is the last Q-register ..J actually displayed.

OSPEED The terminal's output speed in baud or 0 if the speed is not known.

OUTPUT When set to non-zero, suppresses output to the EW'd file.

PAGENUM The number of form feeds read from the input file.

PJATTY Set to a negative value whenever TECO detects that the terminal has been taken away. This negative value means that a complete redisplay must be done.

PROMPT The ASCII value of the prompt character.

PUSHPT (Write only) Same as n^V.

QP HOME Returns a string that says where the Q-register PDL (Push Down List == stack) slot n was pushed from. The form :FS QP HOME$ returns a pointer to the Q-register. The form n@FS QP HOME$ converts the pointer returned by :FS QP HOME$ into the string form.

QP PTR The Q-register PDL pointer.

QP SLOT Read the specified PDL slot.

QP UNWIND (Write only) Like FS QP PTR$ but pops slots back into the Q-registers they came from.

QUIT When set to a negative value, execution will quit at the next opportunity.

Q VECTOR Returns an n character long newly-consed up Q-register vector.

RANDOM Reads or sets the random number generator seed.

READ ONLY When set to non-zero. Attempt to modify the buffer become an error.

REAL ADDRESS Returns the value of the machine address of the start of the buffer.

REFRESH When set to non-zero, this macro is executed whenever TECO really clears the whole screen. It is executed after the screen has been cleared.

REREAD When set to non-negative, the 9-bit TV code will be read by the next invocation of FI.

RGETTY 0 if printing terminal, or contains the tctyp word of a display terminal.

RUB CRLF When set to non-zero, both characters of a CR/LF pair are erased together.

RUB MACRO The macro called by ^R mode when it wants to do a ^? or ^D.

RUNTIME (Read only) TECO's runtime in milliseconds.

SAIL When set to non-zero, the terminal is assumed to support the SAIL character set.

S ERROR When set to non-zero, a failing search within an iteration or a ^P sort will generate an error.

SHOW MODE When set to non-zero, FR will type on the mode line on a printing terminal. Has no effect on displays.

S HPOS Is the horizontal position of the point when everything is taken into account, but assuming an infinitely wide line.

S STRING Is the default search string.

STEP MACRO When set to non-zero and numeric, TECO displays the buffer and waits at the start of every line in a program. When set to non-zero and a string, TECO executes this macro at the beginning of every line in a program. Macros that start with W are never stepped.

STEP DEPTH When set to -1, stepping occurs always. Otherwise, it is the number of the stack level at which to cut off stepping.

SUPERIOR Is the macro invoked when superiors want to put text into TECO.

S VALUE Is the value returned by last search command.

TOP LINE The number of the first line of the screen that TECO should use.

TRACE When set to non-zero, TECO is in trace mode. See ?.

TRUNCATE If negative, long lines should be truncated. If 0 or positive, long lines are wrapped to the next line.

TTMODE When set to non-zero, tells TECO that normal buffer display should display on printing terminals.

TTY INIT (Re)initializes TECO's TTY information.

TTY MACRO Performs user-specified TTY initialization.

TTYOPT (Read only) The TTYOPT word for the terminal. Use the %TOxxx values instead.

TTYSMT (Read only) The TTYSMT word for the terminal.

TYI BEG The value of FS TYI COUNT$ the last time through the main ^R command loop.

TYI COUNT The number of characters read so far.

TYI SINK When set to non-zero, is a macro that is executed every time a character is actually read from the terminal.

TYI SOURCE When set to non-zero, it a macro that is called to obtain "terminal input."

TYO HASH Returns the hash code of screen line n. Doing -1,n FS TYO HASH$ forces line n to be redisplayed.

TYO HPOS (Read only) Holds the horizontal position at which type out will next appear.

TYO VPOS (Read only) Holds the vertical position at which type out will next appear.

TYPEOUT Tells where type out will next appear. If -1, the next type out will appear at the top of the screen. Otherwise, type out will appear just after the last type out.

U HSNAME Determines a user's hsname.

UINDEX (Read only) The user index of the TECO job.

U MAIL FILE The complete file name of the user's mail file.

UNAME (Read only) The user name of the TECO job.

UPTIME (Read only) Returns the time that the system has been up in units of 1/30 second.

UREAD (Read only) Is -1 if an input file is open, otherwise it is zero.

UWRITE (Read only) Is -1 if an output file is open, otherwise it is zero.

VAR MACRO When set to non-zero, a macro can be run whenever a variable is set.

V B Is the distance between the real beginning of the buffer and the virtual beginning.

VERBOSE When set to non-zero, TECO prints long error messages. Otherwise, TECO prints only short messages and ^X must be typed to see the long version.

VERSION (Read only) The TECO version number.

V Z Is the distance between the real end of the buffer and the virtual end.

WIDTH Width of the terminal in characters.

WINDOW The position of the first character in the display window, relative to the virtual beginning of the buffer.

WORD Gets or sets words in the current buffer.

XJNAME (Read only) Returns the xjname of the TECO job.

X MODIFIED Just like FS MODIFIED$, only it doesn't affect the display of the modified flag in the mode line. Thus, the user can track whether changes were made by intervening commands.

X PROMPT Printed and zeroed with each printing terminal prompt.

XUNAME (Read only) Returns the xjname of the TECO job.

Y DISABLE When set to 0, the Y command is legal. When set to 1, the Y command is always illegal. When set to -1, the Y command is treated as @Y.

Z (Read only) The number of characters in the buffer.

^H PRINT When set to negative, a ^H on output will backspace and overprint. Otherwise, ^H will type as a ^ and H.

^I DISABLE When set to 0, ^I is an insert command. When set to 1, ^I is illegal. When set to -1, ^I is a no-op.

^L INSERT When set to non-zero, form feeds read from files always go into the buffer and P and PW never output anything except what is in the buffer.

^M PRINT When set to negative, a ^M on output will output as a CR/LF. Otherwise, ^M will type as a ^ and M.

^P CASE When set to non-zero, ^P ignores case.

^R ARG Is the explicit numeric argument and is 0 (not 1!) if no argument was entered

^R ARGP Describes the ^R command's argument.

bit 2^0 set if any argument was specified

bit 2^1 set if a number was typed

bit 2^2 set if the argument is negative

^R CCOL The comment column.

^R CMACRO Converts the ASCII value n to a form required for ^R command dispatch.

^R DISPLAY When set to non-zero, this macro is executed whenever ^R is about to do a non-trivial redisplay.

^R EXIT (Write only) Exits from the innermost ^R invocation.

^R ECHO When set to 1, the characters read in by ^R should not be echoed. When set to 0, they should be echoed only on printing terminals. When set to -1, they should be echoed on all terminals.

^R EC SD When set to non-zero, this macro is executed whenever a space command is typed. Used for auto-filling and such.

^R ENTERED When set to non-zero, this macro is executed whenever ^R is invoked.

^R EXPT Is the ^U count for the next ^R mode command.

^R H MIN (Read only) Is the leftmost horizontal position requiring redisplay.

^R HPOS The current horizontal position of the cursor.

^R INDIRECT Given a 9-bit character, follows the alias definitions to find what it is equivalent to.

^R INHIBIT When set to non-zero, ^R will not update the display.

When set back to zero, ^R will catch up. ^R INIT Returns the initial definition of the character whose ASCII value is n.

^R INSERT Inserts its argument.

^R LAST Holds the most recent character read by any ^R.

^R LEAVE When set to non-zero, this macro is executed whenever ^R returns.

^R MARK Records the position of the mark.

^R MAX The maximum number of characters of insertions or deletions printed out by ^R on a printing terminal before it switches to printing a description of the change. Default is 50.

^R MCNT The secretary mode counter.

^R MDLY The secretary mode limit value.

^R MODE (Read only) Non-zero while in ^R mode.

^R MORE When positive, –MORE– is used for the ^R mode line instead of –TOP–, –BOT–, and –nn%–. This is used when in an environment where Space means "show me the next screenful." When negative, no –XXX– is displayed.

^R NORMAL When set to non-zero, this macro is executed for all normally self-insert characters.

^R PAREN When set to non-zero, this macro is executed for every ")" character.

^R PREVIOUS Holds the previous (second most recent) command.

^R REPLACE When set to non-zero, ^R runs in "replace" mode instead of "insert" mode.

^R RUBOUT The internal ^R rubout routine.

^R SCAN When set to non-zero and a printing terminal is in use, displays characters that are being moved past.

^R STAR When set to non-zero, a star appears in the mode line if the buffer has been modified.

^R SUPPRESS When set to 0 or positive, built-in ^R mode commands are suppressed and characters insert.

^R THROW Returns control to the innermost invocation of ^R.

^R UNSUPP When set to -1, one character will be unsuppressed.

^R V MIN (Read only) Is the topmost line requiring redisplay.

^R VPOS The current vertical position of the cursor.

_ DISABLE When 0, _ is "search and yank." When 1, _ is illegal. When -1, _ is treated like -.

Appendix E: ASCII Chart

This character set is as specified in ANS standards, and is known as the ASCII character set. It has been extended to include Meta characters (characters with their top, or eighth, bit turned on).

Decimal	Octal	Hex	Graphic	Name (Meaning)
0.	000	00	^@	NUL (used for padding)
1.	001	01	^A	SOH (start of header)
2.	002	02	^B	STX (start of text)
3.	003	03	^C	ETX (end of text)
4.	004	04	^D	EOT (end of transmission)
5.	005	05	^E	ENQ (enquiry)
6.	006	06	^F	ACK (acknowledge)
7.	007	07	^G	BEL (bell or alarm)
8.	010	08	^H	BS (backspace)
9.	011	09	^I	HT, TAB (horizontal tab)
10.	012	0A	^J	LF (line feed)
11.	013	0B	^K	VT (vertical tab)
12.	014	0C	^L	FF (form feed, new page)
13.	015	0D	^M	CR (carriage return)
14.	016	0E	^N	SO (shift out)
15.	017	0F	^O	SI (shift in)
16.	020	10	^P	DLE (data link escape)
17.	021	11	^Q	DC1, XON (device control 1)
18.	022	12	^R	DC2 (device control 2)
19.	023	13	^S	DC3, XOFF (device control 3)
20.	024	14	^T	DC4 (device control 4)
21.	025	15	^U	NAK (negative acknowledge)
22.	026	16	^V	SYN (synchronous idle)
23.	027	17	^W	ETB (end transmission block)
24.	030	18	^X	CAN (cancel)

25.	031	19	^Y	EM (end of medium)
26.	032	1A	^Z	SUB (substitute)
27.	033	1B	^[	ESC (escape, alter mode, SEL)
28.	034	1C	^\	FS (file separator)
29.	035	1D	^]	GS (group separator)
30.	036	1E	^^	RS (record separator)
31.	037	1F	^_	US (unit separator)

32.	040	20		space or blank
33.	041	21	!	exclamation mark
34.	042	22	"	double quote
35.	043	23	#	number sign (hash mark)
36.	044	24	$	dollar sign
37.	045	25	%	percent sign
38.	046	26	&	ampersand sign
39.	047	27	'	single quote (apostrophe)
40.	050	28	(	left parenthesis
41.	051	29	)	right parenthesis
42.	052	2A	*	asterisk (star)
43.	053	2B	+	plus sign
44.	054	2C	,	comma
45.	055	2D	-	minus sign (dash)
46.	056	2E	.	period (decimal point, dot)
47.	057	2F	/	(right) slash

48.	060	30	0	numeral zero
49.	061	31	1	numeral one
50.	062	32	2	numeral two
51.	063	33	3	numeral three
52.	064	34	4	numeral four
53.	065	35	5	numeral five
54.	066	36	6	numeral six
55.	067	37	7	numeral seven
56.	070	38	8	numeral eight
57.	071	39	9	numeral nine
58.	072	3A	:	colon
59.	073	3B	;	semi-colon
60.	074	3C	<	less-than sign
61.	075	3D	=	equal sign
62.	076	3E	>	greater-than sign
63.	077	3F	?	question mark

64.	100	40	@	atsign
65.	101	41	A	upper-case letter ALPHA
66.	102	42	B	upper-case letter BRAVO
67.	103	43	C	upper-case letter CHARLIE
68.	104	44	D	upper-case letter DELTA
69.	105	45	E	upper-case letter ECHO
70.	106	46	F	upper-case letter FOXTROT
71.	107	47	G	upper-case letter GOLF
72.	110	48	H	upper-case letter HOTEL
73.	111	49	I	upper-case letter INDIA
74.	112	4A	J	upper-case letter JERICHO
75.	113	4B	K	upper-case letter KAPPA
76.	114	4C	L	upper-case letter LIMA
77.	115	4D	M	upper-case letter MIKE
78.	116	4E	N	upper-case letter NOVEMBER
79.	117	4F	O	upper-case letter OSCAR
80.	120	50	P	upper-case letter PAPPA
81.	121	51	Q	upper-case letter QUEBEC
82.	122	52	R	upper-case letter ROMEO
83.	123	53	S	upper-case letter SIERRA
84.	124	54	T	upper-case letter TANGO
85.	125	55	U	upper-case letter UNICORN
86.	126	56	V	upper-case letter VICTOR
87.	127	57	W	upper-case letter WHISKEY
88.	130	58	X	upper-case letter XRAY
89.	131	59	Y	upper-case letter YANKEE
90.	132	5A	Z	upper-case letter ZEBRA
91.	133	5B	[	left square bracket
92.	134	5C	\	left slash (backslash)
93.	135	5D	]	right square bracket
94.	136	5E	^	uparrow (caret)
95.	137	5F	_	underscore
96.	140	60	`	(single) back quote (grave accent)
97.	141	61	a	lower-case letter alpha
98.	142	62	b	lower-case letter bravo
99.	143	63	c	lower-case letter charlie
100.	144	64	d	lower-case letter delta
101.	145	65	e	lower-case letter echo

102.	146	66	f	lower-case letter foxtrot
103.	147	67	g	lower-case letter golf
104.	150	68	h	lower-case letter hotel
105.	151	69	i	lower-case letter india
106.	152	6A	j	lower-case letter jericho
107.	153	6B	k	lower-case letter kappa
108.	154	6C	l	lower-case letter lima
109.	155	6D	m	lower-case letter mike
110.	156	6E	n	lower-case letter november
111.	157	6F	o	lower-case letter oscar
112.	160	70	p	lower-case letter pappa
113.	161	71	q	lower-case letter quebec
114.	162	72	r	lower-case letter romeo
115.	163	73	s	lower-case letter sierra
116.	164	74	t	lower-case letter tango
117.	165	75	u	lower-case letter unicorn
118.	166	76	v	lower-case letter victor
119.	167	77	w	lower-case letter whiskey
120.	170	78	x	lower-case letter xray
121.	171	79	y	lower-case letter yankee
122.	172	7A	z	lower-case letter zebra
123.	173	7B	{	left curly brace
124.	174	7C	|	vertical bar
125.	175	7D	}	right curly brace
126.	176	7E	~	tilde
127.	177	7F	^?	DEL (delete, rub out)
128.	200	80	~^@	Meta NUL (used for padding)
129.	201	81	~^A	Meta SOH (start of header)
130.	202	82	~^B	Meta STX (start of text)
131.	203	83	~^C	Meta ETX (end of text)
132.	204	84	~^D	Meta EOT (end of transmission)
133.	205	85	~^E	Meta ENQ (enquiry)
134.	206	86	~^F	Meta ACK (ackowledge)
135.	207	87	~^G	Meta BEL (bell or alarm)
136.	210	88	~^H	Meta BS (backspace)
137.	211	89	~^I	Meta HT, TAB (horizontal tab)
138.	212	8A	~^J	Meta LF (line feed)
139.	213	8B	~^K	Meta VT (vertical tab)
140.	214	8C	~^L	Meta FF (form feed, new page)

141.	215	8D	~^M	Meta CR (carriage return)
142.	216	8E	~^N	Meta SO (shift out)
143.	217	8F	~^O	Meta SI (shift in)

144.	220	90	~^P	Meta DLE (data link escape)
145.	221	91	~^Q	Meta DC1, XON (device control 1)
146.	222	92	~^R	Meta DC2 (device control 2)
147.	223	93	~^S	Meta DC3, XOFF (device control 3)
148.	224	94	~^T	Meta DC4 (device control 4)
149.	225	95	~^U	Meta NAK (negative acknowledge)
150.	226	96	~^V	Meta SYN (synchronous idle)
151.	227	97	~^W	Meta ETB (end transmission block)
152.	230	98	~^X	Meta CAN (cancel)
153.	231	99	~^Y	Meta EM (end of medium)
154.	232	9A	~^Z	Meta SUB (substitute)
155.	233	9B	~^[	Meta ESC (escape, alter mode, SEL)
156.	234	9C	~^\	Meta FS (file separator)
157.	235	9D	~^]	Meta GS (group separator)
158.	236	9E	~^^	Meta RS (record separator)
159.	237	9F	~^_	Meta US (unit separator)

160.	240	A0	~	Meta space
161.	241	A1	~!	Meta exclamation mark
162.	242	A2	~"	Meta double quote
163.	243	A3	~#	Meta number sign (hash mark)
164.	244	A4	~$	Meta dollar sign
165.	245	A5	~%	Meta percent sign
166.	246	A6	~&	Meta ampersand sign
167.	247	A7	~'	Meta single quote (apostrophe)
168.	250	A8	~(	Meta left parenthesis
169.	251	A9	~)	Meta right parenthesis
170.	252	AA	~*	Meta asterisk (star)
171.	253	AB	~+	Meta plus sign
172.	254	AC	~,	Meta comma
173.	255	AD	~-	Meta minus sign (dash)
174.	256	AE	~.	Meta period (decimal point, dot)
175.	257	AF	~/	Meta (right) slash

176.	260	B0	~0	Meta numeral zero
177.	261	B1	~1	Meta numeral one
178.	262	B2	~2	Meta numeral two

179.	263	B3	~3	Meta numeral three
180.	264	B4	~4	Meta numeral four
181.	265	B5	~5	Meta numeral five
182.	266	B6	~6	Meta numeral six
183.	267	B7	~7	Meta numeral seven
184.	270	B8	~8	Meta numeral eight
185.	271	B9	~9	Meta numeral nine
186.	272	BA	~:	Meta colon
187.	273	BB	~;	Meta semi-colon
188.	274	BC	~<	Meta less-than sign
189.	275	BD	~=	Meta equal sign
190.	276	BE	~>	Meta greater-than sign
191.	277	BF	~?	Meta question mark
192.	300	C0	~@	Meta atsign
193.	301	C1	~A	Meta upper-case letter ALPHA
194.	302	C2	~B	Meta upper-case letter BRAVO
195.	303	C3	~C	Meta upper-case letter CHARLIE
196.	304	C4	~D	Meta upper-case letter DELTA
197.	305	C5	~E	Meta upper-case letter ECHO
198.	306	C6	~F	Meta upper-case letter FOXTROT
199.	307	C7	~G	Meta upper-case letter GOLF
200.	310	C8	~H	Meta upper-case letter HOTEL
201.	311	C9	~I	Meta upper-case letter INDIA
202.	312	CA	~J	Meta upper-case letter JEHRICHO
203.	313	CB	~K	Meta upper-case letter KAPPA
204.	314	CC	~L	Meta upper-case letter LIMA
205.	315	CD	~M	Meta upper-case letter MIKE
206.	316	CE	~N	Meta upper-case letter NOVEMBER
207.	317	CF	~O	Meta upper-case letter OSCAR
208.	320	D0	~P	Meta upper-case letter PAPPA
209.	321	D1	~Q	Meta upper-case letter QUEBEC
210.	322	D2	~R	Meta upper-case letter ROMEO
211.	323	D3	~S	Meta upper-case letter SIERRA
212.	324	D4	~T	Meta upper-case letter TANGO
213.	325	D5	~U	Meta upper-case letter UNICORN
214.	326	D6	~V	Meta upper-case letter VICTOR
215.	327	D7	~W	Meta upper-case letter WHISKEY
216.	330	D8	~X	Meta upper-case letter XRAY
217.	331	D9	~Y	Meta upper-case letter YANKEE

218.	332	DA	~Z	Meta upper-case letter ZEBRA
219.	333	DB	~[	Meta left square bracket
220.	334	DC	~\	Meta left slash (backslash)
221.	335	DD	~]	Meta right square bracket
222.	336	DE	~^	Meta uparrow (caret)
223.	337	DF	~_	Meta underscore
224.	340	E0	~`	Meta (single) back quote (grave acent)
225.	341	E1	~a	Meta lower-case letter alpha
226.	342	E2	~b	Meta lower-case letter bravo
227.	343	E3	~c	Meta lower-case letter charlie
228.	344	E4	~d	Meta lower-case letter delta
229.	345	E5	~e	Meta lower-case letter echo
230.	346	E6	~f	Meta lower-case letter foxtrot
231.	347	E7	~g	Meta lower-case letter golf
232.	350	E8	~h	Meta lower-case letter hotel
233.	351	E9	~i	Meta lower-case letter india
234.	352	EA	~j	Meta lower-case letter jericho
235.	353	EB	~k	Meta lower-case letter kappa
236.	354	EC	~l	Meta lower-case letter lima
237.	355	ED	~m	Meta lower-case letter mike
238.	356	EE	~n	Meta lower-case letter november
239.	357	EF	~o	Meta lower-case letter oscar
240.	360	F0	~p	Meta lower-case letter pappa
241.	361	F1	~q	Meta lower-case letter quebec
242.	362	F2	~r	Meta lower-case letter romeo
243.	363	F3	~s	Meta lower-case letter sierra
244.	364	F4	~t	Meta lower-case letter tango
245.	365	F5	~u	Meta lower-case letter unicorn
246.	366	F6	~v	Meta lower-case letter victor
247.	367	F7	~w	Meta lower-case letter whiskey
248.	370	F8	~x	Meta lower-case letter xray
249.	371	F9	~y	Meta lower-case letter yankee
250.	372	FA	~z	Meta lower-case letter zebra
251.	373	FB	~{	Meta left curly brace
252.	374	FC	~|	Meta vertical bar
253.	375	FD	~}	Meta right curly brace
254.	376	FE	~~	Meta tilde
255.	377	FF	~^?	Meta DEL (delete, rub out)

These forms can be used to prevent ambiguity:

```
 94.     136   5E  ^           can be printed as ^=
126.     176   7E  ~           can be printed as ^~
222.     336   DE  ~^          can be printed as ~^=
254.     376   FE  ~~          can be printed as ~^~
```

Bibliography

This bibliography is in two parts. The first part is the list of publications used in the preparation of this book. The second part is the annotated bibliography from the thesis. Documents that are marked with "*" are especially valuable or interesting.

Current

American National Standards Institute (1990) X3J11 Programming Language C. New York: ANSI.

American National Standards Institute (1983) ANSI/MIL-STD-1815A-1983 Reference Manual for the Ada Programming Language. New York: Springer-Verlag. ISBN 0-387-90887-0.

*Apple Computer Corp. (1987) Human Interface Guidelines: the Apple Desktop Interface. Reading, Massachusetts: Addison-Wesley. ISBN 0-201-17753-6.

ibid. (1989) Release 7.0 Macintosh Script Management System (unreleased preliminary). Cuptertino, California: Apple Computer Corp.

*Brooks, Frederick P. (1982) The Mythical Man-Month. Reading, Massachusetts: Addison-Wesley. ISBN 0-201-00650-2.

*Caplan, Ralph (1982) By Design: Why There Are No Locks on the Bathroom Doors in the Hotel Louis XIV and Other Object Lessons. New York: McGraw-Hill. ISBN 0-07-009777-1.

Carroll, Lewis (1865) Alice's Adventures in Wonderland. London: Macmillian and Co.

ibid. (1871) Through the Looking-Glass and What Alice Found There. London: Macmillian and Co.

Crowley, Terrence; Forsdick, Harry; Landau, Matt; Travers, Virginia (1987) The Diamond Multimedia Editor. USENIX Proceedings, Summer 1987.

Finseth, Craig A. (June 1980) Theory and Practise of Text Editing – or – A Cookbook for an Emacs. Cambridge, Massachusetts: M.I.T. Laboratory for Computer Science. Technical Memo TM-165.

Hammer, Michael; Ilson, Richard; Anderson, Timothy; Gilbert, Edward J.; Good, Michael; Niamir, Bahram; Rosenstein, Larry; Schoichet, Sandor (February 1981) Etude:

An Integrated Document Processing System. Cambridge, Massachusetts: M.I.T. Laboratory for Computer Science. Office Automation Group Memo OAM-028.

Ilson, Richard; Good, Michael (March 1981) Etude: An Interactive Editor and Formatter. Cambridge, Massachusetts: M.I.T. Laboratory for Computer Science. Office Automation Group Memo OAM-029.

Jensen, Kathleen & Wirth, Nikalus (1974) Pascal User Manual and Report. New York: Springer-Verlag. ISBN 0-387-90144-2.

Kemeny, John G. & Kurtz, Thomas E. (1985) Back to Basic. Reading, Massachusetts: Addison-Wesley. ISBN 0-201-13433-0.

Kernighan, Brian W. & Ritchie, Dennis M. (1978) The C Programming Language. Englewood Cliffs, New Jersey: Prentice-Hall. ISBN 0-13-110163-3.

*Knuth, Donald E. (1971) An Empirical Study of Fortran Programs. Software Practise and Experience, vol 1, April/May, p 105-133.

Miller, Webb (1987) A Software Tools Sampler. Englewood Cliffs, New Jersey: Prentice-Hall. ISBN 0-13-822305-X.

Myers, Eugene W. (December 1986) A Simple Row-Replacement Method. Tucson, Arizona: Department of Computer Science, University of Arizona. Technical Report TR 86-28.

*Norman, Donald A. (1990) The Design of Everyday Things. New York: Doubleday. ISBN 0-385-26774-6.

Oman, Paul W. & Cook, Curtis R. (1990) Typographic Style is More than Cosmetic. Communications of the ACM, vol. 33 #5, January, p 506.

Phelps, Hermann (1982) The Craft of Log Building. (Translation of Holzbaukunst : der Blockbau.) Ottawa, Ontario: Lee Valley Tools. ISBN 0-9691019-2-9 (bound), 0-9691019-1-0 (pbk).

Qiao, Jinan; Qiao, Yizheng; Qiao, Sanzheng. (1990) Six-Digit Coding Method. Communications of the ACM, vol. 33 #5, January, p 491.

Quarterman, John S. (1989) The Matrix: Computer Networks and Conferencing Systems Worldwide. Bedford, Massachusetts: Digital Press. ISBN 1-55558-033-5.

Reid, Brian K. & Walker, Janet H. (1980) Scribe Introductory User's Manual. Pittsburgh, Pennsylvania: Unilogic Ltd.

Stallman, Richard (1987) GNU Emacs Manual. Cambridge, Massachusetts: Free Software Foundation. Sixth edition, version 18.

Tayli, Murat & Al-Salamah, Abdulla I. (1990) Building Bilingual Microcomputer Systems. Communications of the ACM, vol. 33 #5, January, p 495.

Thorell, L.G. & Smith, W.J. (1990) Using Computer Color Effectively, An Illustrated Reference. Englewood Cliffs, New Jersey: Prentice-Hall. ISBN 0-13-939878-3.

*Tufte, Edward R. (1990) Envisioning Information. Cheshire, Connecticut: Graphics Press.

ibid. (1983) The Visual Display of Quantitative Information. Cheshire, Connecticut: Graphics Press.

The USENET News groups *Comp.editors*, *Comp.emacs*, and *Gnu.emacs* carry editor-related material.

Thesis

This bibliography includes many different types of documents. Some of the documents are user manuals for various editors. Others of them describe the implementation of specific editors. Still others discuss language tradeoffs or input/output system interfaces.

They are grouped by the type of editor that they refer to. Each entry is annotated to help place it in perspective.

Emacs-Type Editors

There are four principal implementations of Emacs-type editors, and there are enough documents to justify their separate listing.

ITS EMACS

Ciccarelli, Eugene (1978) An Introduction to the Emacs Editor. Cambridge, Massachusetts: MIT Artificial Intelligence Laboratory, MIT AI Lab Memo #447, January 1978. – A primer on the editor's user interface.

*Stallman, Richard M. (1979) Emacs: The Extensible, Customizable, Self-Documenting, Display Editor. Cambridge, Massachusetts: MIT Artificial Intelligence Laboratory, AI Memo #519, June 1979. – Provides arguments for the Emacs philosophy.

ibid. (1978) Structured Editing with a Lisp. Computing Surveys, vol 10 #4, December, p 505. – This is a response to the Sanderwall paper (referenced later).

On-line Documentation:

MIT-AI: .TECO.; TECORD > – A more detailed command list for TECO
MIT-AI: .TECO.; TECO PRIMER – A primer for TECO
MIT-AI: EMACS; EMACS CHART – A four-page command list for Emacs
MIT-AI: EMACS; EMACS GUIDE – A detailed user interface manual
MIT-AI: EMACS; EMACS ORDER – A more detailed command list for Emacs

Lisp Machine Zwei

*Weinreb, Daniel L. & Moon, David (January 1979) The Lisp Machine Manual.

Cambridge, Massachusetts: MIT Artificial Intelligence Laboratory. – The user interface for Zwei.

ibid. (January 1979) A Real-Time Display-Oriented Editor for the Lisp Machine. Cambridge, Massachusetts: S.B. Thesis, MIT Electrical Engineering and Computer Science Department. – How Zwei works internally.

Multics Emacs

Greenberg, Bernard S. (in publication in 1980) Emacs Extension Writer's Guide. Honeywell Information Systems, Inc., order #CJ52. – How to write extensions.

ibid. (December 1979) Emacs Text Editor User's Guide. Honeywell Information Systems, Inc., order #CH27. – The user interface.

**ibid.* (March 1980) Multics Emacs: An Experiment in Computer Interaction. Honeywell Information Systems, Proceedings, Fourth Annual Honeywell Software Conference. – A summary of MEPAP (referenced below, also, MIT-AI: BSG; NMEPAP >)

ibid. (April 1978) Real-Time Editing on Multics. Cambridge, Massachusetts: Honeywell Information Systems, Inc., Multics Technical Bulletin #373

ibid., On-Line Documentation:

MIT-AI: BSG; LMEPAP > – Why Lisp was chosen for the implementation language

MIT-AI: BSG; MEPAP > – A detailed history of Emacs in general and the Multics implementation in specific. Very valuable.

MIT-AI: BSG; R4V > – A proposal for a terminal independent video terminal support package.

MIT-AI: BSG; TTYWIN > – A look at the good and bad features of video terminals.

MagicSix TVMacs

*Anderson, Owen Ted (January 1979) The Design and Implementation of a Display-Oriented Editor Writing System. Cambridge, Massachusetts: S.B. Thesis, MIT Physics Department. – How TVMacs works internally. It concentrates on describing not the editor itself but rather the implementations language: SINE.

Linhart, Jason T. (June 1980) Dynamic Multi-Window Terminal Management for the MagicSix Operating System. Cambridge, Massachusetts: S.B. Thesis, MIT Electrical Engineering and Computer Science Department. – A video terminal management system. Contains many useful comments on terminal independence and redisplay problems.

Other Emacs-Type Text Editors

This section covers editors which have the same general user interface as an Emacs (*e.g.*, screen-oriented, similar key bindings) but are not extensible or otherwise fall noticeably

short of the Emacs philosophy.

Finseth, Craig A. (August 1979) VINE Primer. Dallas, Texas: Texas Instruments, Inc., Central Research Laboratories, Systems and Information Sciences Laboratory. – User interface manual for the complete novice.

Schiller, Jeffrey I. (June 1979) TORES: the Text ORiented Editing System Cambridge, Massachusetts: revised from S.B. Thesis, MIT Electrical Engineering and Computer Science Department.

On-Line Documentation:

Kazar, Mike. User manual for FINE, running at Carnegie-Mellon University.
At CMU-10A: fine.{mss prt}[s200mk50]

Non-Emacs Display Editors

Bilofsky, Walter (December 1977) The CRT Text Editor NED – Introduction and Reference Manual. Rand Corporation, R-2176-ARPA.

Irons, E. T. & Djorup, F. M. (1972) A CRT Editing System. Communications of the ACM, vol. 15 #1, January, p 16.

Joy, William (April 1979) Ex Reference Manual; Version 2.0. Berkeley, California; Computer Science Division, Dept of Electrical Engineering and Computer Science, University of California at Berkeley.

ibid. (April 1979) An Introduction to Display Editing With vi. Berkeley, California: Computer Science Division, Dept of Electrical Engineering and Computer Science, University of California at Berkeley.

Kanerva, Pentti (1973) TVGUID: a User's Guide to TEC/DATAMEDIA TV-Edit. Palo Alto, California: Stanford University, Institute for Mathematical Studies in the Social Sciences.

Kelly, Jeanne (July 1977) A Guide to NED: a New On-Line Computer Editor. The Rand Corporation, R-2000-ARPA.

Kernighan, Brian W. (1978) A Tutorial Introduction to the ED Text Editor. Murray Hill, New Jersey: Bell Laboratories Technical Report.

MacLeod, I. A. (November 1977) Design and Implementation of a Display-Oriented Text Editor. Software Practice and Experience, vol. 7 #6, November, p 771.

Weiner, P., et. al. (April 1973) The Yale Editor "E": a CRT-Based Editing System. Yale Computer Science Research Report 19

Seybold, Patricia B. (October 1978) TYMSHARE's AUGMENT – Heralding a New Era. The Seybold Report on Word Processing, vol. 1 #9. ISSN: 0160-9572, Seybold Publications, Inc., Box 644, Media, Pennsylvania 19063

On-Line Documentation:

SAIL: E.ALS[UP,DOC] – User manual again. Stanford University.

Structure Editors

Ackland, Gillian M., et al (?) UCSD Pascal Version 1.5 (Reference Manual). San Diego, California: Institute for Information Systems, University of California at San Diego.

Donzeau-Gouge, V.; Huet, G.; Kahn, G.; Lang, B.; & Levy, J.J. (April 1975) A Structure Oriented Program Editor: a First Step Towards Computer Assisted Programming. Paris: IRIA, Res. Rep. 114.

Teitelbaum, R. T. (?) The Cornell Program Synthesizer: a Microcomputer Implementation of PL/CS. Ithaca, New York: Department of Computer Science, Cornell University, Technical Report TR 79-370,.

Other Editors

Benjamin, Arthur J. (August 1972) An Extensible Editor for a Small Machine With Disk Storage. Communications of the ACM, vol. 15 #8 p 742. – Talks about an editor for the IBM 1130 written in Fortran. Not extensible at all.

Bourne, S. R. (January 1971) A Design for a Text Editor. Software Practice and Experience, vol 1 p 73. – User manual.

Cecil, Moll & Rinde (March 1977) TRIX AC: a Set of General Purpose Text Editing Commands. Lawrence Livermore Laboratory UCID 30040.

Deutsch, L. Peter & Lampson, Butler W. (1967) An On-line Editor. Communications of the ACM, vol 10 #12, December, p 793. – QED user manual.

Fraser, Christopher W. (1970) A Compact, Portable CRT-Based Editor. Software Practice and Experience, vol. 9 #2, February, p 121. – Front end to a line editor.

ibid. (1980) A Generalized Text Editor. Communications of the ACM, vol. 23 #3, March, p 154. – Applying text editors to non-text objects,

Hansen, W. J. (June 1971) Creation of Hierarchic Text With a Computer Display. Palo Alto, California: Ph.D. Thesis, Stanford University.

Kai, Joyce Moore (July 1974) A Text Editor Design. Urbana, Illinois: Department of Computer Science, University of Illinois at Urbana-Champaign. – Describes both internals and externals on the editor. However, the design is a poor one.

Kernighan, Brian W. & Plauger, P. J. (1976) Software Tools. Reading, Massachusetts: Addison-Wesley. – This book has a chapter which leads you by the hand in implementing a simple line editor in RatFor.

*Roberts, Teresa L. (November 1979) Evaluation of Computer Text Editors. Systems Sciences Laboratory, Xerox PARC. – A comparative evaluation of four text editors. Quite well done. Unfortunately, it does not include Emacs (it uses DEC TECO instead).

Sanderwall, Erik (1978) Programming in the Interactive Environment: the Lisp Experience. Computing Surveys, vol. 10 #1, March, p 35. – Talks about the editor for InterLisp.

Sneeringer, James (1978) User-Interface Design for Text Editing: a Case Study. Software Practice and Experience, vol 8, p 543. – User manual and a discussion of user interface concepts.

Teitelman, Warren (October 1978) InterLisp Reference Manual. Palo Alto, California: Xerox Palo Alto Research Center. – How to use the InterLisp (non-display) structure editor.

van Dam, Andries & Rice, David E. (1971) On-line Text Editing: a Survey. Computing Surveys, vol #3, September, p 93. – Contains a general introduction to the problems of text editing. Out-dated technology, however.

Book Index

This is the index to the book and the numbers, of course, reflect the page numbers in the book. How quaint.