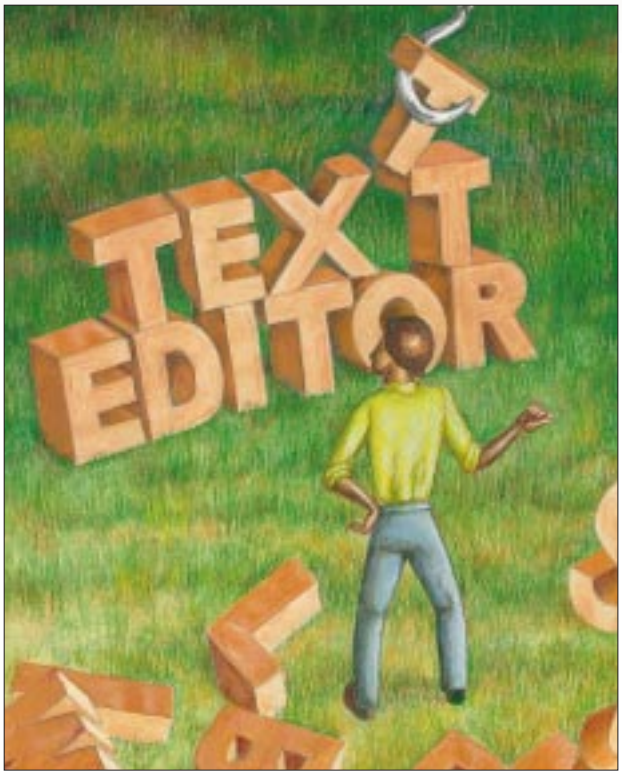
by Peter Collinson, Hillside Systems



JANE MARINSKY

Editing

In these days of ubiquitous word processors people expect to have an easy-to-use text editor on their system. Text editors should be visual with WYSIWYG facilities showing you the state of the file you are creating or changing. Commonly, as new text is typed in from the keyboard it replaces existing text or is inserted where the text cursor points. Navigation around the file should not involve any hidden keys. For example, to move upward, you should be able to use the "up" arrow key on your keyboard. Better still, the mouse should be able to move you up and down the file, moving the text cursor in one click.

If you work exclusively in the visual environment on your Sun Microsystems Inc. machine, then you'll find that many applications provide you with an easy-to-use editing environment by default. For example, when dealing with email, you are presented with a window where you can type a caustic reply to the person who just flamed you. If you want to edit a text file, then Sun has long provided the `textedit` program, which loads the text to be edited into a window and allows you to poke at it with the keyboard and mouse.

However, these window-based editors are generally very simplistic. They are good for originating text files or making simple edits on short files, but they are not very good at allowing complex changes to a file. If a complex alteration is needed, then many users will sit there hacking away, making one change at a time. It's inertia, people use what they know. I regard this type of activity as "the computer controlling the user." There *are* faster, less labor-intensive, and more accurate ways.

Types of Editors

There are a great many flavors of "standard" text editor for UNIX. If asked, many people will know there is a UNIX visual text editor, called `vi`, and will then go on to say "I used it once. It was impossible." I am quite sympathetic to this view.

Most text editors work in exactly the same way. You tell it that you wish to edit a file, the program reads the file into a temporary buffer and waits for keyboard commands from you to change its copy of the file. When you are done editing, it will write the new file out.

The major UNIX text editors predated the introduction of visual environments, using pointing devices like a mouse and bitmapped screens. They actually predated the widespread introduction of enhanced keyboards containing positioning keys. These editors had to use basic keyboard input for the text to be stored and for the commands that instruct the editor in the necessary navigation around the file, or for taking the many other actions that we expect editors to undertake. The programmer designing the editor was presented with a problem: When the user types a character, should it be inserted into the file or should it be interpreted as an editor command?

Designers adopted two ways around

the problem. The first idea is to have *modes*, where input to the program is interpreted in the context of the state that the program is currently "in." So `vi` will start up in command mode, interpreting keystrokes as commands to the editor. Some of these commands will switch the mode into character input mode, where input to the program is added into the internal buffer, changing the file. A particular keystroke, Escape for `vi`, takes the user out of input mode and back into command mode.

Other editors, notably `emacs` and its clones, are *modeless*. These editors have a single mode, where text that is input will be sent to the buffer for insertion into the file. These editors require some way of providing for navigation around the file and for command input. They cannot use the normal character set because it's assumed that typing a visible character will make it appear in the file. They make use of the control key values, a set of keystrokes that are part of the character set but are not used in normal text input.

Key Input

Keyboards have always been able to deliver different codes by using key-chords, so, for example, holding down the Shift key and pressing another key at the same time transmits an uppercase letter. When UNIX was driven from terminals, the computer was sent a code value for each key that was pressed. A different value was sent when the user held down Shift and pressed a key. Actually, the numbers that are sent for the "shifted" and "unshifted" keys are related, pressing the Shift key and typing another key forces a zero in a bit position in the character code that is sent for that key.

Computer keyboards typically provide further modifier keys, allowing other bits in the character code coming from a specific key to be altered. The Control key forces another bit in the character code to be zero and generates a code that is in the lowest 32 positions in the character code set, a section that's used to send positioning information to an output device. For instance, Control-H will send the code that is used to mean "backspace," and Control-M sends the code for "carriage return."

You may ask why we have a key on our keyboard that is used to generate output formatting codes. The reason goes back to punched tape. Teletypes were able to read and generate punched tape, and systems used the tape for program input and output. One way of editing programs was to edit the tape "off-line," by reading an old version in the tape reader and creating a copy in the tape punch until you reached the point that needed changing. At this point, it was perhaps necessary to insert formatting code such as a carriage return. The operator was supplied with a Control key on the keyboard to generate the control characters, not for sending to the computer but for placing on the tape.

So courtesy of some old technology, we have the ability to create 32 control-character codes from our keyboard. The editor can interpret these keys as commands to move around the file. The designers of text editors have often tried to make the action of the key mnemonically related to the legend printed on the keyboard. So in `emacs`, for example, Control-E means move the cursor to the *E*nd of the line and Control-P means move the cursor up to the *P*revious line. Of course there are rare occasions where the user wants to insert a control character into the text, and so the editors provide a quoting character, which indicates that the next character that is input should be inserted directly into the text.

However, modeless editors soon run out of control keys and need to find some way to provide more actions. Some early terminals had a Meta key that was used in a chord to set the eighth bit in the character that was being sent. ASCII is a seven-bit code, designed to be sent down a serial line one bit at a time. The transmitter waggles the line up and down using a predetermined frequency (the baud rate) and the receiver scans the line at the same frequency to reconstruct the value for the appropriate bit in the character that is being sent.

The top bit was originally intended to provide a check that the data is OK. The idea is that the bits that are "on" in the character are counted, and the result must either be odd or even. You could often choose whether you used odd or

even parity. The top bit in the eight-bit code is used by the transmitter to create a "parity" bit. The transmitter counts the bits in the seven-bit character that it is sending and forces the top bit on (or off) to create an even or odd number of bits. When the character is received at the othe other end of the wire, the hardware that reads the character will reject it unless the total number of "on" bits sums to an odd or even value, depending on whether odd or even parity is selected. Because line errors mostly mean that the serial line is forced fully on or fully off, this technique picks up "bad" characters and filters them from the data stream.

If the communication between the terminal and the computer is a reasonable connection, parity checking is not needed and the top bit can be used to extend the character set from seven to eight bits–a ready-made source of additional control characters.

The* emacs *editor further complicates matters by allowing the user to bind any key to any action they wish. There is a default set of actions bound to a default set of keys, but if these don't suit you, you can rebind them.

However, only a few terminals were supplied the modifier key that forced the top bit of the character to be on, so a short key sequence was used to emulate the coding. The Meta key (usually the Escape key) is pressed, followed by a standard key from the keyboard. The editor picks up the character pairing and treats it as a single action. Meta key actions are usually used for navigation and the settings are often related to the control key sequences. So, for example, Control-B usually means "back one character" and Meta followed by "B" means "back one word."

The default `emacs` editor key sequences also use the Control-X character to introduce a set of sequences that extend the available command character set further, providing for "control commands." For example, it's useful to know that to get out of `emacs` or one of its clones, you need to type Control-X followed by Control-C.

The `emacs` editor (and many of its clones) further complicates matters by allowing the user to bind any key (or key sequence) to any action they wish. There is a default set of actions bound to a default set of keys, but if these don't suit you, you can rebind them. It's a bad idea to arbitrarily rebind keys for the standard key set, because one day you'll be in a place without your default setup file and will be lost because your body will not know how to drive the editor. This keeps happening to me, so I know.

These days, keyboards are generally integral to the computer and no longer send ASCII codes directly. Instead, they generate key press events and transmit a value that identifies the physical key on the keyboard when a key is pressed and another when it's released. Software tracks keyboard state and maps the key number onto a character code that's sent to the program.

The X Window System allows you to remap the key code translations. If, like me, you hate the Caps-Lock key, you can simply remove it from your keyboard by telling X to stop using it. Also, on my standard Sun keyboard (designed to support PCs), I find that the Escape key is in the wrong place and is difficult to reach. I want my Escape key next to the key that generates the digit 1, so I've remapped it and swapped the key covers.

Incidentally, you'll find that the arrow keys on your keyboard are probably mapped by the X server into a sequence starting with Escape and an opening square bracket. A letter that's different for each key will follow this sequence. X is emulating an ANSI standard terminal and generating a standard sequence for the key.

Finally, the eighth bit in the character set is no longer available, at least to me, because the ASCII code has been extended into an eight-bit code, with the extra values being used to display characters with accents needed by major European languages. I like to be able to type those characters to give me easy access to my currency symbol and the ability to generate correct spellings of the names of my friends. It means that I cannot use the Alt key on my keyboard to generate eight-bit Meta instructions for editors. I have never done so because I am wary of keyboard portability and don't want to train my body to use keystrokes that will sometimes not be present on a new system.

UNIX Editors

Both `vi` and `emacs` are visual editors, designed originally to work with terminals with addressable screens. The user sees a screenful of information that is a window into the file. With `emacs`, you can split the screen vertically to see parts of the same file or display parts of different files on the same screen. Of course, nowadays, `emacs` is "X-ified," so parts of each file you are editing appear in a separate X window.

Editing on UNIX started with `ed`, which was designed before even glass terminals were invented. It's a line editor, presenting the user with a set of commands that may be applied to any of the lines in the edited file. Essentially, there are two modes: command mode, where you type in commands to make the editor change the data; and text mode, where you type lines of text after a command, and these lines are added to the file.

I still find myself using `ed` to create tiny files or make quick changes. Why? Because it's easy and fast. I can start `ed` quickly and it doesn't involve me searching for the mouse or removing my hands from the keyboard. The `ed` editor is available on every UNIX system and, because it was designed to be used on teletypes, it needs only the simplest of environments, taking input from the user and displaying its results as a data stream. Consequently, `ed` is always usable. If you can log in, you can edit text files with `ed`.

The `vi` editor was developed from the `ex` editor, which was Bill Joy's version of `ed`. He'd coped with some of the perceived problems with `ed`, such as the lack

of any error message other than ? (and TMP when temporary file space was low), and some people thought that "real" error messages would be a bonus. His ex editor also provided prompts. When you start ed it gives you no prompts at all to show that you are expected to type things. Again, some people think this is a bug. I am used to it.

The visual part of vi was glued onto the editing engine supported by ex–and you can move in and out of both editing styles simply–so ex's powerful substitution and global editing features that were inherited from ed are added to a visual front end. I said that vi has two modes: command mode where regular keystrokes are used to move about the file being displayed on the screen; and input mode, where keystrokes are entered into the file being edited. These modes really only apply to visual working, a further line editing mode is supported when you type a colon and enter ex commands to change the file.

On balance, moving about a file in vi takes fewer keystrokes than the regular emacs command set. Quite a lot of thought was put into the design of vi, choosing keys that were in easy reach. For example, H, J, K and L in the center of the keyboard moves you left, down, up and right, respectively. I taught myself to use these keys by playing the "hack" game endlessly for a couple of weeks before the need to do real work crept up on me.

Terminal Types

The vi program had a further problem to conquer: output portability. The fundamental prerequisite for using vi is that your terminal supports "cursor addressing," which makes it possible to send a command terminal to place the cursor at some random position on the grid of characters on the screen, where subsequent characters will be written. Early glass terminals behaved like teletypes, taking characters from the serial line and adding them to the bottom of the screen, then scrolling the whole screen up as each line was added. Even with these simple electronics there was a choice, if the screen is 80 characters wide, what happens when you send 81 characters? Some terminals scrolled auto-

matically. Some just lost the data that was "off" the screen.

As time moved on, the electronics in terminals became more intelligent and soon the computer could send character sequences that positioned the cursor, and the terminal could then place text that arrived in amongst other text on the screen. Terminal line speeds were slow, and so program designers wanted to make use of the stored state on the screen, sending the minimum number of characters to change the screen to match the state of the stored file being edited. The choices offered by the terminal manufacturers multiplied. For example, when you moved the cursor to a point on the screen and typed something, the terminal could overwrite what was there, or it could insert the characters into the screen memory, moving any existing characters to the right.

The `vi` program, then, had a varying range of terminal capabilities to deal with; some terminals could do one thing, some another. Worse, they tended to use different control sequences to achieve the same action. Eventually, the Digital Equipment Corp. VT-100 terminal established a set of control sequences that became an ANSI standard, but by then it was too late to do anything about the vast installed base of terminals.

Originally, programs were converted to use termcap or terminfo to make them portable across a range of terminals. Now, we are creating terminal emulators in software and want them to be compatible with the installed software base.

To cope with this problem, `vi` used a terminal capability database, `termcap`, that defined what each terminal could do and what character sequences were needed to achieve certain effects on the terminal. I suspect that the creation of this database was one of the first worldwide cooperative, free software efforts. People were encouraged to send in terminal definitions that allowed `vi` to work on their brand of terminal. The original text-based system was used by AT&T to generate a binary system, `terminfo`, which defined new capabilities. The terminal capability system was eventually broken out of the `vi` editor, becoming the `curses` library so that any visual program could be made to be portable on any terminal. On the whole, we are left with the legacy of both these mechanisms on systems today.

Nowadays, we tend to use terminal emulators that mimic the behavior of the ANSI standard terminal, and take this for granted. It's a kind of a reversal. Originally, programs were converted to use `termcap` or `terminfo` to make them portable across a range of terminals. Now, we are creating terminal emulators in software and want them to be compatible with the installed software base.

Which Editor Should I Use?

Your choice of editor has long been a religious decision, and you risk being branded as a heretic if you endorse one and not another. My career has moved along with UNIX, so I started using `ed`, then switched to `em`, which was a version of `ed` with simple interactive editing on one line supported by control keys. It was `em` that gave Bill Joy the idea to create `vi`. I then used `vi` for many years.

When `emacs` first appeared, I looked at it. To begin with, it was big, slow to start and lumbered along in comparison to `vi`. Basically, it needed a stand-alone VAX-11 to support it, and at the time we were putting more than 50 people on one VAX, so we couldn't afford that. My arguments about the slowness of `emacs` are always countered by `emacs` lovers who do everything in the editor, using it as their shell, editor, mail program and almost everything else. "I fire up `emacs` at the start of the day, and stay inside it all the time," is a common statement from aficionados. Many of the arguments about slowness and program size have undoubtedly gone away because machines are fast with large memories now.

However, as time went on, I became slowly disenchanted with `vi` and switched to `jove`, largely because `jove` was about the same size and speed as `vi` but supported multiple buffers and multiple windows on the same screen. It also didn't have the idea of the "line" burnt into it quite so firmly. The `jove` editor also supports regular expression matching, which the early `emacs` editors didn't. I have dabbled in other editors, but have tended to stick with `jove`, which runs on every machine I own except my Psion palmtop.

If you are interested in starting with an editor, then dabbling a little with `ed` or `ex` is good training for general UNIX use, lots of what you learn will spill over into other programs and systems. You will find that you can do repetitive edits and other command tasks very quickly after some little march up the learning curve. If nothing else, it will show you the problems that `vi` or `emacs` are trying to solve in their separate ways.

If you need to be portable across UNIX systems, then `vi` is a good choice, it exists on all machines and is consistent across them all, so what you learn on one system will be usable elsewhere. If you've dabbled in `ed` or `ex`, you can dip back into their command set while learning `vi`. The work that you are trying to get done can happen while you are struggling to train your reflexes to know what to type when you think "up three lines and along a bit."

It's also a good idea to have a passing acquaintance with the standard `emacs` command sequences; many systems have these embedded as a line-editing standard. So, for example, you can use the `emacs` line editing set in the Netscape browser URL selection window, because X implements the controls for its standard method for line editing.

***Peter Collinson** runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.