Date: 16 July 1979

From: Bernard S. Greenberg

To: All concerned parties

Subject: Terminal Features Memo

This Multics Technical Bulletin consists of a paper I have written describing what terminal features are most desirable for use in real-time editor and managed-video applications. It is based upon my experiences over the past year and half in supporting terminals for Multics Emacs, as well as the experiences of other researchers outside the company with similar software.

This memo might also serve to some as an introduction to the features of consumer video terminals; it has been found valuable in that use by many.

This memo has received wide distribution within the Company, and substantial distribution within the ARPANET and MIT communities, and has been distributed to some terminal manufacturers, within and outside the Company.

Unlike most Multics Technical Bulletins, this memo is not limited to the Multics Development Community; in fact, it will do the most good if it is distributed as widely as possible to those interested in terminal design and support. It may be reproduced without permission, as long as its title pages and origin are left intact.

Date 5/8/79 (vers 5)
From: Bernie Greenberg (Honeywell, CISL)
Subject: Desirable terminal features for support of Emacs.
To: All interested parties.

Many times I have been approached and asked about certain Emacs screen management features, and have responded that the lack of appropriate terminal features renders the proposal infeasible. When asked to detail what these features would be, a long list at once comes to mind, as well as an equally long list of what is wrong with today's terminals, in specific and in general. Thus, I have here committed to writing these opinions, wherein I state what features are most desirable and undesirable in current offerings, and what features are most needed. This document results from the cumulative experience of interfacing dozens of terminals to Multics Emacs, and learning of the vagaries, peculiarities, and limitations of each. It is hoped that this document will provide some feedback to those who wish to design, sell, or purchase video terminals, from the implementor of a system which attempts to exploit their features.

I am discussing, in this document, character-asynchronous ASCII character-video terminals. I am not discussing storage-tube or other graphics-oriented devices, but devices usually implemented with raster-scan technology. Whether the internal organization of the terminal involves bit-map memory or scanned character-generated video, is not of immediate interest. If any calligraphic device attempts to provide a dynamic, character-asynchronous interface, then it, too, falls in the category under discussion.

I am not claiming that terminals that lack all of the features that I consider worthwhile are necessarily bad terminals, although for many of the things I discuss, this is precisely the case. I am claiming that these features do determine suitability for the support of any real-time video editor. These features and needs are not peculiar to Multics Emacs: other implementors of real-time video editors agree that these needs are universal. Indeed, these discussions are not limited to applicability for mainframe-oriented editing; any conceivable implementation, involving minicomputers, distributed networks, or multilevel programs in different processors distributing the work of editing would find these same criteria to be equally important.

When I use the term "remote host", or "host", in this discussion, I will be referring to the computer system which effectively controls the screen of the terminal under consideration, and receives characters from its keyboard. I do not specifically mean the "central system" as opposed to the communications processor, or imply any geographical distance. I will be referring to what ever system interfaces to that terminal, or to what I am considering as the terminal. By "software", I will be referring to those programs running on that system.

The first part of the discussion will consider things that most terminals have, that we use now, and that they should continue to have.

Support of Full ASCII

Multics, and most other usable time-sharing systems, use the full *ASCII* character set, upper and lower case. All documents on Multics, and most other time-sharing systems, are of intermixed case. It is unreasonable to expect people to edit documents on terminals that can only display upper-case. While these terminals may be adequate in some batch-oriented situations, or for editing source programs in some implementations of some programming languages, they are completely unsuited for use in interactive time-sharing where document preparation and editing are to be performed.

Multics Emacs uses the full ASCII character set, including the ability to transmit all 128 characters from the keyboard. Most other similar implementations are based on the full ASCII character set. The presence of a key to transmit the ASCII "ESC" (escape, 033) character is needed for Multics Emacs. The presence of a "Control" key is mandatory. The control key is expected to be on the left-hand side of the keyboard, although one on each side is optimal. ASCII defines control characters as being 100 (octal) less than the same character without the control key. Alphabetic characters depressed with the control key depressed are expected to produce 100 less than the corresponding upper case alphabetic. Thus, "Control X" on almost any ASCII device will transmit "030" (X = 130, 130 - 100 = 030). The documentability of the editor relies on being able to state that "Control so and so does this..".

I would not even bring this point up, except that the Teletype Model 43 does otherwise, re-using Control J and Control M for less-common codes because a Control J and a Control M can always be generated by the Linefeed and Carriage Return keys (on that or any other ASCII terminal). Thus, the Emacs documentation, which says that "Control X Control M" is "Execute a Multics command" (note the purposeful mnemonic value) does not work on this terminal. Similarly, a larger number of terminals place Control Underscore (037) and Control Circumflex (036) in obscure places, or have no way of generating them at all.

The presence of a "Rubout" key (generating ASCII DEL, 177), is highly desirable. Rubout, TAB, and other ASCII-defined control codes should be accessible on keys that do not need to be shifted to generate these codes; having to "shift" to use "rubout" is a common misfeature.

Feature Accessibility From Host

All terminal screen control functions should be available from the remote host. This is not to be taken for granted; the Tektronix 4023 and the HIS (Honeywell) VIP7700 (for two that come to mind immediately) provide line and character insert and delete through local keys, but these features may not be invoked from the host computer.

Local options settable via switches on the terminal should, optimally, be settable from the host as well; the only way that software can be _sure_ that the user has set all the terminal options correctly for a given application is to set them itself. A program that needs to type out "Please turn the auto-crlf switch off and type 'go'" is one operating under stress indeed.

A video editor must manage the screen entirely and completely. Every last character and cursor position on the screen must be placed there or ordered by the host computer; there is no "local editing" or "local function". If the user appears to be "just typing in text", it is by the grace of the video editor. This is the basis of the video editor philosophy; arguments about relevant implementation efficiency or alternative implementations are not appropriate here.

The terminal must be capable of full-duplex operation, the keyboard and screen completely dissociated. I have observed the ADDS Regent 200 terminal, on which holding down a shift key influences the interpretation of control sequences coming from the host! "Features" like this go a long way towards making a terminal completely unusable. The terminal must be capable of performing screen control operations as directed by the host at the same time the operator is typing; if, at certain line speeds, the microcode cannot keep up, loses controls, loses typed characters, or worse yet, dies, the terminal can be said to be "inoperative" at that speed. The Tektronix 4025 and 4027 perform unpredictably if input is typed while any screen-control operation is in progress. Characters should be transmitted from the keyboard to the host regardless of _what_ is going on on the screen.

The terminal should be resilient; if some communications problem, or even internal microcode problem occurs, the terminal should not enter a state where it must be powered off, the connection dropped, etc. Similarly, if some bad input is received, or thought to have been received, the terminal should "carry on". What _not_ to do (for example, a recent change to the HIS VIP7801) is go into an error state flashing a light, requiring the operator to intervene by pressing a special button.

Uniformity and Predictability of Function

Terminal control operations should operate in a uniform and predictable manner. Operations should be valid at any point of the screen. As a counterexample, the "insert-lines" function on the Tektronix 4025 and 4027 cannot be used on the top line of the screen. The ADDS980 has no clear-to-end-of line function, but an ASCII Carriage return will clear to the end of the current line and position to the next line. Thus, it can be used as a substitute for clear-to-end-of-line everywhere except on the last line. Terminal operations that need special treatment depending upon what the last operation was, or require going into and out of special modes to perform, are highly undesirable. Operations that leave the cursor in out of the way places are also to be frowned upon. Undocumented interactions between commands, not anticipated as "reasonable" by the terminal designers are another common problem.

As a rule of thumb, a terminal has reasonable and predictable features if an expert can add support of this terminal to Multics Emacs (or a similar parametrized screen management formalism, such as "CRTSTY" on the MIT-AI Lab PDP-10's) within twenty minutes, with no documentation other than a list of terminal control sequences. The syndrome of "you can't use insert-chars if the last .... was a ... or you just..." is an all too common effect which leads to user and implementor dissatisfaction in a short time.

Cursor Positioning

A terminal must have the ability to have its cursor positioned to any place on the screen without knowledge of where it is currently. The host must be able to move the cursor to any point on the screen. The character sequences directed at the terminal for this purpose should be as brief as possible; most termina manage to accomplish this in four characters. Some, like the Delta Data terminals, need ten characters (which is almost unacceptable).

Most terminals also support shorter sequences to effect relative cursor motion when the cursor position is known. ASCII BS (backspace, 010) for backwards one character, ASCII CR (Carriage Return, 015) for moving to the beginning of the current line, and ASCII LF (Line Feed, 012) for motion down one line in the same column, are common. Similarly, the ASCII ESCAPE (033) character followed by A, B, C, and D, for relative directions on the four points of the compass, are common. Although not strictly necessary, usage at 300 baud is rendered substantially more responsive by the existence of these features. Similarly, a single sequence to position the cursor home (The upper left-hand screen corner) is common and marginally valuable.

The ability to set tab stops and tab to them is a common and useful feature. Multics expects tabs every 10 spaces, most other systems expect tabs every 8. This implies that fixed, non-settable tabs (as in the DEC VT52) are suboptimal. Tabs allow more optimal cursor positioning, and more rapid output of normal (non-Emacs) output. A very common design defect in many terminals is clearing all tab stops when the screen is cleared. This makes tabs well-nigh useless on these terminals, for screen clearing is a much more useful operation than tabbing.

Cursor positioning without need for knowledge of where the cursor is currently located is also essential to robust terminal operation: should a cursor be mispositioned due to minor malfunction, accidental pressing of "local" keys, or other unpredictable mishaps, successive cursor movement should "recover" from this error rather than compound on it.

While on the subject of cursors, the point should be made that blinking cursors are highly preferable to static ones; the ability of the user to instantly find the cursor is of paramount importance; cursors that "invert" the character on which they sit are preferable to those that obliterate it. The latter are singularly unacceptable when they don't blink, as they totally obscure the character under under them. The Tektronix terminals require switching between two modes to access all features; in one of these modes the cursor is not visible; this is poor.

Mass Text Erasure

The ability to erase all of the screen, or from a given point in a line to the end of a line is crucial. Very few video terminals lack this feature; among them are the Tektronix character-oriented terminals, and the Lear-Siegler ADM3. Without this feature, spaces have to be transmitted at line speed to clear out regions of the screen for new displays. At low line speeds, this is intolerable. Inability to clear the entire screen at less than 9600 baud rules out a terminal for this use.

Insert/delete facilities.

"Insert/delete facilities" refers to two sets of common features, "Insert/delete lines" and "Insert/delete characters." Many popular terminals have both, one or the other, or neither (although insert/delete lines is more common than insert/delete characters). "Insert/delete lines" describes the ability to instruct a terminal to remove entire text lines from any point on its screen, moving lines below it up, without need for retransmitting those lines from the computer system, or "open up" blank lines on the screen, pushing lower lines down and some off the screen, without retransmission of the moved lines. "Insert/delete characters" describes a similar ability to open up blank space, or alternatively, insert new characters, into the middle of an existing line on the screen, or remove characters from the middle of a line, while preserving and moving the remainder (to the right) of the line.

The ability to reorganize lines on the screen by shifting existant text around en masse is critical to the use of video terminals via communications lines. Modelling of documents on screens often involves moving parts of lines on a line (i.e., the insert or deleting of characters), or moving lines around on the screen. The use of terminal controls to effect this motion relieves the software from having to retransmit an entire screen, or a large part of a screen. At 300 baud, which is not at all uncommon, 24 x 80 = 1920 characters takes one minute to transmit. This is an unacceptable consequence of doing something as simple as moving the editor's "cursor" to the first line of a document that is not on the screen.

The ability to scroll is a subset of the ability to insert and delete lines. Although "scrolling", as normally provided, allows text to move continuously "behind" the screen, the inability to move less than the entire screen or more than one line at a time precludes all nature of effective screen managment.

Similarly, the ability to insert and delete characters from the middle of a line is crucial to usability and lack of frustration at 1200 baud and below. Without this feature, typing in the middle of a line requires the rest of a line to be rewritten as each character is entered, which, for a long line at 300 baud, could be <u>seconds</u> per character.

Although one might argue that the software could avoid this by timing the characters as they went out to the terminal and avoiding screen update until they were gone, this is in many situations impossible. The current Multics Communications System does not provide facilites for doing this, and many real and projected networking situations preclude this as well.

Timing and Padding Problems

Terminal microprocessors should have enough computational resources, and their programs so coded, that the vast bulk of screen control operations do not require carefully measured sequences of "pad" characters to time out the microprogram. A reasonable terminal in this regard, the HIS VIP7200, requires one or two pad characters for a total screen clear at 9600 baud. A poor terminal in this regard, the Delta Data 4000, documents a whole table of pad requirements at different speeds for different operations. During the building of support of this device, it was found that the documented pads were often not enough, some operations that did not need padding, according to the documentation, required some, and random and non-anticipatable padding after and <u>before</u> certain operations had to be added. To this day, we have padding problems in the support of the Delta Data 4000. The conclusion seems to be that if extensive padding is required, the microprocessor/microcode combination is too slow to be used in a bug-free way at the speeds being supported.

The worst case of padding requirements is that of the Hazeltine 2000 terminal which requires an exact amount of padding; if too much is given, an unrecoverable error situation results. Given packet and character-oriented transmission networks, it is often impossible to make any statement about the time required to output a character sequence at all. In some network situations, it may be totally impossible to determine the ultimate user's line speed. The conclusion seems to be that while small and occasional padding requirements can be dealt with, extensive or exact padding requirements reduce a terminal's usability substantially.

* * * * * * * * * * * * * *

Now I will discuss things that most terminals do wrong.

"Function Keys"

Keys that provide local screen-control function occupy useful keyboard space, make the operation of a terminal more difficult, and scare naive users. They have no use at all in a real-time editor environment. On some terminals, these keys (clear screen, move cursor, etc.) cause these operations to happen locally no matter what the state of the terminal. This only serves to confuse the editor, which now can not manage its cursor or screen. On other terminals, these keys transmit terminal-defined "escape sequences" to the host. If interpreted locally by the terminal, these keys would effect the desired operation (although, as in the previous case, this is still not useful or desirable). However, with the terminal in full duplex, they transmit codes to the host which are interpreted as editor commands which generally have nothing to do with the function labelled on the key (nor could they be made to do so without destroying the editor's command set and documentability. The result is useless keys that cause user confusion.

These keys never describe operations that the system user wants to perform. They describe terminal-control operations such uas "clear the screen", "delete-lines", etc., which are primitives for a host to control a terminal not for a user to edit a document. A user never wants to "clear the screen", he or she wants to "refresh the screen with the document". A user never wants to "delete characters from the middle of a line", he or she wants to "remove a word or sentence". Thus, even if these keys could be made to "do what they say", they would be of little worth.

Another meaning of the term "Function key" is for keys like the above, which transmit predefined escape sequences, but are not labelled. Again, they are not useful in an environment where key commands are chosen for mnemonic value. ESCAPE-E in Emacs is "End of Sentence": a set of predefined keys that transmit ESCAPE-E, ESCAPE-F, ESCAPE-G, etc. is of no particular value or worth.

Yet a third meaning of the term "Function Key" (as on the Concept 100) is that of a downline-loadable key that transmits a particular sequence of characters, as instructed to by the host, when depressed. This functionality is easily provided in software from the host, and is not needed, although among the similar features being discussed, this type of "function key" is most readily adaptable to use in an integrated editing environment.

The extension of the keyboard via shift keys (see below) provides more commands, more mnemonic commands, more symmetric command sets, and greater ease of typing than any nature or sort of function keys.

Auto-Repeating Keys

This very common misfeature causes a key held down more than a second to start transmitting repeatedly as long as it is held down. Presumably, this is to facilitate underlining, spacing out, and other typewriter-like features. However, most implementations of this "feature" apply it to <u>all</u> keys, including control codes, ESCAPE, carriage-returns, etc.

A very common problem encountered by people learning to use ASCII real-time editors is created by holding down a key hesitantly, while groping for or thinking about the next key. this is especially common in the case of two-key sequences, such as those starting with the escape key. While groping for "escape d", the Emacs novice holds down "escape", then hunts the keyboard for "d". After a second, an infinite stream of "escape"s begins arriving at the host. The same is true for any key on many terminals.

The problem is accentuated by the fact that characters are not echoed locally, but by the host. In a network situation, the user with his or her hand resting on the "a" key may not see the stream of a's appearing until seconds later, at which point more "a"'s are already on their way.

The ability to insert twenty a's is easily provided by software (in Emacs, for example, one says esc-20 a), and need not be provided by the terminal. I have witnessed much more frustration because of the presence of this feature than for the lack of it; if terminal-generated repeating is to be had, a "repeat" key that causes other keys to repeat is acceptable.

Keyboard Features

An excellent feature provided by some terminals' keyboards is "roll-over", which is the ability of a terminal to remember in what order keys were depressed, and transmit the key codes in that sequence, even if some of the keys were depressed before the others were released. Fast typists often depress new keys before fully releasing old ones. Terminals poor in this regard ignore all keys depressed while any other key is fully or partly depressed, or transmit garbage in this case. This causes the typist to observe "errors" that he or she did not in fact make. Terminals that support "two-key roll-over" will correctly transmit one key depressed while some other key is down. The best in this regard, such as the HIS VIP7801, having "n-key roll-over" will correctly send out the exact sequence of keys depressed by a person putting the palm of their hand down on the keyboard.

Tactile or audible, verification that a key has been adequately depressed to transmit is highly desirable. Keyboard feedback is a large component in a typist's development of accurate typing habits on a given terminal. The average computer terminal is still highly inferior to the standard office typewriter in this regard. The old IBM 2741 keyboards were excellent in this regard; IBM did much research in typewriters to develop this product. In a real-time editing environment, where response to some commands can take a half a minute, and, in a network environment, simple echoing of characters upon the screen can take a second or seconds, the typist's feel for what he or she indeed typed becomes particularly important. Keys that "commit" to transmitting once depressed past a certain point, and do not transmit again until released <u>past</u> that point would help a great deal.

Acoustic feedback is less valuable than tactile feedback. When keys have "clickers", no one seems to be able to agree how loud they should be. Although potentiometers to control this are desirable, the tradeoff between noise and feedback should not have to be made. Not only are noises made by terminals undesirable in quiet environments (the usability therein being one of the principal advantages of video terminals), but they may be inaudible in noisy environments (e.g., computer rooms). Use by the deaf and those wearing headphones are serious considerations not to be ruled out. (Much the same is true, by the way, for terminal "bells", or "beepers", which ought be at least selectable as an option to be mapped into screen-flashing or similar silent activity).

The matter of key placement is one of the most frustrating and impossible to circumvent. Two general keyboard layouts have acheived popularity, that derived from the office typewriter (which seems to rapidly be becoming the standard), and that derived from the old Teletypes. Gross and significant deviations from either of these standards are a significant problem. The HIS VIP7200 is perhaps the worst offender I have encountered in this regard. Users continually hunt for the "/" (which is where the "+" key is on most terminals) and other unpredictably misplaced keys. The "@" (which, on Multics, causes a line of input to be deleted) is where the left-hand shift key is on most terminals, with the shift key to the left of that. This causes many attempts to hit the shift key to send an "@", killing a line of typed input.

The misplacement of ASCII function and terminal control keys is a much more significant and widespread problem then bad placement of ASCII characters. The Teleray 1061 places the Rubout key next to the Control key; the latter is often held down for extended periods of time: when the former is mistaken for the latter, the former's auto-repeating feature causes large bodies of text to be "rubbed out" before the user realizes what has happened. The VIP7200 places the "break" key, which breaks the transmitted carrier, perilously close to the "return" key: most systems (including Multics) use the "break" key to abort (perhaps restartably) a running computation and/or discarding of unwanted output, and the result is often loss of a small or large amount of work, and notable user aggravation. The "break" key should be as out of the way and as hard to reach as possible: the same applies to any key that performs transformations upon the terminal's state or the state of the communications line.

Obscure "Features"

Some terminals implement certain functions, or implement certain functions in certain ways, that make organized use of the terminal difficult.

A good example is the "automatic carriage return and line feed" performed by many terminals when a character is output in the last column. In non-managed video environments (i.e., the so-called "glass tty"), this feature is useful. Yet, when organized screen management is attempted, software must account and compensate for this action: characters output in column 80 act "differently" than characters anywhere else. Experience has shown that features like this should at least be controllable, most preferably from the host itself.

A classic mis-design in this regard is the flow-control protocol of the DEC VT52: If it receives characters faster than it can process them, it sends a "Control S" (Code 023) to the host. a "Control S" is interpreted by many DEC operating systems as "Stop typing at once". This works tolerably well, unless the terminal is used with some other operating system, which may interpret Control S to mean something else (for example, it is "Search" in Emacs). In the words of another real-time video editor implementor, "The DEC VT52 goes out of its way to make it difficult for you to use the Search command." There is no way that the software can differentiate between a Control S typed by the user to mean "Search", and one issued by the terminal because of input buffer overflow. The aforementioned terminal also pre-empts the use of Control Q to mean "output may now resume." If terminal padding and processing requirements require flow-control protocols, the charcters sent to control this function ought at least be host-settable. In general, the more that is host-settable, the better.

Another good example is the HIS VIP7200, which causes a bell to ring when column 72 is passed and a printing character is transmitted. This "local bell management" causes the bell at column 72 to be an undocumented feature of any subsystem being used on the terminal. "Why did Emacs ring the bell?" is a common question. It didn't, is the answer.

It is easy enough for Emacs, or any other program, to cause the bell to ring when a certain column is passed, if that is what it wants to do. It is impossible, however, to turn off this "feature" of the VIP7200 if we do <u>not</u> want this bell to go off. The real bug here again is the terminal designer's preconceived assumptions about the modes in which the terminal would be used.

A worse class of bugs involves mis-implemented features. The Delta Data 4000, again, implements "Insert lines" by ingeniously placing "markers" in its screen memory. So ingenious is this implementation that use of insert-characters at a point above these "markers" on the screen causes the bottom of the screen to roll backwards, unpredictably. Furthermore, if the terminal's cursor is positioned beyond one of these "markers" on an inserted line, all characters that the software attempts to print there are discarded, in an equally undocumented fashion. Between these two bugs, Emacs has to have a highly special case, disabling <u>all whitespace optimization</u> for this, unfortunately, popular and widespread terminal, and refraining from using its insert-characters feature <u>at all</u>.

The point of this example is that any checkout of this terminal using any real-time video editor on any system, or consultation with any person familiar with these issues before the release of that microcode would have uncovered these problems, and hopefully, caused them to be fixed (although in all fairness, the Delta Data 4000 predates most real-time video editors).

A related implementation problem is evident in the Human Designed Systems Concept 100 video terminal, which makes a differentiation between spaces on the screen put there by outputting a "space" character and spaces not written into. Sundry terminal features treat these two differently. There is no obvious difference to someone looking at a screen between the two, but in order to use the insert/delete features, software must know which spaces are "real". This requires that software attempting to support this terminal keep a special map of which kind of spaces are where, something that is necessary for no other kind of terminal. This makes support of this terminal a burden, and general, terminal-independent software difficult.

Another, less crippling example, is that of the ADDS 980, which has five different, asymmetric cursor-positioning commands. Supporting this terminal on any system involves writing ingenious code, where simple tables suffice for most other terminal types. The problem here seems to have been design of the command repertoire without adequate study of other products.

It seems to be the case that many usable terminals have "un-get-around-able" bugs and mis-features, that all persons dealing in terminal and real-time editor support have to program around, disable features on account of, warn their user communities about, and apologize for (explaining that it is not their software, but "bugs in the terminal"). If a terminal support system has to maintain a map of "special marks" or "which spaces are real" or "how far I can use on this line without outputting extra spaces to move the end-of-line over" or contain highly special-case code for support of some given terminal, that terminal can be guaranteed to be a source of frustration, dissatisfaction, and negative remarks. The answer here seems to be better communication and checkout.

Fields

Many current video terminals have the ability to highlight underline, blink, etc., text, a facility known as "visuals" to the terminal-design trade. These facilities are almost uniformly designed to support "forms" applications, an admittedly economically important domain, but not the one we are dealing with here.

What I want to do, as an editor implementor, is cause underlined text to be shown as underlined. What is more, I would like the standard ASCII representation of underlined text to cause underlined text to appear. That is, I want A, backspace, underscore, to appear as an underlined A, just as it does on every printing terminal. This requires one extra bit per character position of screen memory, causing the lowest scan position of each character to be "force-or'ed" on, if on. Underlined characters should be deletable by insert/delete character operations, and the clear to end of line and clear to end of screen operations should remove them as they would any other characters, and <u>not</u> leave underlines on the screen (as in the HIS VIP7800).

Current implementations of underlining and other field features often have problems that make the features unusable even if the programming difficulties were to be circumvented via complex field management: the Teleray 1061 requires a screen position to hold the start-and-end-of-field marks; this makes it impossible to display an underlined word correctly. The HIS VIP7800 limits itself to sixteen underlined "fields" per line.

Editors usually view underlining, highlighting, and other visual attributes as properties of <u>text</u> (like fonts, case, etc.), not properties of regions of the screen. When text with "visual attributes" is moved around or erased by terminal control operations, the attributes should be moved around or erased with it. There is a need for a terminal command to say "all text that I am going to output, until the terminal command that undoes this one, is to be underlined and blinking (or whatever)", as well as for the currently common commands that affect text already on the screen.

* * * * * * * * * * * * * * *

Now I will consider features that some terminals have, that make them extremely usable for managed video environments:

Screen regions

This feature is present on the Concept 100 terminal. It allows the host program to define a rectangular area, a region of the screen, to which all operations will be limited until the next such re-definition. This allows multiple documents to be displayed on the screen simultaneously. Two such documents may be placed "side by side", the host program redefining the "region" appropriately when necessary. With this feature, for instance, insert/delete line operations (scrolling) may be performed on the left-hand document without affecting the right-hand document. Without this feature, insert/delete lines (or for that matter, ANY of the screen control operations common on terminals) could not be used at all if side-by-side documents are to be supported. Then all of the advantages gained by the use of these features (including clear-to-end-of-line) would become inaccessible. This ability to define smaller "screens" on the main screen and operate upon them independently is very common in video-managed environments.

A limited form of region-definition is available on the new DEC VT-100. It is like the region-definition feature defined above, but, (in the DEC implementation) is limited to "horizontal" regions whose width is always the full width of the screen. Issuing of appropriate commands causes the region to scroll the way most video terminals do when a newline character is issued on the last line.

Emacs defines a concept of a "window", a slice of the screen in which a given document is being edited. Scrolling through a document in a window is usually effected by deleting lines at the top of a window then inserting lines in the middle, and writing in the new text (On terminals without insert/delete lines, there is no choice but to rewrite the entire window). This has the disquieting effect of causing all the text below the window being scrolled to jump up and down during this operation; the effect on the new DEC terminals is extremely pleasing; ability to scroll an arbitrary region of the screen in this fashion would be even more valuable.

An optimal implementation of region definition would provide a two-character escape sequence, and the four vertical and horizontal coordinates involved as four binary bytes (best offset by octal 40, to allow printability). An optimal implementation of region scrolling would provide two two-character escape sequences (one for up, one for down), and a number of lines to scroll, encoded as a 40-offset binary byte. The previously-defined region would be scrolled in its entirety, deleting scrolled-off lines and inserting blank lines for the lines made afresh. Optimally, the cursor should not even be affected, although possible implementations of this might have the cursor as designating one pair of arguments (e.g., the upper-left-hand corner of the scroll-box) and explicit arguments to the scroll command to define the lower-left-hand corner, or leave it or require it to be "home" in the region. I consider the DEC implementation, which involves going to the bottom of the region and issuing linefeeds to scroll up, or going to the top and issuing ESCAPE-M's, both one at a time, inferior.

Multiple Operations Per Character

It is highly desirable to be able to delete a large number of lines at a time; it is no more time-consuming for a terminal to move the same body of text to a lower position in screen memory than a given one (as a matter of fact, there will be more blank lines, so it is <u>less</u> time-consuming). It is slow and annoying to watch multiple lines be deleted one-by-one, then re-inserted one-by-one when scrolling is needed. The software knows exactly how may lines it wants to insert or delete, and the terminal can do it efficiently, but the representation of the commands and the transmission line speed make multi-line insert/deletes, which are extremely common in video editors, unattractive. Extended line insert/delete commands that take a count of how many lines to insert or delete would be extremely useful. Similarly, a "insert characters" or "delete characters" command that took a count of how many blank character positions to open up or how many to delete would be attractive.

Note that opening up many character positions for character insert, with the software then overwriting them, is preferable to "enter insert mode", "write the characters", "exit insert mode". Placing terminals in "modes" is highly dangerous. Any unanticipated problem that may occur while the terminal is in that mode can have disastrous effects. However, the usual alternative offered to insert-mode is a terminal command that opens up one character position (for example, on the Teleray 1061). This requires many overhead characters to be sent for inserting a multi-character string, which makes it less preferable than "insert mode", in spite of the problems of the latter.

* * * * * * * * * * * * * * * *

Now I will discuss features that most terminals don't have, but that I would very much like to see.

Large Screens

The 24 x 80 video screen is a major limitation. Formatted documents may not be viewed in their final form. Programs whose listings are usually produced on online printers develop a cramped, 80-character style when prepared on 24x80 terminals. Attempting to read or edit text typed in on wider, printing, terminals results in either "folded over" lines, which are confusing, unsightly, and reduce the effective screen utilization even further, or truncated lines, which throw away information and are most unnatural to edit.

Effective screen-only interactive environments necessitate dealing with many documents, programs, conversations, texts, etc., at once, just as the printing-terminal oriented user will have many books and listings open. The approach being taken by advanced video systems is to have many regions of the screen with different displays in them. The 24 x 80 environment makes this all but impossible, making video terminal interactive time sharing often frustrating.

For word-processing systems, a 60-line screen, at least 85 characters long, is a bare minimum. For full-line-width programming, 120 columns is not unreasonable. It is true that the time required to transmit large screens like this is potentially problematic, but effective screen utilization and management reduce the need to do this frequently.

The limitation on screen depth is often as serious as that on screen width; the inability to see a whole "eight and a half by eleven" page of text, or a section of a program more than 24 lines long is frustrating and problematic. Attempts to use multiple screen regions to increase the effectiveness of the screen cause even smaller fragments of text to be displayed.

Line Speed

While not strictly a terminal issue, line speed seems intrinsic to effective utilization of video display devices. Many terminal features (e.g., insert/delete lines), while valuable at any speed, are present to avoid intolerable screen-filling delays at low speed. Effective screen management, when many documents, programs, and scripts are involved, can involve constant refreshing and rewriting of large portions of the screen. Having reasonable line speed (1200 baud is in some sense the barest minimum for this purpose) allows powerful techniques such as interactive search and replace operations, showing document context around each occurence of the string being searched for or replaced, to be shown. At low speeds, the screen-filling engendered by such techniques would make them too slow and frustrating to be usable at all. Reasonable terminal speeds allow entire modes of interaction that could not even be considered at lower speeds.

The same is even more true of systems running at 10 kilobaud or better. This can usually be acheived only with hard-wired lines, or shared-logic systems such as the kind described below. There are enough of these hard-wired lines in use today that error-free operation at 9600 baud (or better) is an important criterion of terminal acceptability.

Shift Keys

A large repertoire of "Shift"-type keys adds tremendously to the potential command-set of an editor. By a "shift" type key, I mean a key which, when held down, affects the output of other keys. "Shift" and "Control" are generally like this, and with their aid, 50 or so basic keys can be made to generate 128 ASCII characters. Additional keys of this kind would generate additional "characters" via an encoding technique involving a reserved character, such as described below. The use of these keys allows repeated sequences of complex editor commands, without the possibility for error engendered by possibly mis-synchronized multi-key sequences.

Each "shift" type key provided doubles the effective number of editor commands, by virtue of any previously describable combination of shift keys (and some basic key) being available with or without the new shift key being depressed. MIT has for some years now been experimenting with keyboards with two extra shift-like keys, "TOP" and "META", which may be depressed or not depressed independently with any other key. This allows a flexible and symmetric editor command set, for instance, "Control K" is "Kill a Line", META K is "Kill a Sentence", and Control META K is "Kill an expression". Similar keyboards are also being utilized at Stanford University.

The standard technique for encoding these extra keys is to reserve one ASCII character for indicating that the next two characters to be transmitted are an encoding of which shift keys were depressed, and what basic character was struck. A bit pattern is reserved within this representation to indicate that the basic character used as the reserved character was itself struck on the keyboard.

Thus, if Control \ (034, octal), is chosen as the reserved character, then (recalling that "a" is 141, octal), then the following character sequences might be transmitted by the following key-combinations:

```
a    141        (lower case a)
A    101        (Shift (upper case), a)
     001        (Control, a)
     001        (Control, Shift, a)
                          (All these so far are standard
                           on all ASCII terminals today)
     034 001 141
                (Meta, a)
     034 002 141
                (Top, a)
```

```
        034 003 141
                  (Top, Meta, a) (Note the bit or'ing)
        034 002 001
                  (Top, Control, a)
    \   134       (\)
        034 000 034
                  (Control \)
```

The aforementioned MIT keyboards go one step further in treating "Control" as an "extended bit", which (via software) represents non-ASCII constructs such as "Control 3" via a formalism like the above. This augments the power and symmetry of the editor's command set even further.

Overstrike Capability

Multics makes frequent use of overstruck characters. It is central to the entire APL implementation. The simple case of overstruck underlines appears in almost every Multics file. Multics users expect to see overstrikes on their terminal output and line printer output. Although it is clear that bit-wise memory, or at least n-tuplicated character memory is necessary for the implementation of this feature, it is highly desirable, and lack of it is one of the most common complaints of video terminal users on Multics.

Positional Sensing Devices

The entire theory of real-time video editing centers upon the user's positioning the terminal's cursor to a character, word, paragraph, or other construct that he or she wants to delete, modify, move, add text to, etc., and then issuing appropriate commands or inputting text to effect the desired change. In most implementations, cursor movement is achieved by the user striking keys whose documented function (in the Editor) is to cause cursor movement; the editor responds to these keystrokes by both moving the cursor and taking cognizance of a new "point of interest" in the document being edited.

Another common and effective use of cursor movement is to allow the user to select one of a set ("menu") of options by displaying the possible options upon the screen, and allowing the user to position the cursor to a given option, and "select" it by depressing some key. This technique is extremely effective in relieving the user of the need to remember a large number of options or features, or their names.

Some advanced word-processing and video-oriented editing systems make use of electromechanical or electromagnetic devices attached to the terminal to move the cursor for these purposes. A fairly typical one is the "mouse", a small box (named for its size and shape, its cord being its "tail") which can sense in what direction and how far it has been moved. Sometimes this is implemented by wheels on the bottom of the mouse, controlling potentiometers, and other times by inertial sensing devices or more sophisticated analog devices. The direction and length of movement of the mouse in the X-Y plane to can be determined in the terminal (by sampling of the analog devices, which is relatively cheap). The terminal responds instantaneously to mouse movement (the user moving the mouse on the table) by "tracking" the mouse, which is to say, moving a (not necessarily the "only") cursor around the screen, "mapping" the mouse's movement on the screen. This requires no interaction with the host. When a button on the mouse is depressed, the mouse's "position on the screen" is transmitted to the host, encoded. Most forms of mice provide several buttons to increase the potential command repertoire. This facility allows a given text construct, menu item, one of several displays on the screen, etc., to be identified instantly, even in the worst load and response-time situations. There is no need to press keys of any kind, perform searches, guess word, character, or line counts, or so forth, with even less load on the host then would otherwise be required.

Among other implementations of position sensing devices are "joysticks", which are directionally-sensitive batons which may be moved in the X-Y plane to effect cursor movement, light pens, which are photosensitive devices applied to the screen by the user, which sensing the raster scan, enable the terminal to deduce the screen position where the light pen was placed, and magnetic tablets, which, by scanning a rectangular table with electromagnetic waves, allow the placement of a magnetosensitive "pen" to be used to track user movement over the screen.

Note that all of these devices, from the least sophisticated, expensive, and convenient (mice), to the most elaborate (magnetic tablets) have identical features of instant response, reduced load on the host, better user interaction, and small load on the terminal processor.

(It should be pointed out that tracking of a mouse or other positional-sensing device by the host is highly desirable, and allows sophisticated forms of interaction such as blinking a menu item to which the mouse is pointing and so on. However, this requires either sampling of the analog devices by the host, or the transmission of an amount of information which is not feasible over communications lines.)

Downline Loadability

The ability to place programs of my own choosing in terminals would substantially augment the performance and response of Multics Emacs. Through the implementation of admittedly complex protocols, the terminal and the remote host can split the computing load of real-time editing. This is NOT to say that local terminal editing is desirable! An integrated editor interface that provides powerful functionality, consistency and ease of use, and responsiveness is desirable no matter how it is implemented. User-provoked interaction via "transmit keys" and non-integrated software designs does not fit this criterion; integrated, distributed processing does.

Downline loadability would provide for the ability to customize all of the interface requirements and designs discussed above without the need for customized terminals.

Novel Design

Many interesting, integrated designs for video terminal interfaces have been devised. Earl Killian and Eugene Ciccarelli of Bolt, Beranek, and Newman (Cambridge, Mass.) have developed a scheme where every screen control operation is defined as an exchange of a rectangular screen area with some other rectangular screen area. An infinite blank area lies offscreen; insert/delete lines and characters, as well as region management and selective clearing all are special cases of this operation.

Downline loadability would provide a path for experimentation with such interfaces and an evaluation of their effectiveness.

* * * * * * * * * * * * * * * *

The long run:

The most effective character-oriented screen managment systems I have seen or heard about (MIT AI Lab, Xerox PARC, Stanford AI Lab) involve bit-map screen memory controlled by a minicomputer, with video being generated at the minicomputer location, and distributed by coaxial cable. A single minicomputer interface, via high speed to a mainframe I/O controller then suffices, replacing many communications interfaces. Or, in remote situations, a high-speed communications link between the mainframe's communications processor and the minicomputer suffices. Shared software, shared controller hardware, common screen memory, and shared I/O resources result from this design. The shared high-speed link allows screen filling at megabaud speeds, allowing a whole new genre of interaction techniques, without the need for expensive modems for each terminal.

The use of bit-map memory for video implies the ability to support graphics, and use multiple fonts, thereby allowing typesetting to be done on line. Overstrikes simply fall out; the single minicomputer interface can be designed to implement any or all of the terminal communications features designed above.

The centralized controller approach also allows more efficient character transmission. No human typist can type faster than 300 baud, no matter what the OUTPUT speed of his or her terminal. Input characters can be sent over telephone cable for long distances at very low speeds with no loss of response at all, allowing all the bandwidth available to be used for output.

Note that full-screen transmission (as done by "local editing" terminals), involving "transmit keys", transmits huge volumes of data at line speed, involving severe taxing of communications processor and line resources. Character-at-a-time real-time video editing makes no such demands, and is infinitely more powerful.

Many of the centralized-controller facilities marketed are oriented towards line-at-a-time transmission and synchronous lines. While the encoding technology (e.g., bisync, Polled VIP) is not of particular interest, the necessity of the user to hit "transmit" is unacceptable.

It should be pointed out that binary synchronous communications, with headers and trailers and all natures of overhead, is not particularly inefficient in a character-at-a-time environment. If a user is typing, sending one character at a time at typing speed, (and presumably, that character is being echoed at typing speed), twelve characters per second will fly back and forth for a good typist. If the basic line speed is fast, this is negligible, even at a 1000% overhead (10 overhead characters per every one) is still only 120 characters per second. If the line speed is lower than that, more than two people can't even type at the same time. For output, output is sent either charaters at a time, which is shown above to be no problem, or screens at a time, which is efficient.

I think the near future of interactive video systems lies in these concentrated, character-at-a-time, bit map environments.

* * * * * * * * * * * * * * * * *

Summary

The following, then, is a summary of desirable features in video terminals for real-time editing:

- o Barest minimum: without these, a terminal may be considered totally unusable.

  - o Full ASCII, upper and lower case, transmit all codes.
  - o Standard interpretation of ASCII Control key.
  - o Any "editing" or screen control function that exists is accessible from host.
  - o Capability of completely full-duplex (remote echo) operation, with no interference between keyboard and screen, at any speed at which terminal is claimed to work.
  - o Minimum 24 x 80 screen.
  - o Ability to move cursor to any location regardless of where it is.
  - o Ability to erase to end of line, and erase all of screen.

- Necessary features, without which a terminal is "poor," although possibly not totally unusable.

  - Quick, automatic recover from "error".
  - All terminal control operations operate in a uniform, context-independent manner at any point on screen.
  - Ability to insert/delete lines (absolutely necessary at low speeds).
  - Ability to insert/delete characters (ditto).
  - Blinking cursor, inverting if a "box".
  - Minimal padding. Padding on any operations other than clear screen and insert/delete lines is suspect. "Exact" padding is completely unworkable.
  - Multi-key roll-over.

- Highly desirable, uncommon features, that would help a lot.

  - Larger screen than 24 x 80.
  - Host control of all variable function.
  - Ability to have underlined characters without need for "fields".
  - "Shift-like" keys that cause prefix sequences to be sent for each key depressed while they are held.
  - Rectangular "region" definition, that limits all screen control operations to that region.
  - Scrolling of regions.
  - Ability to insert or delete more than one line or character at a time.
  - Bit-map overstriking
  - Character-associated text attributes.
  - Position-input devices.

- Bad features to be avoided. These cause the most cursing.

  - Operating-system or product-line dependent assumptions in choice of characters and protocols.
  - Self-repeating keys.
  - Poorly placed keys, deviating significantly from standards, and control-function keys to easy to hit by accident.
  - Unreliable operation at high speeds.

I would like to thank Earl Killian, Charles Frankston, Lee S. Parks, Dave Moon, and Robert Kerns for their valuable commentary upon this document, and suggestions thereto.

(END)