TO: MTB Distribution

FROM: Bernard S. Greenberg

Date: 04/18/78

Subject: Real-time Editing on Multics

I. What is it?

Real-time editing is the ability to edit or enter text seeing at all times what one has entered, as it sits in your segment, not as you have typed it. Real-time editing is the ability to modify a text, watching what happens as each command is typed, as opposed to asking the editor "Now, let's see what you did for what I told you". Real-time editing is taking advantage of intelligent and semi-intelligent display terminals: utilizing their ability to display text and alter it without retransmitting the entire screen. Real-time editing is the ability to say "Change this 9 HERE to a 10" as opposed to "Change the 9 which is followed by a close parenthesis, a space, a 17, a comma and a 0 to a 10", and find out that there were two of them.

Real-time editing brings the ability to have the text-entry subsystem participate in the preparation of text. Language assists, such as automatic PL/I indentation, "shorthand" expansion, or text justification, can be performed as the actual text is being typed in. Need to indent, expand, or post-process to match "begin"'s, "end"'s, or parentheses, vanishes.

Real-time editing is the text-processing technology of the 1980's, based on the latest data terminal technology of today. We can have this now.

II. Why do we want it?

Text preparation and program editing time are reduced by orders of magnitude by real-time editing. Instead of encoding instructions to a conventional text editor to try to convince it to do what you want, you DO what you want, watching what you are doing as you are doing it. Almost all of the steps of conventional editing vanish. Most notable are the absence of the "print" command and the substitution string, the most common of editor inputs.

A service-quality printing terminal (e.g., Terminet 300) can be purchased for $3000, and can communicate at 300 baud. A 1200 baud terminet is around twice that price. Printing terminals are noisy, slow, expensive, and difficult to maintain. They generate waste paper, and are in and of themselves a constraint on communications technology.

Microprocessor technology drives the price of intelligent terminals down daily. Intelligent display terminals capable of operating at 19.2 KILOBAUD are available for $800 or so. Even an extremely clever display terminal (i.e., the Delta Data 4000) is cheaper than the 300 baud terminet. Several Multics sites rely heavily on display terminals; they are the data entry technology of the present. The microprocessor will consign the printing terminal to the same place it consigned the slide rule.

Multics is now very weak in its support of display terminals. Users of such terminals, whose numbers are increasing in the Multics community, can now choose between "page length" processing, or watching data float off the screen at line speed. On the input side, editing using a display terminal on Multics is particularly frustrating, for text being edited vanishes off the screen as fast as you can type editing commands, and there is no hard copy to look back at. Indeed, most time spent attempting to use "qedx" or "edm", or even "TECO" from a display terminal is spent saying, "Let's look at these few lines again and see what they say". This is absurd, for video terminal technology was designed to facilitate editing of data as you see it.

Multics must adapt to video terminal technology, and support the growing variety of video terminals in the same flexible and general way it has supported printing terminals. Multics must keep in step with the forefront of technology from which it came. Multics must have the best, and be the best.

III. How does it work (on the surface)?

Real-time editing is accomplished by invoking an editor, a user-ring Multics program, as are conventional editors. The editor clears the terminal screen, except for a line near the bottom stating constant information, such as the fact that the editor is being conversed with, a "buffer name", and the pathname of what is being edited, if any.

At once, the user can begin to type text. There is no "input mode" or "edit mode". As the user types text, including newline, space, and tab characters, the characters appear on the screen as he or she types them. If a mistake is made, the striking of a "#" causes the previous character to be wiped off the screen as though it were never typed, as an @ will strike out a line. Even at this elementary level, the text appears correct and as intended at all times, instead of conglomerations of pound signs and at-signs. The displayed text is always right. The column position is always accurate (unlike today, when line editing forces us to type
"\fa" if we care what column we are at).

As the user types text, it is entered into a "buffer," as in conventional editors, which may have been initially filled from a segment, and may be written out to that or some other segment. As characters are entered, the terminal's cursor (a blinking mark which "runs" (lat., "curreo", I run) all over the screen) provides a visible indication where the next character will go. Normally, it is in the position right after the last character typed, so characters follow each other as they are typed. The basic notion of real-time editing identifies the cursor, on the screen, with a conceptual "pointer" into the editor's buffer, such as the "current line" of qedx or edm, or even more closely, the "." of TECO. The cursor is maintained by the editor to be at that character on the screen which is the "current character of the current line" in the editor's conceptualization. Thus, normal entry of characters "moves" the editor's "pointer" as characters are typed, which, as a side effect, moves the cursor on the screen as well.

Each character typed in this way may thus be viewed as a command to the editor to "insert" that character at the current "pointer", and move the pointer to the right of that character. Thus, typing "a", "b", and "c", in sequence means "put in an "a", a "b", and a "c"". Contrast this with qedx, where "a" means "append", "b" means "go to buffer", and "c" means "change". Similarly, "#" means "make the character to the left of the "pointer" disappear", which is what it has always meant in Multics anyway, but in the context of the real-time editor, the character actually disappears: both from the buffer and the screen.

Now the editor can be convinced to move its "pointer" around to any text in the entire buffer. This might be useful, for example, to insert a word between two words we had already typed. If we could convince the editor to move the "pointer" between the two words, simply typing the new word would put those characters right there (for we have already decided that "a" meant put an "a" to the right of the pointer and move the latter by one). As they would be typed, the user would see them appear in the middle of the chosen line as he or she typed them, the rest of the line moving over to the right as the new characters were typed, one by one. And if a mistake were made, just typing "#" would delete the character to the left of the pointer, as it always does, and that character would summarily disappear from the screen and the buffer, the rest of the line moving to the left visibly.

The editor can be convinced to move its pointer by typing "non-printing characters" at it. For instance, "control B"

(1) move the pointer "backward", i.e., to the left, and "control F" moves it forward. "control P" goes to the previous line, and "control N" goes to the next line. Now each time one of these characters is struck, not only does the editor move its pointer within its buffer, but moves the cursor on the screen to the corresponding position in the displayed text. Thus, the user moves the cursor around via these controls, to the point where editing is desired, and inserts or deletes characters once he or she has gotten there.

A buffer (representing a segment being edited) can surely be longer than the number of characters which can be displayed on one screen. If an attempt is made to move the pointer out of the part of the buffer which is on the current screen, the editor will provide for moving the viewed "window" (the 20 or so lines being displayed) to a different set of lines, so that the pointer is always visible on the screen as the cursor. With intelligent display terminals, this can often be done by actually instructing the terminal's microprocessor to move existing lines around on the screen, up or down to make room for new lines which Multics will transmit.

Now this is only the basic idea. There are much better and more sophisticated commands available, such as "Capitalize the word at which the cursor/pointer points" or "Move the cursor to the matching close-parenthesis of the open-parenthesis where it is now", or "Delete the next 6 words" or "Put over here the 6 words I just deleted from somewhere else". Some notion of the flavor of these commands may be gleaned from the protoype command repertoire in the appendix. In each one of these cases, a visual indication of what was done is displayed immediately on the screen, not by leaving a record of the editing commands, but by modifying the displayed text on the screen. It is more akin to the notion of editing a text with a perfect pencil and eraser than with a computer. The user is always viewing the text as it appears after the last character he or she has typed. Thus, there is no need for the editor "print" command, which means "Let's see what I've done". You are always looking at what you have done. One moves the cursor to THE character or word one wants to change or delete, pointing at it directly, instead of trying to invent a context which specifies it uniquely, as in printing-terminal oriented editors.

The editor manages the screen with knowledge of its capabilities and limitations. Text neither rolls off the top as editing is performed, nor need be printed in hardcopy to view what one is working on. The terminal, the user, and the editor cooperate and interact in a fashion which is completely new to Multics.

---

(1) "Control B" is generated on a ASCII keyboard by pressing the "B" key while the "CTRL" key is depressed.

IV. Why do you need all those silly control characters? They're not multicious at all.

It seems to make more sense that "a" should mean "enter an "a"" than "append". Since all the "good" characters (i.e., the printing characters other than #, \, and @) are used up in this way, we need control characters to say anything else. The default character assignments are reasonably mnemonic, e.g., "control B" for backward a character, "control E" for end of line, etc. Since this gives exactly 26 possible commands, and many, many more are desired, there must be multi-character escapes to express many of the commands. Thus, the ASCII "escape" (or "alt mode") key is used as a prefix to extend the number of commands, as is the "control X" character (1) in the default command assignment. Thus, the most common and useful commands are assigned to single-keystroke sequences (2) while less common ones require the "escape" prefix, and so on, until the least common require their full command-name to be typed (in response to escape-X, the default character for "read the name of an extended command and execute it").

The alternative to using "control characters" is to have "input" and "edit" modes, where the meaning of all printing characters changes in "edit" mode, as in qedx, edm, and TECO. This is one of the worst failings of these editors, as users happily delete lines by typing "dcl" in edit mode, write out segments named "ho", and insert sequences of "w", "q", and "pl1 foo" in their segments. However, printing-terminal editors are constrainted to printing characters, for the user must look carefully at his or her "substitute command" before unleashing it by typing carriage-return, in order to ensure that it will hopefully do the "right thing". This, of course is completely unnecessary if there is a blinking dot on the screen pointing at what you are about to delete or change.

All ASCII video terminals can generate control characters. There is no need to be able to see characters which are editor commands unless you don't believe that they are going to do the right thing if you "activate" them. All "delete" or "kill" type commands can be undone. If some command character typed deleted the wrong thing, one need only type "control Y" (for "Yank") to get it back. Since the effect of the deletion is

---

(1) All of the character-to-command assignments can be changed dynamically by any user at will, if he or she so desires. This goes as far as saying "when I type "~" I mean "E", because the "E" key on my terminal is broken."

(2) Not counting the "CTRL" key, on which one tends to hold one's little finger during other sequences than text-insertion.

viewed instantaneously, no further work is lost. Thus, there is no harm in simply commanding the editor to do what you want in real-time, undoing it if it was wrong. You don't ever want to see what you typed (with respect to editor commands): you only care what you have in the buffer, which you always see. Thus, you do not have to "plan" and "plot" editor commands, as with the conventional editors, before activating them.

Holding a "Control" key to achieve a different function of a key is no better or worse than holding "shift" to capitalize a letter. So much for control characters.

V. Can't intelligent terminals do this locally?

Well, some of them certainly can do some of it. Many terminal manufacturers provide a facility whereby you can edit the text on the screen via the use of specially marked keys on the terminal. Response is instantaneous, since only one user is using the terminal. When the screen is all correct, one can hit "transmit" and the screen will be transmitted to Multics.

This approach has several problems. The most serious is that there is no cooperation between Multics and the terminal. All changes made are made locally. If text is deleted, it is gone, and cannot be gotten back. You cannot edit any part of your segment except that which appears on the screen. No known terminal on the market provides for searching, word processing, or, in general, any form of text manipulation more sophisticated than the insertion or deletion of single lines or characters. The power and generality of the Multics software cannot be used.

Furthermore, since Multics does not cooperate in this venture, a random message or unexpected transmission messes up the displayed screen beyond repair; all work is lost, and Multics cannot regenerate it because it was not in on the editing. It is a situation as dynamic as setting lead type offline. Any unexpected situation spills the type on the floor.

Another good problem with the local editing approach is that not all terminals have the same local editing capabilities. Many very good and popular ones (such as the DEC VT52) cannot insert characters in the middle of a line, or move lines around locally. Using local editing, these operations are simply ruled out. Using real-time editing on the central system, it is a simple matter for the editor to simulate these operations in terms of the terminal's repertiore. At 1200 baud and better, the difference is barely noticeable.

Transmitting entire screens, or even lines, at line speed, places severe constraints upon the central system receiving end. Large buffers and split-millisecond allocations are required. This technology also requires high line bandwidth in the "transmit" direction at any speed. At lower speeds, the

time required to transmit a "screen" is simply annoying.

Local editing is not extensible. One cannot write editor "macros" for a read-only memory.

VI. Why don't you use those buttons with the arrows on them, and the "program function keys"?

Experience with this kind of thing has shown that one can edit faster if one keeps ones fingers on the normal typing keys of the keyboard, without moving them around for editing operations. It is partially this simple matter of "hand efficiency".

Not all terminals have the same set of special-function and program-function keys. The number, nature, and meaning of these keys is a function of the manufacturer and model of the terminal. By using normal keys and their "control" codes, the description and availability of editor functions is a specifiable in one place, the single document for the editor, regardless of what type of terminal is being used, what keys it has, or what functions it has available. Usage of the editor from a Honeywell VIP 7200, a Perkin-Elmer FOX, or a MIT-AI Knight TV Display is identical.

My sympathies do not lie with them that cannot understand that "control B" means backwards, for its lack of a left-pointing arrow, given that they understand the notion of "\f". The full capabilities of the intelligent display can be utilized without the use of random and variegated local keys; they are simply not useful in this context.

VII. What have you done?

I have designed and implemented a real-time display editor on Multics. Running on the MIT and CISL machines, it has, in the month since its inception, acquired an actual user community of about a half dozen. Display terminals are prevalent in the MIT Computer Systems Research and Programming Development Office groups.

The command repertoire and interface philosophy of this editor were borrowed from the "EMACS" editor running on the MIT AI/Mathlab PDP-10's. "EMACS" stands for "Editor Macros", for a screen editor implemented by Richard Stallman of the MIT AI lab as a set of "macros" (1) in ITS (2) TECO. Mr. Stallman is fully

---

(1) "Macro" is the term used in TECO and many other editors to describe user-coded programs and extensions. It is a poor term.

(2) For "Incompatible Time Sharing," the MIT-developed operating system used on the Artificial Intelligence Lab's PDP-10's.

aware of this endeavor, and most enthusiastic about it. Having used EMACS during the preparation of the 1978 offering of my MIT Lisp Course, I became acquainted with this particular interface, and enamored of it and what it represents. Nevertheless, EMACS is not unlike, in basic ideas, any of a half-dozen other state-of-the-art screen editors that come to mind, and neither MIT nor DEC has any proprietary rights to the particular commands and character assignments. The idea of real-time video editing is also in the public domain. The fact that I have preserved the interface of ITS EMACS has allowed an existant user community to utilize the Multics version of EMACS immediately.

EMACS is considered to be state-of-the-art. Cognoscenti have observed that no other video editor offers any particular advantages of substance. It is high time that Multics editing technology left CTSS-oriented editing and became state-of-the-art.

Multics EMACS, as my editor currently calls itself, is implemented in Lisp. I will discuss this below. It is extensible, programmable, self-documenting, and its code is transparently legible. It is already a powerful tool which I have used in my "normal" activities with great success. It already supports seven different makes/models of video terminal. It takes a skilled programmer about 10 minutes to construct a support extension for a new type of video terminal. The need to construct these packages in the editor as opposed to in Multics stems from the fact that Multics does not support video terminals with any kind of mechanism like the TTT with which printing-terminal extensibility is implemented.

The response to this project has been overwhelmingly enthusiastic from the user community. People have offered the opinion that it is high time that Multics had something like this. At sites such as USL, (which runs DD4000's exclusively, Multics EMACS' "favorite" terminal), the anticipated response can hardly be envisioned.

VIII. I have heard that you patch the FNP with hphcs_.

Indeed, this is true.

For asynchronous terminals, Multics is currently a totally line-oriented system. Character at a time interaction is essentially impossible. The FNP (1) will not transmit a line to Multics until a new-line (2) character is typed. The FNP-6180 interface protocol is organized upon the transmission of completed lines. It is precisely this technology which fosters the growth of line-oriented editors and premeditated substitute commands.

The Multics ARPANET implementation will transmit each character as it arrives, and wake up a the server Multics process. Since other hosts support character-at-a-time I/O, specifically the TIPs (3) the MIT community has had to resort to logging in to Multics from the MIT TIP or the PDP-10s in order to use Multics EMACS, because Multics itself cannot perform character-at-a-time input.

Thus, I have found it necessary, to create in the FNP (with the help of Larry Johnson, one of the local FNP experts) a CCT (4) which breaks on every character, and patch it into use for my channel every time I use Multics EMACS. I do this on the CISL development system during the time that "mini-service" is run. This causes every character to be treated like a "newline", and transmitted to the central system from the FNP. The response time, through the FNP, the central system interrupt side, the ring 4 program and back out again, manifests itself as a character-echo faster than LSLA echoing, comparable to HSLA echoing. Admittedly, there is little or no user load on CISL, and it takes quite a bite out of the system. This does not make it less useful or less good.

This "break and ship buffer on every character" mode is being proposed as an interim MCS extension to allow debugging and

---

(1) Front-end Network Processor, being a Datanet 355 or 18X.

(2) Carriage-return, line-feed, or either, depending upon various modes.

(3) A TIP (Terminal Interface Processor) is a node on the ARPANET which can be dialed up, like the FNP, from phone lines, and will allow the user to log into any host on the ARPANET "from" it. In the jargon of the ARPANET, the TIP is thus the "user" end of a "TELNET" connection, and the "foreign host" (i.e., the desired system) is the "server" end.

(4) Character conversion table, the hardware table in the FNP which tells an HSLA subchannel which characters to interrupt on

experimentation. It is planned as a tty mode for the near future. For demo and experimentation purposes, it will be completely adequate. It may even be totally adequate, and better DIA protocols may not need to be devised for some time.

The current line-oriented DIA (1) protocol involves three handshakes between the central system and the FNP to ship that line over. Negotiations to allocate a central system buffer, and negotiations to allocate a mailbox over which to negotiate the central system buffer are involved. This overhead is unacceptable, both in terms of number of interrupts, CPU time, and response time under load for highly interactive time sharing applications such as real-time editing. The DIA mailbox protocols were partly designed for NPS compatibility; these goals must be reexamined.

Other systems ship characters with process "destinations" at regular intervals (say 1/30 sec.) between front-end processor and central system in multiplexed buffers. Such a scheme for DIA transfer maintains the same throughput with complete character-at-a-time response.

There are other schemes to reduce the overhead of a DIA transaction, including the sending of short messages in the mailboxes as opposed to negotiated buffers. Other schemes involve the regular shipment to the FNP of the addresses of available buffers. The scheme proposed above is not as radical as it may first appear; it is not unlike the ring-zero demultiplexing currently being proposed for IBM 3270 support.

(1) Direct Interface Adapter, the communications path between the FNP and the central system.

IX. That's going to take some bite out of the system, isn't it?

Yes. You get what you pay for. If Multics is not responsive or fast enough to support this highly interactive application, we cannot truly claim we have a truly interactive time-sharing system. Current performance via the ARPA net on MIT lags three or four characters behind typed input. This, however, is going through the substantial overhead of the Network Control Program, the IMP dim, the packet-switched ARPA network, etc. Yet, these results are on MIT with "real users" logged in. It is felt that Multics response may not be as bad as some fear; if the TTY dim must be subjected to the same type of scrutiny and recoding that has been performed upon page control and the Disk Dim, then this is as good an excuse as any. People want functionality as well as performance for their computer dollars.

X. Why did you have to write it in Lisp?

The language Lisp is extremely attractive as a tool for the development of user-extensible, modular, efficient programs. It overcomes many of the disadvantages of block-structured languages such as PL/I, Algol, and Pascal.

In Lisp, I can make functions 3 or 4 lines long that I can call from any place in the entire subsystem with a call overhead of about six instructions, and a stack frame overhead of usually 0 to 4 words (on the Lisp stack, of course). In PL/I, for example, I am forced to make the choice between external and internal procedures. The call overhead for external procedures is unacceptable, being near two dozen instructions including the callout, entry, and return operators. The minimal 64-word stack frame is also unacceptable. For trivial functions, such as "go forward a character", the external procedure must be ruled out.

The internal procedure is similarly problem-laden. By definition, it may only be used in the procedure in which it is defined. This means that either its text must be duplicated, lexically and in object code, in all source modules that wish to use it, or one very large source module must be used. In either case, the current implementation causes the stack overhead of the internal procedure to be paid for even when it is not being called, adding to the frame of the procedure in which it lives. In the first case, modularity is sacrificed because everything must be recompiled or changed if one internal procedure changes. In the second case, maintainability is sacrificed, because one gigantic procedure must be recompiled for the most minimal change. Furthermore, internal procedures cannot be traced.

In neither case can user extensions call these internal procedures easily, or with any kind of transparency or

modularity.

The programmer efficiency of most block-structured languages also leaves a great deal to be desired. Most characters in a source program are syntactic constructs of the language, not user variables and functions. The weight of mandantory declaration bears heavy upon the smallest source module of most subsystems. Lisp object code suffers not one instruction for the lack of these declarations. Optional non-generic operators address the data-type issues adequately.

The Lisp environment provides a "process" not unlike the Multics process environment, where any piece of the subsystem may be called from any other. User code can call any function in the subsystem. (1) Unlike PL/I, this does not reduce the efficiency of the subsystem. The Lisp environment is finely tuned for allocation in well-defined increments, as this is one of the tenets of the language. As dynamic an application as a real-time editor relies heavily upon this. This path is more efficient than the standard area path.

A large Lisp subsystem can be debugged incrementally, as functions are added (either by the developer or a user extending it). Not only is recompilation/rebinding (worsened by the procedure call issues outlined above) not necessary, but the support of an interpreter is available for debugging. No other compilable language on Multics has such a facility.

Where function call is still too expensive, Lisp provides a macro facility, whose invocations look identical to function invocations. It is tremendously powerful, and useful. No other compilable language on Multics provides any macro facility.

Many of the traditional complaints about Lisp are founded in darkness. For example, be aware that production subsystems in Multics Lisp are not interpreted, but compiled, by one of the finest Lisp compilers available. The Lisp compiler (2) produces standard Multics object segments to run in the Lisp environment, offering a performance improvment of over 100 over interpreted code. Compiled and interpreted functions can call each other freely.

"Lots of Irritating Single Parentheses", as the acronym

---

(1) Whether some limitation on this is desirable is another issue.

(2) A particularly interesting program about which I have written an extensive document of probable interest to those with an interest in compiler theory. It is available upon request.

"Lisp" (1) has been accused of denoting, are a problem only when proper editing tools are not available. The parenthesis and S-expression balancers of EMACS (and Multics EMACS) reduce the grief of Lisp editing by a factor of 10. Once the hang of it has been acquired, it is found that the time to prepare a Lisp function (say in the editor) that works can be measured in seconds, while a comparable PL/I preparation might take 20 minutes, including all the declaring, compiling, binding, debugging etc.

The efficiency of character-handling is also not an issue. Multics EMACS contains a small amount of assembler (2) code for inserting and deleting characters from the active line. Note that PL/I cannot utilize an mrl instruction to open up a line, either. Calling a LAP program from Lisp still takes the same 6 instructions: PL/I still needs a dozen or more.

Lisp is not the optimal language for all tasks. Dealing with machine objects already laid-out in storage is not particularly efficient. The representation of data objects does not lend itself well to packing bits and characters, etc. The structure concept of COBOL and PL/I is superior for these applications. But in the editor, neither of these are issues.

XI. Does this mean that we have to support Lisp?

Yes and no. Support means a lot of things. We do not need to promote it as a product, or as an up-front user language at this time. These things can come in time.

We must install the Lisp environment support in whatever library this editor would be installed in. We must be responsible for fixing bugs if necessary, and making minor extensions as needed. These commitments do not take much personpower.

We do not need to distribute an "Orange Cover" Honeywell user document, although that option remains open to us. Documentation of Multics Lisp is available to us, in on-line form, and the creators of this documentation have offered it to us. This documentation is currently available internally, and can continue to be so.

We do not need to create marketing documentation at this time. We should train some more developers within the Multics software group at this time: a tutorial document prepared by me is available internally. It should not be necessary at

---

(1) Actually for "List Processing Language".

(2) Actually LAP, the Lisp Assembly Program, a type of ALM which produces Lisp-loadable object segments.

this time to train supporters outside the development group; this issue can be addressed later.

Some documentation must be distributed in order to enable potential writers of editor extensions to code them effectively in Lisp. This is not a large problem.

We feel that the time spent supporting Lisp in whatever capacity will buy itself back in an increased number of development tools. There are already some private tools in Lisp of some importance. Any support of Lisp would vastly augment their status.

The Multics Lisp implementation is an implementation of MACLISP, one of the more well-thought-out and complete implementations of Lisp available. It is completely operative, and has no known major bugs. Major extensions are not needed at this time. We can have it now.

XII. What about "editor macros"?

It has been observed by those who have addressed this problem before that editors which try to offer a single language for interactive editing and macro-writing develop a compromise which is inadequate at both. The qedx interface is not unreasonable for a line-oriented printing-terminal editor. The commands are natural, and simple, for that application. However, developing "macros" (complex pre-written functions) out of these functions is a feat of such magnitude that a former Multician pinned to his door a qedx macro so complex that it could actually play tic-tac-toe. There are no facilities for program control, variables, subroutines, etc., in this language. It is not a good programming language. Various attempts to add these features are ill-conceived, and do not make it any more so suited. The qedx language is still a human-to-editor imperative language by which to negotiate changes to a segment.

The other end of the spectrum is TECO, either on Multics or the PDP-10's where it was born. In an attempt to provide functions suitable for program-writing, such as loop constructs, character-at-a-time movement, etc., TECO requires someone editing a segment to devise little programs to do something as simple as "change all foo's to bar's". Teco is oriented towards program writing; the program-instructions are really too primitive to be used as commands in a line-oriented environment. On the other hand, the single-character names of the commands, a feature to allow use as an editing language, cause TECO programs to be notoriously abstruse. Thus, TECO fails at both.

It is the opinion of those who have previously addressed this issue that there should be two languages

associated with an editor; the keystroke language by which a user edits his or her segment, and the language for writing macros. This does not constrain the programming language to be terse, nor the keystroke language to be primitive. Thus, ITS EMACS was implemented via TECO, as a programming language, but with an newly designed editing language optimized entirely towards editing.

The Multics EMACS editing language is the provided list of control characters and escapes.

The Multics EMACS programming language is Lisp. Lisp is a complete and powerful programming language, with all of the features one would expect in a higher-level language. The Multics EMACS programming language is enriched by the functions provided in the editor environment. Many of these functions are the ones normally invoked in response to keystrokes, such as the function "backward-char" which is normally invoked when "control B" is typed. If you know what "control B" does, you know what invoking "backward-char" does. I contend in dead earnest, by experience, that a "macro" written in Lisp for Multics EMACS is more readily comprehended by one who does not know lisp than a comparable macro in TECO or qedx would be by one versed in these languages. So transparent is this Lisp code. This is largely due to the lack of syntactic overhead in Lisp, and the aforementioned Lisp macro facility.

I have devised and debugged Multics EMACS editor extensions (a better term than "macros") in real-time, by typing them into a buffer, staring at it, saying "OK, editor, now load it" and now "try it". If it doesn't work, your are still looking at it. Edit it, try it again. You never leave the editor. You see the function and its results in front of you. It is quite effective.

Via this technology I have devised esoteric functionality such as hitting a control-character which fills in the declaration for the PL/I subroutine name I just typed, and another which allows me to "edit" the listing of a directory, deleting the segment whose name I have pointed the cursor at, and editing the display in parallel.

Any user-function, or any editor-provided function, can be hooked up to any key, or invoked explicitly by its name.

I fail to see why any person who can write an editor macro in qedx or TECO would have problems, given a well-documented starting point, writing editor extensions in Lisp. Knowledge of the internal organization of the editor is not necessary. What is more, the display is managed automatically by the editor. User extensions (or builtin functions, for that matter) are not aware of the existence of the display; whatever transformations upon the buffer were performed by the user code

or the editor code, these changes will be made automatically to the screen when they are all done, via the technology known as "redisplay". All in a terminal-independent fashion.

XIII. What about "The Editor"? Is this instead of the proposed qedx extensions?

If you have a display terminal, yes. I cannot envision any reason why anyone would want to use qedx or any of its derivatives given the capability for real-time video editing.

If you don't, probably not. Although EMACS does work on printing terminals, I would prefer qedx, and given that, the current research into conventional editing is well worth it. The only advantage to using EMACS on a printing terminal is the large number of user extensions that people will have developed. ITS EMACS moves the printing-head around instead of the cursor, and it is curious to watch.

XIV. Where can I see this thing?

In Cambridge, it is best demonstrated on the CISL Delta-Data 4000 during CISL service, 1:00-3:15 PM. If you have any kind of display terminal and ARPANET access, you can probably use it on the MIT machine. Contact me. (HVN 261-9330, or 617-492-9330, or Greenberg.Multics on MIT, CISL, or System M, or Greenberg @ MIT-MULTICS (ARPANET), or BSG @ MIT-AI or MIT-MC, or care of Honeywell Information Systems, 575 Technology Square, Cambridge, Mass., 02139.)

I hope to have it operative on the VIP 7200 in Phoenix, System M, in the very near future.

Current repertoire

The following is a command list that I hand out. It is nowhere near complete, in terms of what I plan. It is here simply to give an idea of what some types of possible things are. This is the current repertoire of the implmentation.

The Multics real-time editor is an interactive, display-oriented text-preparation and editing facility designed after the EMACS TECO macros of R. Stallman and others on the MIT-AI PDP-10.

This is an elementary list of editor commands. Almost all printing characters represent themselves and go in as text; Carriage-return or linefeed may be used to terminate lines. The symbol "^" means "control" - "^A" means depress the "a" key while the "CTRL" key of the terminal is depressed. The symbol ESC means that the key labelled "ESCAPE" or "ALTMODE" should be depressed and released prior to pressing the next key. I.e., ESC-D means the two-character sequence, ESCAPE D. (Lower case can be used, i.e., ESC-d).

The cursor is considered to be the LEFT edge of the blinking cursor, i.e., when in the leftmost column, it is considered to be before the first character.

^A Go (move cursor) to beginning of line.
^B Go Back one character.
^C No-op. See ^L.
^D Delete character at (to right of) cursor.
^E Go to end of line.
^F Go Forward one character.
^H This is a backspace- do NOT USE.
^I Same as a TAB.
^J Same as a linefeed.
^K Kill to end of line, except when already at end of line, delete the linefeed (merge lines).
With numeric arg (see below), kills that many lines.
^L Redisplay the screen. Use if terminal or editor starts losing, or somebody sends you a message, etc. On FNP Multics, follow this by ^C because FNP will not transmit ^L alone.
^M Same as carriage return.
^N Go to next line, same horizontal place.
^O Open up space, insert a linefeed, move cursor back. See ^U. ^U^U^O for instance will open up 16 lines.
^P Go to previous line, same place.
^Q Quote the next character, i.e. insert it literally, as ^Q# to get a pound sign in. Same as \.
^R Reverse search. Leave cursor positioned before matching string, don't move cursor if not found. See ^S.
^S Search. Will prompt at bottom of screen for search string. End the search string with ESC.
^T Twiddle (transpose, interchange) the last two characters typed, like, I like Mut^Tics becua^Tse... etc.
^U Multiplier. When not followed by a number, multiplies the next command by 4 for each use. I.e., ^U^D deletes 4 chars. ^U^U^D

	deletes 16. With a number, uses that, i.e., ^U13x inserts 13 x's.
^V	not implemented yet.
^W	Wipe (kill) all text between cursor and the-mark. Can be retreived with ^Y.
^X	Control X commands are two-character sequences, listed below.
^Y	Yank (retrieve) killed text to cursor. Unkills last killed word, line, or region (^W). With an argument, goes that many killings down a 10-position ring-buffer of old killings.
^Z	does a Multics Quit.
^@	Sets the-mark to be where the cursor is now.
\	Causes the next character to be inserted literally.
@	Kills all the text on the current line.
#	Deletes the previous character (before the cursor, which is usually the last character typed, like in normal Multics.)
ESC-[	Go to beginning of paragraph, right before first word.
ESC-]	Go to end of paragraph, end of last line. Runoff control lines count as paragraphs.
ESC-<	Go to beginning of buffer.
ESC->	Go to end of buffer.
ESC-B	Go backward one word, leave cursor before first character of it.
ESC-C	Capitalize the current or last word, move to after it.
ESC-D	Delete the word to the right of cursor, and all whitespace between cursor and it.
ESC-F	Go forward one word, leave cursor after last character of it.
ESC-G	Do to point/mark region what ESC-Q does to a paragraph.
ESC-H	Set point and mark around the current paragraph.
ESC-L	Lowercase the current or last word, move to after it.
ESC-Q	"Fill" the current paragraph, like runoff with ".na". With argument (i.e., ESC-1 ESC-Q), fill and adjust like runoff with .fi and .ad. See ^XF to set fill-column.
ESC-U	Uppercase the current or last word, move to after it.
ESC-W	Like ^w, but doesn't wipe, just puts in kill ring.
ESC-X	Prompt for the name (and args) of an extended command. See below.
ESC-Y	I don't like what I just ^Yanked. Get rid of it and yank the previous thing in its place.
ESC-\	Delete all whitespace surrounding cursor on current line.
ESC-#	Delete the word to the left of the cursor, and all whitespace between it and cursor.
ESC-^A	Go to beginning of Lisp function, i.e., last line with open paren in first column.
ESC-^F	Skip over exactly one balanced S-expression, including parenthesized lists. This is VERY powerful, and may be used to balance parentheses in PL/I or anything else. Diagnoses missing ")"'s.
ESC-^N	Skip to end of current Lisp list. Used internally by ESC-^F.
ESC-^O	Break line at this point, indent new line like this line.
ESC-ESC	Prompt for string for Lisp to evaluate. For hackers only.

ESC-(number)

Causes the next command, if one of the following, to be repeated that many times. I.e., ESC-35-^D deletes 35 characters. ESC-4-m inserts 4 m's. Same as ^U when used in this way.

Good for ^B ^D ^F # ^N ^P ^O ESC-B ESC-F ESC-D ESC-#

Also tells ^Y how many back to yank.
Also tells ^K how many lines to kill.
Also tells ESC-Q whether to adjust or not.

^X commands:

^X. Set "fill prefix" to what's between beginning of line and cursor. The "fill prefix" is inserted automatically by linefeed, ESC-Q/ESC-G, and autofill. It is also prepended to lines that are yanked, if they follow yanked linefeeds.

^X^R Read a file into buffer, leave you at first position of first line. Prompts for file name. Terminate file name with CR or LF.

^X^W Write out buffer to file. Prompts for file name. Terminate with CR or LF.

^XB Switch to new buffer, or old one. Prompts for buffer name, which is terminated by CR or LF. Buffer name shows at bottom of screen.

^XF Set "fill" column for ESC-Q and speedtype/autofill stuff to horizontal position where cursor is now.

^X^B Shows listing of buffers and pathnames. ^XB somewhere else to get back to what you were doing.

^X^L Lowercase all letters between cursor and the-mark.

^X^S Write out buffer to last file read or written in this buffer.

^X^X Exchange the cursor and the-mark, to verify what you are getting into before typing ^W.

^X^U Uppercase all letters between cursor and the-mark.

^X^M Prompt for a Multics command line. Terminate with CR or LF. Multics commands that produce output may well screw up your display, may have to ^L.

You may edit in the minibuffer (the bottom of screen prompting area), ^Y'ing things from elsewhere, etc. If you try multiple lines, you only see 1 line at a time. Use ^QCR or ^QLF to get linefeeds in when linefeed is the terminating character.

Extended commands - invoke by ESC-X. Type command and args, if any, in minibuffer, where it puts you. Terminate by CR or LF.

replace Global substitute. Will prompt for two strings, terminated by ESC. Replaces all occurences of first string by second, leaves you after last occurence.

speedtype Enter speedtype (word abbreviation) mode.

setab Set a speedtype word abbrev. E.g.,

```
setab bsg Greenberg
```

Can take multiple pairs of args, if that's convenient.

fillon Set auto-fill mode. Speedtype mode sets this automatically.
filloff Turn off speedtype and autofill modes.
put Takes one arg, a "variable" name. Same as ^W, but puts text in that "variable" instead of on the kill stack. Use getback to get it back.
getback Takes one arg. Like ^Y, but yanks back the named "variable" which is its arg.
lvars List names and lengths of all variables ever "put".
quit Exit the editor.

The editor accepts META characters from AI TV's, and does TELNET break/interrupt processing.

This document, of course, was prepared with what it describes.