Date: 3 March 1980

From: Bernard S. Greenberg

Subject: Emacs Paper for Honeywell Conference

To: MTB Distribution

Attached is a reproduction of my paper, "Multics Emacs, an Experiment in Computer Interaction," which I will present at the Fourth Annual Honeywell International Software Conference, in Minneapolis, on March 25.

Bernard S. Greenberg
Honeywell Information Systems, Inc.
Cambridge Information Systems Laboratory, MSD/LISD
575 Technology Sq. (Mail Sta. MA22)
Cambridge, MA, 02139
HVN 261-9330

Overview

Multics Emacs is a video oriented text preparation and editing facility being released as a product in Multics Release 8.0 in early 1980. Multics Emacs features the ease of use of stand-alone word processing coupled with the power of the full Multics program environment. Multics Emacs is coded in the Lisp language, and is the first released Honeywell software coded in Lisp. The use of Lisp has provided an extensibility which has nourished the development of a wide variety of features which have brought Multics Emacs far beyond its original goals.

Multics Emacs marks the entry of Multics into the arena of video oriented user interfaces. While its original conception was as a text editor, it has grown into an entire user environment: an embedded mail system, interactive message system, features for compiling and debugging programs with automatic aid, and other features, have since brought Multics Emacs out of the domain of text editors and into the domain of comprehensive paradigms for user interaction. Valid questions have been brought to the forefront about the exact roles of "editors", "editing features", "buffers", and so forth in an integrated user environment. These issues will be dealt with in more detail later on. What is more, Multics Emacs has provided a starting point for other current research on alternative video oriented interaction scenarios on Multics.

Multics Emacs is a member of a class of what we have designated as "mainframe video editing systems": those which run on mainframe computer systems, usually general purpose, medium or large scale computer utilities, yet interact in a very tightly coupled loop with the user, interacting on every character typed, and maintaining on a screen a model of text being edited. This style of text editing is usually associated with stand-alone "word processing" systems, which optimally combine the cost effectiveness of microcomputer technology and video display devices. However, stand-alone "word processing" systems can be no more than what they are; the power of a total, integrated computer utility is absent, as is any possibility of application of the video interaction paradigm to any problem except text preparation.

Mainframe video editing systems, running on multi-user computer utilities, pose a problem to system designers, insofar as the system overhead required to interact so tightly with the user is usually prohibitive. This almost always reduces the cost effectiveness of such systems far below that of stand-alone

word processing for comparable tasks. However, one large portion of the motivation for these mainframe systems is precisely the large set of tasks and design goals to which stand-alone word processing is completely unsuited; this reduces the number of conclusions to be drawn from such comparisons.

The mainframe implementation of Multics Emacs has been directly responsible for the realization of its development cycle; The implementation on the Multics system, running on the Honeywell Level 68, using a high-level language, has allowed trial implementation and testing of features on an incremental basis. Given a running Emacs environment, an Emacs developer can create and debug new function from within the environment, while using all the features of the environment. Since the inception of the subsystem, all extensions of all sizes have been developed in this way. This ability is the deliberate result of the choice of Lisp as the implementation language; this will be discussed in more detail below.

Relevant History of Real-time Editing at MIT

Multics Emacs evolved from a line of mainframe editors which acquired video capability as an evolutionary step: editing features in display terminals were not a model for these editors; stand-alone word processing systems had not yet appeared.

The starting point for this editor family was the TECO editor on the ITS operating system at the MIT Artificial Intelligence Laboratory. TECO (for Text Editor and Corrector) is a mainframe editor which maintains buffers, (if there are many, one is selected at a given time) containing files being edited, and a virtual pointer to a given character position in the selected buffer. The TECO user types a string of "commands" at TECO: typically, these commands move the virtual pointer, add, delete, and display text around it, and so forth.

Two other features of TECO are relevant to the history of Emacs. Teco allows commands to be grouped into macros by the user, which may be stored and invoked by name. In this way, the TECO user can build libraries of his or her own commands, and through successive levels of subroutinization, large and powerful extensions. The other relevant feature of TECO is a vast and powerful set of control and data primitives designed to facilitate such programming; recursion, hashing, non-local control transfer, conditions, iteration, and a variety of other higher-level constructs are available. Stallman [Stallman] discusses some of these features in depth.

As video display usage became widespread at the MIT A.I. lab, TECO acquired a feature whereby it could divide a video screen into two regions. User interaction (the typing of commands, responses of TECO, etc.) appeared in the lower window (delimited screen region). The upper window would be used to show a visual representation of the text in the buffer around the virtual pointer (or "point"). Every time the user completed

an interaction, TECO updated the upper window, modifying the text in the window to show the modified state of the text in the buffer. This operation is known as redisplay. This notion of a continuously, automatically updated model of text on a screen is the central characteristic of the video editor.

The next development was the onset of "real-time editing". A feature ("Control R Mode") was added to ITS TECO which, when activated, placed the video terminal's cursor in the text display window, at the point in the display corresponding to the virtual pointer ("point") in the buffer. In this mode, single characters were read from the keyboard, in character-at-a-time fashion. "Text" characters (ordinary printing characters) were interpreted as requests to place themselves in the buffer at the current point ("self inserting"): thus, text was inserted at the current virtual pointer simply by typing it, with no need for an "insert text" command. ASCII control characters, and combinations of ASCII control characters with other characters, were interpreted as requests to invoke editor commands. The "connection" or binding between an ASCII control character and TECO editing command was chosen for mnemonicity, e.g., "Control D" to Delete the character at the virtual point. After each typed character (be it text or command, or more precisely, when no more input was buffered), a redisplay was performed to update the image of the buffer as well as update the position of the terminal's cursor to correspond to the virtual point.

The net visible effect of "Control R mode TECO" was much like today's stand-alone word processing, or "terminal editing", with the crucial difference that a tremendously powerful mainframe editor was involved. The illusion of "editing the text on the screen by typing characters" is common to all three kinds of editing. This has since become the standard paradigm for use of a video screen as an aid for text creation and editing. Those who have attempted to teach the use of computers for editing to the computer-naive have universally found this paradigm simpler and more readily grasped than that of the classic time sharing editor based line editors. At this time, (early 1970's) similar systems (TVEDIT, E, etc. [EDOC]) had also appeared at Stanford University and other places.

The final prehistoric evolutionary step of Emacs was a TECO feature whereby arbitrary macros could be assigned to keys in Control R mode. A user could then construct his or her own commands of unlimited power or sophistication, and have them invoked by a single keystroke in Control R mode. Thus, a command to "move the current virtual point to the end of a sentence" could be coded as a TECO macro, and associated with a key, which when pressed in Control R mode, would appear to act as a "key which moves the cursor to the end of a sentence."

This development led to a proliferation of packages of macros intended for use in Control R mode at MIT AI in the mid 1970's. Each of these packages contained a large repertoire of useful function, featuring knowledge of many common text constructs, including many used in programming languages,

presenting an integrated interface for invoking these sequences from keystrokes. Of these packages, one known as EMACS (for Editing Macros) achieved dominance. Richard Stallman [Stallman], the chief developer of EMACS, details more of this history, and how EMACS differs from these earlier packages.

A central feature of the philosophy embedded in EMACS was that of editor construction by extension. The interfaces and keystroke commands provided by EMACS (on ITS) form a unified whole, documented and presented as an editor, not "a collection of macros to be used in Control R mode". The user of EMACS is unwaware of the existence of the underlying TECO. Similarly, EMACS encourages the construction of further packages by adding levels, using the facilities (functions, protocols, etc.) provided both in EMACS and natively in TECO.

[In keeping with Multics and ITS usage, we use "EMACS" to designate the ITS editor, "Multics Emacs" to designate the Multics editor in specific, and "Emacs" to designate the Multics editor when its differences from the ITS editor are not relevant, or either subsystem when distinctions are not relevant.]

The notion of extension is a critical one: the ability for both users and implementors (the distinction here deliberately blurs and vanishes) to add new levels of function and thus build either "larger editors" or major facilities (major modes) within EMACS is the most visible single distinguishing feature of EMACS. Typical extensions create specially tuned sub-editors oriented towards (for example) editing Lisp programs, editing PL/I programs, preparing English text, etc.

The concept of sub-editors tuned for programming language editing is significant. Many languages have syntactic constructs which are difficult to deal with without automatic help: the balancing of parentheses in Lisp is a classic case in point. Here, and in an increasing number of cases, the difference between a language or set of language features being usable or not is made by the existence of an editor with special features for that language.

Inception of Emacs on Multics

In early 1978, Multics' text preparation facility consisted of two text justifiers, one being phased in and the other being phased out, some powerful dictionary tools, a batch-mode abbreviation expander, and two editors. Both editors were half duplex, line-at-a-time, printing-terminal oriented editors in the classic time sharing mold. One was an unmodified reimplementation of the EDL/EDA editor interface of CTSS, intended for the most naive users. The other, the Multics standard editor, was a stripped down version of Bell's QED editor [CG40], a venerable warhorse which had been used to enter and modify all of the Multics system for years. A version of TECO implemented by MIT also existed, but was very weak (compared to ITS TECO), was also half duplex and printing-terminal oriented, and had never acquired a large following.

By early 1978, almost all Multics programmers and users were using the Multics standard editor, or a greatly augmented private version thereof, which had acquired a large number of popular features without altering its basic design. At this time, the author, in preparation for an annual lecture series at MIT, encountered EMACS on ITS, and immediately began contemplating what it would take to implement such a subsystem on Multics. It was then clear to the author that the next step in Multics editor development, which had been stagnating, would not be evolutionary, but revolutionary. EMACS provided a well debugged model, which had evolved through substantial design iteration.

At once, the problem of lacking character-at-a-time I/O on Multics had to be overcome. Since its inception, Multics has historically interacted on a line-at-a-time basis; the preponderance of half duplex printing terminals for Multics' first decade is largely responsible for this orientation. The front-end/mainframe protocols are organized for line transmissions. Outside of Emacs, Multics provides no form of display support.

As soon as experimentation with EMACS-like concepts was desired, character-at-a-time I/O was effected on an experimental basis by a patch to the front-end software. This patch sufficed for many months; yet, the overhead implied by this mode of transmission (and the implementation in terms of this patch) was a cause for concern in many quarters. Those who had worked hard towards Multics performance goals were alarmed at the prospect of a subsystem that interacted on every character. The patch existed for many months on the development site at Honeywell's Cambridge Information Systems Lab; experimentation only spread to the MIT Multics site by use of the character-at-a-time support implicit in the ARPANET and the Multics ARPANET implementation.

The second problem in a trial implementation was the choice of a programming language. Historically, all Multics programs have been written in PL/I. Multics PL/I [AG94] is one of the fullest implementations of the ANSI PL/I standard extant, and has evolved over the years as the sole system support language implementation for Multics. Its object code efficiency, robustness, and maintenance are superlative. Thus, PL/I seemed to be the natural choice. Being able to view the ITS experience in perspective, it seemed as though marked efficiency could be gained by implementing an EMACS-like editor directly in PL/I, as opposed to as a system of macros in some other language (viz., TECO), and avoid the interpretive overhead of that latter language. Stallman [Stallman], in retrospect, speaks of the deficiencies of TECO as an implementation language as well.

However, one of the chief lessons of the ITS experience was the value of extensibility: EMACS as an environment in which editing subsystems can and ought be created by user/implementors. The power to grow is the greatest power of all: A direct implementation of an EMACS-like interface, no matter in what language it was realized, would have to have modular, simple low- and medium-level interfaces for utilization by user code.

Various scenarios for extensibility in a PL/I-based implementation were evaluated. The Multics process environment is one of the classic models of extensibility in the literature, and it is PL/I-based. The ability to extend and customize one's Multics process environment via PL/I subroutine call and definition has provided the model for many operating systems since. Yet, several features of PL/I pointed away from its choice as the Multics Emacs implementation language. Given that any reasonable implementation of an EMACS-like modularity would associate editor primitives (e.g., "move the virtual pointer forward a character", "delete the current character", etc.) with PL/I subroutines, extension code would degenerate into a sequence of subroutine calls. Calls between separately compiled modules are expensive. Calls to internal subroutines are cheap, but by definition, such subroutines are not accessible to other modules. Thus, if externally accessible (i.e., usable by extension) procedures were to be had, they would have to be of the (expensive) external kind, which would add substantial overhead to even the smallest editor primitive. Furthermore, PL/I is notorious for requiring declaration of the smallest artifacts of every module; all variables used, all external names, etc. Programs consisting of hundreds of lines of declaration and ten lines of code are not uncommon. It seemed as if people were going to write extensions, the overhead of declaring each editor primitive to be used and its parameters, as well as every global variable and its data type and precision would stand squarely in the way.

This led to the choice of Lisp as an implementation language. In Lisp, there is no external/internal subroutine distinction. All functions (the Lisp procedural abstraction) are coequal. Lisp inter-function calls all have the same

more than a PL/I internal call. Lisp calls are traditionally very cheap. Lisp programs are traditionally written with many small (i.e., ten or fifteen line) functions, which therefore use inter-function call very heavily. Thus, function calling has been highly optimized in Lisp, and much of the overhead associated with PL/I calling, e.g., setting up a control frame thread for the benefit of the PL/I signalling mechanism, is not present. What is more, every Lisp function in a given environment may be accessed by any other function, unless very special measures are taken, and similarly for every global variable. Of course, this can be a mixed blessing, in terms of both programming style and the pitfalls of a global namespace.

Lisp's notion of data abstraction also seems more well suited to subsystem building. In Lisp, one can define a "data type" by program convention only, without "informing the language" in any way. For instance, Multics Emacs defines editor buffer pointers (or "marks", conceptually inter-character pointers to text, dynamically updated as text is added and deleted) out of Lisp list nodes. Lisp programs in Emacs can pass around marks, either to each other or to primitives which manipulate marks, or store marks, without any knowledge, or even a declaration, of the internal structure or implementation of a mark. Here, Lisp fosters an isolation of levels of the implementation, which is highly desirable, and extends to within the internal levels of Multics Emacs itself.

Another very powerful feature of Lisp, specifically of MacLisp [Moon], the dialect in use on Multics, is the macro feature of the language, via which the syntax of the language itself can be extended. The "macro language" of Lisp is Lisp itself; Lisp programs are represented (at compile time) by Lisp data in a "public" representation. Lisp allows programmers to specify code to run at compile time to implement a macro-defined language; this is possible because of the Lisp data representation of Lisp programs, which allows the compiler itself to be a Lisp program. This in turn allows construction of highly specialized languages built out of Lisp: the Multics Emacs extension language is one such, and is expounded on at length in the Appendix. The success of the extension language as the vehicle for Emacs extension is remarkable, and a testament to the power of the Lisp macro facility.

Multics MacLisp has a fully mature debugging system, I/O facilities, and the ability to interface to other facilities in Multics. Multics MacLisp also has a powerful compiler, and all "production" programs are compiled (although the existence of the Lisp interpreter is invaluable during debugging). Many other Lisp systems lack these features, and are thus ill suited to development of production software.

The efficiency of Lisp is also raised when considering Lisp as a serious contender for a systems implementation language. The existence of the compiler ends all efficiency arguments about Lisp being "an intepreted language". The need to allocate

storage and garbage collect is often raised as well; sagacious storage management policies, which ought be used in any program in any language, put this "problem" well within limits. Even though traditional programming style in textbook presentations of Lisp often consumes storage in a liberally wasteful fashion, it is possible, with minimal added difficulty, to code in a fashion which is not wasteful of storage. Part of the problem here can be traced to what the author considers gross philosophical flaws in the classical presentation of Lisp. (For a presentation of the alternative view, see [LispNotes]).

A very closely related effort to Multics Emacs was the Lisp Machine at the MIT Artificial Intelligence Lab [Chineual], which has been under development during the entire history of Multics Emacs. All software on the Lisp Machine is coded in Lisp, including all parts of the operating system, the user utility programs, and most notably, the editor, ZWEI [Weinreb], which is EMACS-like. There was substantial design crosscurrent between Multics Emacs and ZWEI during the simultaneous development of both; the common features of these Lisp-coded EMACS-like editors were of great interest to both developers. The Lisp machine as a whole provided many models of Lisp-coded, full fledged, interactive user environments.

One of the unplanned benefits of Lisp which has proved to be of inestimable value is the ability to develop Emacs extensions from within Emacs: the ability to write Lisp functions, one by one, in an Emacs buffer in Lisp Mode (a sub-editor suited to editing Lisp programs) and test and debug them by observing their effect on the invocation of Emacs which is editing them. This paradigm has been directly responsible for the large growth of Multics Emacs extensions.

Communications Efficiency

In order to reduce the overhead associated with very tightly coupled user interaction (and the associated problem of response) in a multi-user computer utility, implementors of mainframe video editing systems have devised various techniques and communications strategies, whose general import are usually to move processing of typed characters further and further down the levels of the operating system and communications software. The further down such processing is moved, the fewer levels of software must be invoked to respond to each typed character. Such techniques involve increasingly complex data management and synchronization protocols between levels and nodes of communications software the further down into the operating system they are moved.

The technique used in Multics Emacs to reduce the character-at-a-time expense is called "negotiated echo". It is a scheme which optimizes the handling of the most common case of interaction, namely, the insertion of a printing character into the buffer at the end of a text line and its subsequent appearance on the screen, in response to its being typed by the user. When prerequisite conditions are met, the Multics

that negotiated echo can begin: the front-end will then echo (retransmit to the screen) all printing characters typed by the user until an "end condition" is met. Characters so echoed appear on the screen as typed, just as if a redisplay had occurred after each was entered into the buffer. When an "end condition", such as reaching the end of a line, or the typing of a non-echoable character occurs, all characters are sent to the mainframe and negotiated echo stops.

When negotiated echo stops, characters are shipped to the mainframe in character-at-a-time fashion as they arrive. Only when the mainframe requests to reinitiate negotiated echo, and no characters are in transit, does negotiated echo resume. To determine whether or not characters are in transit, both communications processor and mainframe keep a count of processed characters since the last front-end echoed character. The request to restart negotiated echo includes the value of this count as perceived by the mainframe. As long as Emacs is in the state of having its virtual pointer at the end of a line (with some other constraints not mentioned here) requests will be made for characters via negotiated echo as opposed to raw characters. Thus, if resynchronization fails, repeated attempts will be made to resynchronize until no characters are in transit.

The echo negotiation protocol is viewed as a three level hierarchy; Emacs, the Multics mainframe communications software, and the front end software are each prepared to echo characters. Each requests the next level to produce characters, some leading prefix of which may have been echoed by that or lower levels. Each level reprocesses those that have not been echoed, and echoes the leading prefix thereof which is echoable. This architecture allows for multiple types of communications processor, some of which may not support negotiated echo. It also allows for arbitrary cessation of negotiated echo at any level for reasons unknown to the higher levels (for instance, running out of buffer space to hold echoed untransmitted characters).

The resynchronization technique described above succeeds only when the network delay between front end and mainframe is not on the average longer than the mean inter-character time of the typist. For multi-node networks with long packet delays, this resynchronization technique will not work.

Further features of the echo negotiation protocol include the ability for the mainframe to deterministically stop negotiated echo in progress, and the ability to dynamically redefine which characters are considered "non-echoable". An example of the use of the former feature is the interactive message facility, which aborts echo-negotiated input upon arrival of messages. An example of the use of the latter feature is the semicolon character in a sub-mode of the PL/I program editing mode: semicolon is the PL/I end of statement character, and in this mode, typing it triggers an automatic position to the indented beginning of the next line.

Echo negotiation has achieved its goal of reducing the mainframe interrupt and wakeup overhead of Emacs use to well within manageable limits. The vast majority of interaction with Emacs consists of entering text at the end of a line, whether it be new documents, programs, or even long-named editor requests. As long as the system is not overloaded, it has the added benefit of causing character echo to be instantaneous. When the system load increases, resynchronization takes correspondingly longer, and characters default to being processed in increasing numbers by Emacs as opposed to the communications software. This is seen by the user as decreased response.

The Place of Emacs in Multics

Multics Emacs has achieved wide popularity at the two Multics exposure sites, at MIT, and at Honeywell's Large Information Systems division in Phoenix, Arizona. About one hundred people use it regularly. Those with video terminals rarely revert to any other form of editing once having seen Emacs. The truly naive as well as the sophisticated master the user interface in short order, paralleling the ITS experience. Due to various economic situations, the extra expense resulting from the machine overhead of this form of editing seems to be no deterrent. The availability of video terminals is currently the controlling factor of Emacs use at these two sites.

When Emacs users invoke Emacs, they tend to interact with it for a long time, editing many files at length. Since interacting with Emacs is vastly different from interacting with other Multics facilities, the user becomes acclimated to the Emacs mode of interaction: this tends to prolong the user's stay in Emacs. Since starting up an invocation of Emacs is a slow and expensive process, there is added incentive to stay "inside" Emacs as long as possible. Toward this end, a number of "modes" have been created which parallel existing function in Multics, but operate within the Emacs environment. These features always utilize Emacs screen management and editing capabilities implicitly, and are often more attractive and powerful than the native Multics facilities when a video terminal is in use. This parallel function has rightly generated some controversy.

Typical of this is the Emacs mail system, which places incoming and outgoing mail in buffers and windows, to facilitate real-time editing of the mail, paging through mail while reading it or responding to it, and automatically generating replies. There exists a complete and integrated Multics mail system, outside of Emacs, and many have validly raised the point that the existence of another one, inside Emacs, nowhere as complete, is questionable at best.

However, the task of mail composition seems to overlap so largely with the task of text editing, that integration with a text editor seems appealing. In the standard Multics mail system, a sharp distinction is made between "inputting" mail and "editing" mail. The use of multiple windows to read and reply to mail, with the ability to page back and forth, is so natural

Echo negotiation has achieved its goal of reducing the mainframe interrupt and wakeup overhead of Emacs use to well within manageable limits. The vast majority of interaction with Emacs consists of entering text at the end of a line, whether it be new documents, programs, or even long-named editor requests. As long as the system is not overloaded, it has the added benefit of causing character echo to be instantaneous. When the system load increases, resynchronization takes correspondingly longer, and characters default to being processed in increasing numbers by Emacs as opposed to the communications software. This is seen by the user as decreased response.

The Place of Emacs in Multics

Multics Emacs has achieved wide popularity at the two Multics exposure sites, at MIT, and at Honeywell's Large Information Systems division in Phoenix, Arizona. About one hundred people use it regularly. Those with video terminals rarely revert to any other form of editing once having seen Emacs. The truly naive as well as the sophisticated master the user interface in short order, paralleling the ITS experience. Due to various economic situations, the extra expense resulting from the machine overhead of this form of editing seems to be no deterrent. The availability of video terminals is currently the controlling factor of Emacs use at these two sites.

When Emacs users invoke Emacs, they tend to interact with it for a long time, editing many files at length. Since interacting with Emacs is vastly different from interacting with other Multics facilities, the user becomes acclimated to the Emacs mode of interaction: this tends to prolong the user's stay in Emacs. Since starting up an invocation of Emacs is a slow and expensive process, there is added incentive to stay "inside" Emacs as long as possible. Toward this end, a number of "modes" have been created which parallel existing function in Multics, but operate within the Emacs environment. These features always utilize Emacs screen management and editing capabilities implicitly, and are often more attractive and powerful than the native Multics facilities when a video terminal is in use. This parallel function has rightly generated some controversy.

Typical of this is the Emacs mail system, which places incoming and outgoing mail in buffers and windows, to facilitate real-time editing of the mail, paging through mail while reading it or responding to it, and automatically generating replies. There exists a complete and integrated Multics mail system, outside of Emacs, and many have validly raised the point that the existence of another one, inside Emacs, nowhere as complete, is questionable at best.

However, the task of mail composition seems to overlap so largely with the task of text editing, that integration with a text editor seems appealing. In the standard Multics mail system, a sharp distinction is made between "inputting" mail and "editing" mail. The use of multiple windows to read and reply to mail, with the ability to page back and forth, is so natural

that some have wanted to learn to use Emacs for this reason alone. Certainly, if Multics had integrated video management (which is at this time under serious design consideration), the mail system could use it (and will) to advantage: indeed, the Emacs mail system is indeed a way of getting "video managed mail" if nothing else. However, the large percentage of the mail composing/reading task which is editing mandates that the most potent editing technology available be used, and this is Emacs. Emacs seems a more likely candidate to contain a mail system than the mail system to contain an Emacs, so this is the way it was done. (On ITS, an Emacs-embedded mail system exists as well).

The unique nature of the Multics process environment, specifically, the ability to call any procedure or subsystem known to Multics, if proper interfaces exist, allow a wide panorama of function to be subsumed into Emacs, and experimentation with video interfaces to Multics function to be performed. Creating function via the Emacs extension language and calling of external Multics routines begets utility without having to build an environment from ground up, and buys video management for free (by virtue of the automatic redisplay).

Prototypical of many "special purpose" Emacs modes is the directory editor, "DIRED", which exists in both Emacs implementations. The user, inside Emacs, invokes the directory editor via a sequence of command characters. A display listing all files in the storage system directory to be "edited" is placed in a buffer (and thus, by virtue of the redisplay, displayed on the screen). Normal Emacs commands can be used to position to any line (each line describes one file of the directory) of the display. In this mode, no commands which would cause the buffer to be modified may be issued. However, commands such as "delete this file" and "show me the contents of this file" are available as keystrokes. Thus, the user moves the cursor around the display of the directory listing, examines files, and marks them for deletion (they are actually deleted when the directory editor is exited). The user never has to specify file names, and sees a "large picture" of what files are in the directory at all times. This "menu"-type interface is typical of many advanced video systems [PARC].

Another class of special purpose modes invokes large scale Multics subsystems from within Emacs, and processes their output in a useful way. In Lisp mode, a single keystroke invokes the Multics Lisp compiler upon the function at which the cursor is pointing, and upon its completion, incorporates the object program into the running MacLisp environment, and displays the compiler's diagnostics on the screen. In PL/I and FORTRAN modes, the same keystroke invokes the appropriate compiler upon the source program being edited. The compiler's diagnostics are placed in a buffer in a second window, and are analyzed to identify the source lines flagged as in error by the compiler. A dedicated command (keystroke) in PL/I and FORTRAN modes moves the cursor in the source program to the "next line flagged as in error by the compiler", and moves the "current point" in the

diagnostics buffer to the next diagnostic, such that it is displayed in the diagnostics window automatically. The "pointers" to the source lines are kept as "marks", and are thus valid through the new editing of the source program. In this way, multiple windows are used to "reply to" source program errors in the same way that mail responses are generated.

An interactive message processor in Multics Emacs creates buffers associated with senders of interactive messages. These buffers have their key bindings so set up that typing lines into them will send those lines as interactive messages to the associated user. As messages sent by the other user appear at the end of the buffer automatically, a "conversation" can be held with another user simply by "going to" such a "message buffer". Multiple message buffers (like any other buffers) can be displayed on the screen in multiple windows, and thus several conversations can sometimes be seen scrolling simultaneously, automatically, on a Multics Emacs screen. This facility even allows automatic routing and response to interactive messages coming from foreign network sites.

The "ultimate" Emacs mode, in some sense, is one now under development, called "Multics Mode". In this mode, the full flexibility of the Multics User I/O system is exercised to connect Multics Virtual I/O streams [AG91] to Emacs buffers. The net effect of this connection is to "run a Multics process from inside an Emacs buffer". The user stays in Emacs for the life of the process. Carriage return submits a line of the buffer to the Multics command processor; output produced by Multics appears in the buffer as it occurs. When the Multics I/O system requests input, the user types into the buffer and a carriage return transmits the (possibly edited) line back to the I/O system. Thus, Emacs editing becomes applicable to all Multics interaction, including searching, scrolling back through previous interactions, transactions with other Emacs buffers and so forth. With Multics mode, Emacs literally subsumes Multics, and the editor/environment distinction vanishes. With the advent of Multics mode, the question has been raised as to whether Multics Emacs is an editor with its own support mechanism, a de facto video system, a video system that should be within a Multics video support system, or the Multics Video system incarnate.

Experience and Conclusions

In the two years since its inception, Multics Emacs has grown from an experimental Lisp program to a twenty thousand line subsystem encompassing widely diverse Multics facilities and used across the country. It has inspired a wide variety of reaction, which in many ways is telling about the state of the computer marketplace.

In most ways, Multics Emacs shares the ITS Emacs experience: novice and experienced users find Emacs easy to learn and to use, and those who use video terminals rarely revert to earlier editing habits. People seem to become

productive and proficient with Emacs in less time than with the "conventional" editors. Skilled programmers build extension subsystems to accomplish sophisticated tasks, and libraries of personal Emacs extensions abound.

The impact of Emacs upon Multics, however, is quite unparalleled in the ITS experience. Cognoscenti at once recognized the ostensible similarity between Emacs and stand-alone word processing systems, and attempted to identify Emacs as an integral part of Multics Word Processing. Ways were sought to support dozens, or hundreds of Emacs users, in vain attempt to approach the economy of the stand-alone word processors. Nevertheless, the chief goal of Multics Emacs has been achieved, namely, to provide Multics with as powerful, advanced, and flexible an editing system as possible. It is a large augmentation of the power and capability of Multics, and a boon to those who already have or would have Multics systems: it is not intended to allow Multics to compete with stand-alone word processing. The power of a mainframe real-time editor, is not in its non-existent ability to replace dedicated minicomputer word processing systems, but in its extensibility, which allows it to perform more highly specialized and sophisticated editing tasks (e.g., the PL/I-FORTRAN error diagnostic mode above) than those of which any minicomputer system is capable.

The emergence of Multics Emacs has sensitized Multics users to the advantages of integrated video support, and real-time line editing. The widespread conversion of time sharing users to video terminals has left the printing-terminal oriented Multics interface far behind, and Emacs has let users see how video terminals can be managed intelligently. As a result, there is now substantial agitation for integrated video support in the Multics terminal support and communications areas.

Multics Emacs has been responsible for renewed interest in Lisp: many people wanting to write Emacs extensions have pursued Lisp (some knew no other programming language), and have rapidly become enamored of it. Multics Emacs is a tremendously potent tool for the creation and debugging of Lisp programs, taking all of the pain out of indentation, parenthesis balancing, and similar mechanizable tasks.

The power of Multics Emacs as a tool for developing itself, i.e., extensions, cannot be overstated. The line-by-line, function-by-function incremental input and debugging of code facilitated by Lisp mode and the active MacLisp environment allow code to be produced and debugged in seconds, and finely incrementally developed and reiterated in the same way that a sculptor polishes and examines his or her work. There is no interactive program development facility on Multics or ITS anything like Emacs developing its own extensions: the power of this facility has been directly responsible for the large number and wide variety of extensions that have come about.

Multics Emacs has also provided a testbed for experimentation with alternative Multics user interfaces, and highly customized environments. The user interface of Multics does not extend well to video terminals, and would need be almost completely overhauled for effective video terminal utilization: the mail system is a case in point. Multics Emacs has provided a path for experimentation with "entire alternative interfaces" to Multics.

Multics Emacs has provided a subject matter of interesting discourse between the Multics Development Community and other researchers at MIT, Stanford, and elsewhere working on similar related issues. In addition to a valuable infusion of new and radically different ideas into Multics, there has been a give-and-take with those investigating similar interfaces [Stallman] [Weinreb] [Anderson] [Schiller] during its development. Multics Emacs in fact "grew up" with these related editors, which were an active, state-of-the-art research topic at the time.

The choice of Lisp was a bold one: it has been the central artifact responsible for the rapid development of tremendous function in Multics Emacs. It is clear that an implementation in PL/I, with no concessions to elegance or modifiability, like one in assembler language, would support a larger number of users at smaller cost, but little of the present function would have been developed.

The choice of EMACS as an interface was an extremely fortunate one: all subsequent observation has indicated that many comparable mainframe (and small computer) text editor interfaces do not share the symmetry and regularity of EMACS. The lack of an input/edit mode distinction, the treatment of buffers as uniform strings of characters, and the uniform action of all commands with respect to the context in which they are used are the most significant contributors to this regularity: these attributes have been largely responsible for its ease of learning, and its conceptual advantage over the conventional Multics editors.

Acknowledgements

The author is deeply indebted to a large number of people for assistance and encouragement during the prehistory and development of Multics Emacs. I would like to thank:

William York, Gary Palter, and Richard Lamson, the other developers and maintainers of Multics Emacs, each responsible for numerous features, fixes, and enhancements, and infinite work.

Richard Stallman, for developing the entire concept of Emacs, and offering fervid and enthusiastic support from the first day.

Honeywell Information Systems, Inc., specifically Charlie Clingen, John Gintell, and Steve Webber, for the foresight to allow resources to be committed to this project, and for developing it into a Honeywell product.

Dan Weinreb and Dave Moon, for all natures of assistance, information, encouragement, and support too numerous to mention.

Larry Johnson, Jerry Stern, and Robert Coren, the maintainers/developers of the Multics communications software.

Earl Killian, Eugene Ciccarelli, and Bruce Edwards for early guidance and continued support, and their vast experience in terminal support and real-time display software.

Paul Schauble, for his contributions of FORTRAN mode and the FORTRAN/PL/I diagnostic scan mode.

Roger Roach and the staff of the MIT Information Processing Center for allowing experimentation with character-at-a-time real-time Lisp-coded editors on their service system, and allowing an Emacs user community to grow there.

The MIT AI Lab, for allowing me to use their system, and become familiar with ITS.

Lee Parks, Charles Frankston, Carl Hoffman, Lindsey Spratt, Suzanne Krupp, Margaret Minsky, Gerry Sussman, and a large number of other people for all sorts of help, encouragement, and ideas along the way.

Multics Emacs is defined and documented by two published Honeywell Manuals, [CH27], which describes the total user interface, and [CJ52], which details the extension writing language and facilities.

[AG91] Multics Programmer's Manual, Reference Guide. Order #AG91. Honeywell Information Systems, Inc.

[AG94] Multics PL/I Language Specification, Order #AG94, Honeywell Information Systems, Inc.

[Anderson] Anderson, Owen T., "The Design and Implementation of a Display-Oriented Editor Writing System", S.B. Thesis, MIT Dept. of Physics, January, 1979.

[CG40] QEDX Text Editor User's Guide, Order #CG40, Honeywell Information Systems, Inc.

[CH27] Emacs Text Editor User's Guide, Order #CH27, Honeywell Information Systems, Inc., December 1979.

[Chineual] Weinreb, D. & Moon D., "The Lisp Machine Reference Manual", MIT Artificial Intelligence Laboratory, 1979.

[CJ52] Emacs Extension Writer's Guide, Order #CJ52, Honeywell Information Systems, Inc., February, 1980.

[EDOC] Online Documentation on E editor (E.DOC[ALS,UP]), Stanford University A.I. Laboratory, Palo Alto, California.

[Halbert] Halbert, Daniel C., "A LISP Debugger for Display Terminals", S.B. Thesis, M.I.T., Cambridge, MA, 1978.

[ITSDOC] Eastlake, et al., "ITS 1.5 Reference Manual", AI Memo 161 and revisions, MIT Artificial Intelligence Lab, Cambridge, Mass.

[LispNotes] Greenberg, Bernard S., "Notes on the Programming Language Lisp", MIT Student Information Processing Board, 1976, 1978

[Moon] Moon, David A., "The MacLisp Reference Manual", MIT Project MAC, 1974.

[PARC] Teitelman, Warren, "A Display-Oriented Programmer's Assistant", Xerox Palo Alto Research Center, Report CSL-77-3, Palo Alto, CA, March 8, 1977

[Schiller]
Schiller, Jeffrey I., "TORES: The Text ORiented Editing System", revised from S.B. thesis, MIT Dept. of EE & CS, June 1979.

[Stallman]
Stallman, Richard M., "Emacs, the Extensible, Customizable Self-Documenting Display Editor", AI Memo 519, MIT A.I. Lab, June 22, 1979

[TECDOC]
Online documentation for TECO, MIT AI system.

[Weinreb]
Weinreb, Daniel L., "A Real-Time Display-Oriented Editor for the Lisp Machine", S.B. Thesis, MIT Dept. of EE & CS, January, 1979.

Appendix: The Extension Language

Multics Emacs extensions, whether part of the standard editor, loadable libraries, or user written, are written in Multics MacLisp, augmented by a set of Lisp macros provided as a lexically includable program fragment with Emacs. Extensions are written in an environment consisting of the native MacLisp functions (other than I/O), functions in the basic editor and standard extensions, and occasionally the redisplay code. The basic editor functions provide the ability to manipulate the current point, and the buffers, and inspect and change the contents of lines and buffers. Lisp macros are provided for syntactic sugaring of commonly used syntactic cliches, such as "create a temporary variable, assign a mark at the current point to it, perform some code, and free the mark", as well as to augment the basic expressive power of MacLisp.

The writer of extensions creates MacLisp functions via the Emacs command definition facility which associates with the defined function name a set of properties facilitating argument checking and prompting, as well as documentation. (All Emacs commands have online documentation, which can be obtained in many forms and in many ways, including explicit requests for information about the function of any given key.) In addition to invoking supplied functions in the extension environment, functions defined via the Emacs command definition facility may invoke each other (as may any Lisp functions), or be "connected" to keys, so that they will be invoked automatically by Emacs when selected keys are struck.

Extensions use as data strings, integers, buffer names, and marks (see above). The basic Lisp data types (symbols and lists (implemented via conses)) are only occasionally used. In fact, reasonably expert extension writing has been accomplished by persons completely ignorant of fundamental Lisp data object types. The fact that marks are implemented as lower level Lisp objects is transparent and irrelevant to the Extension writer: he or she is not allowed, and never has reason, to "decompose" them; such is the elegant nature of the Lisp object abstraction. The extension writer has no knowledge of or dealings with the internal representations of any data structure of the editor.

The most useful class of function used in extensions are those which are already capable of being invoked on behalf of user keystrokes by Emacs. For instance, "forward-word" is very commonly used in editing to position the cursor past the current word, which is how the Emacs user conceptualizes "what Escape-F does" (Escape-F being the two key sequence standardly used to invoke this common command). The extension writer, on the other hand, conceptualizes the "forward-word" function as moving the current buffer point to beyond the current word. Using these functions in extension functions is a valuable technique: the extension programmer can always experiment with the function to be used by invoking it in the normal interactive (i.e., via keystroke) way to determine details of its behavior.

Here is a simple example of an extension function based upon commands normally available through the keyboard. Its name is "bracket-word", and it places the word at which the cursor points in angle brackets:

```
(define-command bracket-word
        &documentation "Puts angle brackets around the word
        at which the cursor points."
        (forward-word)
        (insert-string ">")
        (backward-word)
        (insert-string "<"))
```

The function "insert-string" has the same effect as the interactive user typing a sequence of self inserting (trivial, printing) characters. The invocations of forward-word and backward-word position the current point prior to the insertions of the character strings. The end result of running this function would be the same as if the user had typed Escape-F (which invokes forward-word), a right-angle-bracket, Escape-B (which invokes backward-word), and a left angle bracket. The net result on the buffer (and the screen) is the same. However, the intermediate states which would be visible to the user typing the above sequence will not be visible on the screen when this extension is run, only the final state will be. This is because the interactive driver invokes the redisplay after each command character is typed, but this function (as is visible by inspection) does not invoke the redisplay, it invokes only what it is seen to invoke.

The most common extension environment macro is "save-excursion", which is used to remember the location of the current point, and restore it after the execution of the included code within the macro. For example, the following extension function places a star at the beginning of the current line, but leaves the cursor at the same place at the current line: (Bear in mind that the position is remembered via a mark, which is relocated automatically as the buffer text changes)

```
(define-command put-star-at-beginning-of-line
        (save-excursion
           (go-to-beginning-of-line)
           (insert-string "*")))
```

The "save-excursion" macro encompassing the invocations of go-to-beginning-of-line and insert-string ensure that the current point will restored after these functions run. Another similar macro, save-excursion-buffer, is used to restore the selection of buffer during its dynamic scope. As switching out of a buffer saves the location of the current point within that buffer, save-excursion-buffer subsumes the task of saving the point within that buffer.

Another set of very common macros in extension writing are those dealing with marks, providing for the creation thereof, and freeing at the end of the contained code. The macro

execution time: that mark will denote the point in the buffer which is current at the time the code contained in the macro begins execution. The following extension function deletes two words forward from the current point:

```
(define-command delete-two-words-forward
       (with-mark here
          (forward-word)
          (forward-word)
          (wipe-point-mark here)))
```

When delete-two-words-forward is invoked, a mark designating the current point in the buffer is created, and assigned to the local variable named "here". The generation of the mark and the local variable are all artifacts of the "with-mark" macro. The two calls to forward-word are then executed, presumably moving the buffer point (but not the saved mark) two words forward in the buffer, and then the function wipe-point-mark is invoked, passing that mark as an argument. The function wipe-point-mark deletes all text between the current buffer point and the point designated by the mark (saving it, incidentally, for possible user recovery). At the end of execution of delete-two-words-forward, the mark created by the macro is freed.

Another class of Emacs extension environment macros are those used to supplement (or reimplement) features in MacLisp thought to be inadequate, either for learning purposes, or ill adapted to the extension environment. For example, the extension documentation teaches the use of the "if" macro as opposed to the native MacLisp "cond" as the fundamental conditional construct. "if" is much simpler and straightforward, suffices for almost all cases, and is similar to the conditional construct in almost all languages other than Lisp. The native MacLisp "cond" is much more general and powerful, but this power is not often needed, and seems to have presented a stumbling block to those learning Lisp. Another macro of this class is "do-forever", and its exit form, "stop-doing". The native MacLisp "do" has two forms, one like the FORTRAN "do", and the other a powerful multi-variable generalization of this. Most often, the extension writer wants to iterate not over integer variables, but over buffer lines or characters: the iteration variable is thus the global editor state, and the need to specify or deal with variables which are almost never needed is undesirable. "if" and "do-forever" are illustrated by the following extension function, which either finds the first blank line of the buffer or complains if there are none:

```
(define-command find-first-blank-line
       &documentation "Moves cursor to the first blank line of
       the buffer."
       (go-to-beginning-of-buffer)
       (do-forever
          (if (line-is-blank)
              (stop-doing))
          (if (lastlinep)(display-error "No blank lines!"))
          (next-line)))
```

The form "(stop-doing)", if executed, causes control to exit the "do-forever" form. The function "lastlinep" (the suffix "p" is traditional Lisp nomenclature for predicates) tests for the current point being on the last line of the buffer. The function "display-error" causes an error message to be printed at the bottom of the screen, and a non-local transfer of control out of find-first-blank-line, aborting its execution. This non-local control transfer provides the reason that a "stop-doing" is not needed after the call to display-error.

Experience with the extension language has shown that its meaning is so transparent that the underlying Lisp is all but invisible: the emphasis of Lisp shifts from its data world to its being a formalism for organizing function invocation.