MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 643

25 February 1982

A Local Front End for Remote Editing

by

Richard M. Stallman

Abstract: The Local Editing Protocol allows a local programmable terminal to execute the most common editing commands on behalf of an extensible text editor on a remote system, thus greatly improving speed of response without reducing flexibility. The Line Saving Protocol allows the local system to save text which is not displayed, and display it again later when it is needed, under the control of the remote editor. Both protocols are substantially system and editor independent.

Keywords: Communications, Display, Editor, Extensible, Networks.

This report describes work done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505.

Introduction

The Local Editing Protocol and the Line Saving Protocol together solve the problem of combining a small dedicated local computer with a flexible shared remote one, providing all of the flexibility and power of the remote computer with most of the speed of the local one.

The Local Editing Protocol enables the local computer to perform most of the user's editing commands, with the locus of editing moving between the local and remote systems in a manner invisible to the user except for speed of response.

The Line Saving Protocol allows the remote editor to tell the local terminal to save a copy of some displayed text, and later refer to the copy to put that text back on the screen without having to transmit it again.

The Local Editing Protocol provides two types of benefits: improved response time for commands executed locally, and improved throughput on the remote system because the locally-handled commands are transmitted in bunches rather than individually, thus reducing the amount of scheduling and context-switching overhead.

The Line Saving Protocol speeds up display updating across slow connections by eliminating the need to output text which has been output before.

Both the Local Editing Protocol and the Line Saving Protocol are extensions of the SUPDUP protocol for hardware-independent display terminal control [see SUPDUP], and are suitable for use over any network that can transmit 8-bit bytes in both directions. They are independent of the command set of the text editor, and also of the precise manner in which unusual characters are displayed on the screen by the text editor.

This protocol has been implemented with the MIT Artificial Intelligence Lab's Lisp Machine system serving as the local system, and EMACS [see EMACS] running under the Incompatible Timesharing System on the Lab's PDP-10 as the remote system. The Lisp Machine [see Lisp Machine] is a powerful computer, chosen to facilitate testing the protocol, but the protocol does not require such a powerful system as the user end. It is intended for use on 8 and 16 bit micros.

This paper omits the lowest level details of the Local Editing and Line Saving Protocols. These details can be found in another paper, along with the rest of the SUPDUP protocol [see SUPDUP]. The precise protocol described there can be used by systems capable of the terminal-independent output required for the SUPDUP protocol. Alternatively, the higher-level description in this paper can serve as a guide for the design of special-purpose local editing protocols.

All numbers below are octal, except those preceding the word "bits" in describing the size of a data object, and dates.

The test implementation was written in the summer of 1981.

The Local Editing Protocol

1. Background Scenario

The typical display editor runs on a central timesharing system. The user has a local computer, a terminal, which sends his commands to the timesharing system. Eventually the editor runs, processes the commands, and produces display output which is sent to the terminal to appear on the screen. The terminal and central computer communicate using two streams of characters, which often run through networks.

With a system organized in this fashion, there is often a considerable delay between when a command is typed and when its effects appear on the screen. There are two causes of this: the delay in transmission through the network that connects the local system to the remote system, and the nonzero response time of the timesharing system.

A natural idea for providing more users with fast response is to make the editing operations execute in the user's terminal. Some editors transfer text to the programmable terminal, which executes the editing commands and later sends the text back to the central computer. Business form-filling data entry often works this way also.

When the ideal display editor had only thirty or so commands, all simple and fixed, it was as easy to implement them in a programmable terminal as on the central computer. Such an editor was used at MIT [see IMEDIT]. But nowadays users are growing to like powerful editors such as EMACS. A powerful display editor may have hundreds of editing commands that range in complexity from Move Forward Character and Delete Character up to Compile Function, Fill Paragraph, Delete Sentence, Check Spelling. Such an editor is a large program which includes an interpretive programming system such as Lisp, and it demands the power and flexibility of the large central computer. It does not fit in a programmable terminal.[1]

An editing command may be unsuitable for local execution because it refers to text far away from what is on the screen, because it refers to other data bases, because its definition is too complex, or because it was written or customized by the user.

When complex editors became available, users at MIT preferred them to the editor-in-the-terminal despite their slower response. They no longer accepted the simple but weak editor after having used the powerful one. The simple editor-in-the-terminal fell

---

[1] Some modern microprocessor chips might be satisfactory as CPUs, but they are rarely configured with enough real memory. They may support virtual memory, but the terminal has no swapping device. If the user's terminal becomes a true personal computer powerful enough to run a large Lisp program, then remote editing will not be needed, but with anything less, it is still desirable.

into disuse and has been forgotten for many years. But users would still prefer the instantaneous response such as they would have had with that editor.

The Local Editing Protocol is an attempt to provide a remote editor with fast response. The terminal or local computer performs the few simple editing commands which make up most of the user's keystrokes, while the central or remote computer contains the large amount of code which implements the large number of other editing commands, including user-defined commands. Because nearly all of the users keystrokes involve the few commonest editing commands, a small and fixed set of capabilities in the terminal are sufficient for this.

Some editors have been designed specifically to use editing or echoing features existing in timesharing systems. These include the editor E at Stanford [see E], which uses a system input-editing facility for editing within a line, and the editor Z at Yale [see Z], which uses system echoing for arrow keys. Some timesharing systems have received new features designed to allow insertion of text in an editor to be displayed by the timesharing system's ordinary input echo mechanism. EMACS benefits from such a feature on MIT's Incompatible Timesharing System. Multics EMACS is served by a similar feature [see Echo Negotiation]; but since echoing on Multics is done by an asynchronously operating front end processor, echo negotiation was obliged to attend to the problems of synchronization. The success of echo negotiation helped inspire the Local Editing Protocol.

2. SUPDUP

Here is a brief description of the SUPDUP protocol on which the Local Editing Protocol is built.

SUPDUP allows the user of a nonprogrammable full-duplex terminal connected to a local system to communicate over the Arpanet [see Arpanet] or Chaosnet [see Chaosnet] with a remote system as if he were on a terminal directly connected to that remote system, and obtain proper output on his screen, without requiring the remote system to know the language for operating the user's terminal. The local system, which presumably does know how to operate the terminals connected to it, translates display output from SUPDUP to the terminal's own language. Some directly connected[2] programmable terminals also use SUPDUP to talk to the central computer.

SUPDUP assumes a pair of streams that transfer 8-bit bytes. One stream goes from local system (or programmable terminal) to the remote system (or central computer). This is called the *input* stream. The other goes the other way and is called the *output* stream.

When a SUPDUP connection is created, the local system sends terminal-characteristics information on the input stream. This information says *what* the terminal can do, but not how. For example, it says whether the terminal can move its cursor arbitrarily, whether it can erase less than the entire screen at a time, and whether it can scroll. The height and width of the screen are also included.

Once the terminal characteristics have been sent, the two streams operate independently, each using its own language.

The SUPDUP output language uses character codes 0 through 177 to represent single-column graphic characters (including space) only. Character codes 200 and up are used as cursor motion, erasing or screen manipulation commands. For example, code 203 means Erase to End of Line. Code 217 means Move Cursor, and is followed by two characters which are interpreted as numeric cursor positions. Some of the 200-codes which were not previously used are defined to transmit the control information of the Local Editing Protocol.

The input data format in the SUPDUP protocol encodes the 12-bit characters of MIT extended ASCII into sequences of one or more 7-bit characters. Not all of the 12-bit codes are assigned meanings in MIT extended ASCII, and some of the previously undefined codes are used by the Local Editing Protocol and Line Saving Protocol as special commands, as described below. These commands can also be followed by additional characters which server as arguments.

Because protocol commands from the local system are transmitted in the same stream as the user's type-in, the protocol commands and type-in characters are read in the correct order by the remote system. The same holds for the protocol

---

[2] That is, no local computer aside from the terminal. Modems and telephone lines do not count because they are transparent.

commands and display output in the output stream. This property is vital to the functioning of the protocol.

Although there is no reason a local editing protocol could not use ordinary ASCII, it must be able to intersperse protocol commands with the user's type-in; the remote system must be able read them without knowing whether a protocol command or a type-in character will come next, and must be able to distinguish the commands from all possible user type-in once read. Encoding a wider byte and making the commands out of byte values which are not meaningful as input is a natural way of doing this. Use of ASCII control characters or the Escape character for protocol commands, as is done by the simple flow-control protocol used by DEC terminals, is very harmful if it prevents those characters from being used independently as user commands.

Here is a representative set of output command codes, to give the flavor of the SUPDUP output language. Arguments are simply byte values from 0 to 177.

Move-Cursor *vpos hpos*
: Move the cursor to the specified position.

Erase-to-End-of-Line
: Erase from the cursor to end of line. The cursor does not move.

Erase-to-End-of-Screen
: Erase from the cursor to end of line, and all lines below the cursor.

Clear-Screen
: Clear the screen and return cursor to the top left corner.

Erase-Char
: Erase the character after the cursor without moving the cursor.

Insert-Lines *n*
: Move all the lines at or below the cursor's vertical position down *n* positions. The bottom *n* lines are lost and *n* blank lines are created at the cursor.

Delete-Lines *n*
: Delete *n* lines starting at the cursor's vertical position, moving following lines up and creating blank lines at the end of the screen

Insert-Chars *n*
: Insert *n* characters after the cursor. Move text from cursor to end of line *n* positions right, creating *n* blank positions after the cursor. The characters in the last *n* positions on the line are discarded. The cursor does not move.

Delete-Chars *n*
: Delete *n* characters after the cursor. Move text from cursor to end of line *n* positions left, discarding the contents of the first *n* positions after the cursor, and making *n* new blank positions at the end of the line.

3. Design of the Protocol

Like SUPDUP, the Local Editing Protocol is suitable for use between a programmable terminal and a central computer, or between a local computer equipped with nonprogrammable terminals and a remote computer connected to it over a network. The protocol works the same either way. From now on, we refer to the two computers involved as the *local system* and the *remote system*.

Any local editing protocol must accomplish these things:

1. The remote editor must be able to tell whether the local system supports local editing.
2. The representation used for the remote editor's text display output must contain sufficient information for the local system to deduce the text being edited, at least enough to implement the desired editing commands properly.
3. The remote system must be able to tell the local system when to do local editing. This is because the remote system is not always running the text editor, and the editor may have modes in which characters are not interpreted in the usual way. If the protocol is to work with a variety of editors, the editor must be able to tell the local system what the definition is for each editing command, and which ones cannot be handled locally at all. Extensible editors require the ability to tell the local system a new definition for any character at any time.
4. The local system must be able to verify for certain that all the input it has sent to the remote system has been processed completely. We call this process "synchronization". To attempt local editing if this condition is not met would cause timing errors and paradoxical results.
5. The local system must be able to tell the remote system in some fashion about any editing that has been performed locally.
6. Local editing must cease whenever an unpredictable event such as a message from another user causes output which the local system will misunderstand.

Each component of the Local Editing Protocol is present to satisfy one of the design requirements listed above. So the description of the protocol is broken up by design requirement. Each requirement is the subject of one section below.

3.1. Telling Whether the Local System Supports Local Editing

Initializing a SUPDUP connection always involves sending several bytes of terminal capabilities information from the local system to the remote system. A previously unassigned bit in one of them is used to signify that the terminal can do local editing.

3.2. Making Output Comprehensible for Editing

The primary purpose of the SUPDUP protocol is to represent display terminal I/O in a hardware-independent fashion. A few new display commands plus some conventions as to how the existing commands are used by remote editors are enough to allow the contents of the edited text to be deduced.

Output to the terminal in the SUPDUP protocol uses codes 0 through 177 for single-column graphic characters only. Other output commands are sequences of one or more characters, whose first character code is 200 or above.

A local editing terminal must not only obey these commands but also keep a record of the characters on the screen suitable for implementing editing commands. This basically consists of keeping an up-to-date matrix which describes what character is at each screen position. However, it is not as simple as that. The problem cases are:

1. Space characters and tab characters in the text being edited must be distinguished. Although a tab character may look the same as a certain number of space characters at one particular time, it does not behave like that number of space characters if text is inserted or deleted to the left of it.
2. Control characters may be output in a fashion indistinguishable from some sequence of printing characters. For example, EMACS displays the ASCII character control-A as "↑A". Editing commands which operate on single characters may be confused by this.
3. One line on the screen may not correspond to one line of text being edited. EMACS represents a long line of text using several screen lines. Some other editors allow text to be invisible before the left margin or after the right margin.
4. Not all of the screen space is used for displaying the text being edited. The rest of the screen is used for other things, and the local system must avoid moving the cursor into them or editing them as if they were text.
5. Space characters at the end of a line in the text being edited must be distinguished from blank space following the displayed line, since some editors (including EMACS) distinguish them. Both the Forward Character and End of Line commands need this information.

For each of these problem cases, the Local Editing Protocol has an editor-independent solution:

1. We enable the local system to distinguish spaces from tabs in the text being edited by defining an output command Space-for-Tab. It is used by the remote editor to represent the spaces which make up the representation of a tab character. The local system displays it just like a space character, but records it differently.
2. An editor-independent solution to the problem of ambiguous display of control characters is a new output command, Multi-Position-Char *n ch*,

which says that the next *n* screen positions are part of the display of one character of text, with code *ch*. Using this, EMACS outputs a control-A using five characters: Multi-Position-Char 002 ↑A ↑ A. The local system must record this, keeping track of which positions are grouped together to represent one text character. The record should be cleared when the positions involved are erased.

3. To deal with continuation lines, we define two new output commands, Line-Beginning-Continued and Line-End-Continued. These tell the local system that the beginning or end of the screen line, respectively, is not really the beginning or end of a line in the text being edited. For EMACS, Line-End-Continued on one line would always accompany Line-Beginning-Continued on the next, but this might not be appropriate for other editors. EMACS always outputs a Line-Beginning-Continued for the first line on the screen, since EMACS does not attempt to think about the text before what appears on that line.

   Both of these commands must set bits that accompany the lines when they are moved up or down on the screen. The Line-End-Continued bit should be cleared by anything which erases the end of the line, and by Delete-Chars within the line. The Line-Beginning-Continued bit should be cleared by anything which erases the entire line.

4. Any parts of the screen that are being used for things other than the text being edited can be marked off limits to local editing by setting the editing margins. Inside each edge of the screen there is an editing margin. The margin is defined by a width parameter is normally zero, but if it is set nonzero by the remote system, then that many columns or rows of character positions just inside the edge are made part of the margin. By setting the margins, local editing can be restricted to any rectangle on the screen. The margins are set by means of a special output command.[3]

   If the editor supports multiple windows, the editing margins can be used to restrict editing commands to the screen area of the selected window. Editing within one window can then be handled locally, but commands to select different windows cannot be, without extending the protocol. If the same text unit is on display in more than one window, and both displays are to be updated by editing either one, then local editing should not be requested by the remote editor.

5. Space characters at the end of a line are distinguished by means of conventions for the use of the erasing commands Erase-to-End-of-Line, Clear-Screen, and Erase-Char. The convention is that Erase-Char is used only to clear text in the middle of a line, whereas the others imply that all the text on the line (or all the text past the point of erasure) will be reprinted.

   Although the area of screen erased by Erase-to-End-of-Line or Clear-

---

[3] Actually, Define-Char with special arguments is used to do this. See below.

Screen becomes blank, the screen-record matrix elements for those areas is filled with a special character, distinct from the space character and from all graphic characters. Character code 200 will serve for this. Spaces in the text are output by the editor as actual spaces and are recorded as such in the screen-record matrix. Blanks created in the middle of the line by Insert-Chars are recorded as space characters since character insertion is done only within the text of the line. Blanks created at the end of the line by Delete-Chars are recorded as character 200 since they are probably past the end of the text (or else the text line extends past this screen line, which will be reported with Line-End-Continued).

If the screen record and the editor obey the above conventions, the end of the text on the line is after the last character on the line which is not 200.

3.3. Defining the Editing Commands

When a remote editor requests local editing, it must tell the local system what editing function belongs to each character the user can type. If character meanings change during editing, the editor must tell the local system about the changes also. In most display editors, text is inserted by typing the text, so we regard printing characters as editing commands which are usually defined to insert themselves.

The Local Editing Protocol defines a new output command, Define-Char, for this purpose. A Define-Char command usually specifies the editing meaning of one user input character. The first time the remote editor enables local editing, it must first specify the definitions of all user input characters with Define-Char commands. On subsequent occasions, Define-Char commands need only be used for characters whose meanings have changed.

Each Define-Char command contains a user input character and a function code. The function codes are established by the Local Editing Protocol specifically for use in Define-Char. Normally the function code specifies the editing definition of the associated user input character. A few function codes indicate the special forms of Define-Char command which set parameters or initialize everything.

No local system is required to handle all the function codes. For example, our preliminary implementation does not handle the vertical motion, insertion of line breaks, or arguments. Any function code which the local system prefers not to handle can be treated as code 0; the characters with that definition are simply not handled locally. **The local system is also free to fail to handle any command character locally for any reason at any time.**

Here are some Define-Char function codes, with the rules for implementing them in the local system. We have chosen an illustrative sample. Refer to the paper on SUPDUP [see SUPDUP] for a full list.

0 This character should not be handled locally. Either it is undefined, or its definition in the remote editor has no local equivalent.

1 Move forward one character. Do not handle the command locally at the end of the text on a line.

2 Move backward one character. Do not handle the command locally at the beginning of a line.

3 Delete the following character. Do not handle the command locally at the end of the text on a line, or on a line which is continued at the end.

5 Move backward one character. Treat a tab as if it were made of spaces. Do not handle the command locally at the beginning of a line.

6 Delete the previous character. Treat a tab as if it were made of spaces. Do not handle the command locally at the beginning of a line, or on a line which is continued at the end.

7 This is the usual function code for printing characters. The insertion mode parameter says what characters assigned this function code ought to do. They can insert (push the following text over), or replace (enter the text in place of the following character). Or they may be not handled at all. See function code 32, which sets the insertion mode. When the current mode is "replace", if the following character may occupy more than one position, the command should not be handled locally. When the insertion mode is "insert", the command should not be handled locally on a line continued at the end.

10 Move cursor up vertically. Do not handle the command locally if the cursor would end up in the middle of a tab character, or past the end of the line, or if either the line which the cursor starts on or the previous line is continued at the beginning.

16 Insert a line-break at the cursor and move the cursor after it.

17 Insert a line-break at the cursor but do not move the cursor.

20 Move cursor to beginning of current line. Must not be handled locally if this line has been marked as continued at the beginning.

22 Use the definition of another, related character. The related character for a lower case letter is the corresponding upper case letter. Some other characters have related characters defined.

27 Specify a digit of a repeat count. The low seven bits of the command character with this definition should be an ASCII digit. This digit should be accumulated into a repeat count for the next command. The next character which is not part of the repeat count uses the repeat count. The argument-specifying characters cannot be handled locally unless the command which uses the argument is also handled locally. So those characters must be saved and not transmitted until the following command has been read and the decision on handling it locally has been made. If it is impossible to handle all the specified repetitions of the command locally, the command should not be handled locally at all.

43 Scroll up. Scrolls region of editing up a number of lines specified by an accumulated numeric argument. Do not handle locally if no numeric argument accumulated locally. Local handling is possible only if the local system knows the contents of lines off screen due to use of the line saving protocol.

All insertion and deletion functions should take special notice of any tab characters (output with Space-for-Tab) on the line after the cursor. Unless the local system wishes to make assumptions about the tab stops in use by the remote system, no insertion or deletion may be done before a tab. In such a situation, insertion and deletion commands should not be handled locally. We could define a way for the remote system to communicate the tab stops, but it is probably not worth while.

Some functions are deliberately left undefined in particular unusual cases, even though several obvious remote definitions could easily be simulated, so that they can be used by a wider variety of editors. For example, editors differ in what they would do with a command to move forward one character at the end of the text on a line. EMACS would move to the beginning of the next line. But Z would move into the space after the line's text. Since function code 1 is defined in this case not to be handled locally, either EMACS or Z could use it.

3.3.1. Additional Parameters Affecting Command Execution

In addition to the definitions of user input characters, the local system needs to know certain other parameters which affect what editing commands do. These are the word syntax table, the fill column, the insertion mode, and the editing margins.

Commands which operate on words must know which text characters to treat as part of a word. This can be changed by activities on the remote system (such as, switching to editing a different file written in a different language). A special form of Define-Char command is used to specify the word syntax bit for one text character. Some relevant function code definitions include:

23 Move cursor to the end of the following word. Do not handle locally if the word appears to end at the end of the line and the line is marked as continued at the end.

25 Delete from cursor to the end of the following word. Do not handle locally on a line marked as continued at the end.

31 This function code does not actually define the accompanying character. Instead, it specifies the word syntax table bit for one ASCII character. As arguments, it requires an ASCII character and a syntax bit, which are encoded to fit where normally would be an editing command character code.

EMACS has an optional mode in which lines are broken automatically, at spaces, when they get too long. In such a feature, certain characters[4] which are normally self-inserting may break the line if the line is too long. Such characters cannot simply be defined as self-inserting. Instead, they should be defined as *self-inserting before the fill column*. The fill column is a parameter whose purpose is to specify where on the line these characters should cease to be handled locally. A few other function codes are affected in a similar manner. A special form of Define-Char sets the value of the fill column. Examples:

---

[4] In EMACS, the Space character and the Return character.

40 Self-insert, within fill column. Interpreted like code 7 (self-insert), except do not handle locally if the current cursor column is greater than or equal to the fill column parameter (plus the left editing margin).

41 Set Fill Column. This function code specifies the value of the fill column parameter. The numeric value of the accompanying "character" is the new value of the parameter. If the fill column parameter is zero, the fill column feature is inactive.

Since many editors, including EMACS, have modes which ordinary printing characters either push aside or replace existing text, the Local Editing Protocol defines an *insertion mode* parameter to control this. If the insertion mode parameter is 1, ordinary printing characters insert. If the insertion mode parameter is 2, they overwrite existing text. If the parameter is 0, all ordinary printing characters become unhandleable. Note that which characters are "ordinary printing characters" is controlled by the assigned function codes.

A special form of Define-Char command is used to set the insertion mode parameter. To switch between insert and overwrite modes, it is sufficient to change this parameter; it is not necessary to change the definitions of all the printing characters. Also, simple commands which change the insertion mode can be handled locally, though we have not actually defined any such function.

32 This function code is used to specify the insertion mode parameter. The accompanying "character" should have numeric code 0, 1 or 2. This "character" is the new setting of the parameter. 0 means that the ordinary printing characters, those characters with function code 7, should not be handled locally. 1 means that they should insert, and 2 means that they should replace existing text. The definition of character code 0, 1, or 2 is not changed.

Another special form of Define-Char is used to set the *editing margins*, which say what part of the screen is used for local editing.

34 Set margin. This function code specifies one of the editing margins. A two-bit field to identify which editing margin, and a value for the margin, are encoded in the place of an editing command character code.

Another special form of Define-Char command initializes the definitions of all user input characters, all the word syntax table bits, and the insertion mode to a standard state. This state is not precisely right for any editor, but it gets most characters right for just about all editors. It sets up all ASCII printing characters to insert themselves; aside from them, the number of commands handled locally for any given editor is small. When an editor is first transmitting its command definitions to the local system, it can greatly reduce the number of Define-Char commands needed by first initializing all characters, and then fixing up those command definitions and parameters whose standard initial states are not right.

33 This function code ignores the accompanying character and initializes the definitions of all command characters, as well as the word syntax table, editing margins, fill column and insertion mode.

3.4. Synchronization

The local system sees no inherent synchronization between the channels to and from the remote system. They operate independently with their own buffers.

When the local system receives an input character from the user which it cannot handle, either because its definition is not handleable or because local editing is not going on at the moment, it sends this character to the remote system to be handled. After processing the character, the remote system may decide to permit local editing. It must send a command to the local system to do so. By the time the local system receives this command, it may already have received and sent ahead several more input characters. In this case, the command to permit local editing would be obsolete information, and obeying it would lead to incorrect display.

The Local Editing Protocol contains a synchronization mechanism designed to prevent such misbehavior. It enables the local system, when it receives a command to begin local editing, to verify that the remote system and the communication channels are quiescent.

Before the remote editor can request local editing, it must ask for synchronization. This is done by sending the output command Prepare-for-Local-Editing. After receiving this, the local system lays the groundwork for synchronization by sending a resynchronize command to the remote system. This is a two-character sequence whose purpose is to mark a certain point in the input stream. By counting characters, later points in the input stream can also be marked, without need for constant resynchronize commands.

The resynchronize command begins with a special code which was previously meaningless in the SUPDUP input language. The second character of the sequence is a identifier which the remote system will use to distinguish one resynchronize command from another. It is best to use 40 the first time, and then increment the identifier from one resynchronize to the next, just to avoid repeating the identifier frequently.

When the remote editor actually wishes to permit local editing, after sending Define-char commands as necessary, it sends a Do-Local-Editing command, also known as a resynch reply. This command is followed by two argument bytes. The first one repeats the identifier specified in the last resynchronize command received, and the second is simply the number of input characters received at the remote system since that resynchronize command.

When the local system sees the resynch reply, it compares the identifier with that of the last resynch that *it* sent, and compares the character count with the number of characters *it* has sent since the last resynch. If both match, then all pipelines are empty; the remote system has acknowledged processing all the characters that were sent to it. Local editing may be done.

If the resynch identifier received matches the last one sent but the character counts do not, then more input characters are in transit from the local system to the remote system, and had not been processed remotely when the Do-Local-Editing was sent. So local editing cannot begin.

If the identifiers fail to match, it could be that the remote system is confused. This

could be because the user is switching between two instantiations of the editor on the remote system. After each switch, the newly resumed editor will be confused in this way. It could also be that the remote editor sent a resynch reply for the previous resynch, while the last one was on its way. In either case, the proper response for the local system is to send another resynch. It should wait until the user types another input character, but send the resynch before the input character. This avoids any chance that a resynch itself will prevent local editing by making the pipelines active when they would have been quiescent.

> A consequence of sending the resynch before the next input character is that, if the remote system sends a reply to an earlier resynch, the pipelines are not quiescent. It will eventually reply to the latest resynch after it has processed the remaining input characters.

The first Do-Local-Editing after a Prepare-for-Local-Editing is usually preceded by many Define-Char commands to initialize all the editing commands. Later Do-Local-Editing commands are preceded by Define-Chars only for commands whose meanings have changed.

Since the character count in a Do-Local-Editing cannot be larger than 177, a new resynchronize should be sent by the local system every so often as long as resynchronization is going on. We recommend every 140 characters. Once again, the resynch should be sent when the next input character is in hand and ready to be sent. If the remote system sees more than 177 input characters without a resynchronize, it should send another Prepare-for-Local-Editing.

Once synchronization has been verified and local editing begins, it can continue until the user types one character that cannot be handled locally. Once that character has been transmitted, local editing is not allowed until the next valid Do-Local-Editing.

3.5. Reporting Results of Local Editing

When the local system has done local editing, it must eventually report to the remote system what it has done, so that the changes can be made permanent, and so that following commands handled remotely can have the proper initial conditions.

In the Local Editing Protocol, the local editing is reported by transmitting the editing commands themselves!

This is the simplest way to do it, and has no other disadvantages. The remote editor must already know how to interpret its own command language! In addition, the editing commands are usually a fairly brief description of the changes made. Editor command languages are designed with this as a goal.

Finally, some editing commands have far-reaching and vital side effects which are not immediately visible. For example, the command in EMACS to delete a word after the cursor also saves this word in a buffer so that other commands can insert it again. The local system cannot do this, because the commands to reinsert the text are not handled locally.[5] If the delete command is reported to the remote EMACS by sending

---

[5] Even if they were normally handled locally, the user could always write a new one which the local system did not know about.

the command itself, EMACS will eventually save the deleted word just as it would have if the command had been handled remotely. The local system and protocol need not concern themselves with the remote editor's policies on how to save killed text.

The locally-handled commands being transmitted to the remote system must be marked as such to avoid double-echo. This is done by preceding each batch of locally-handled commands with a two-character sequence beginning with a character code which was previously meaningless in the SUPDUP input language. The second character of the sequence encodes the number of locally-handled commands in the batch.

When the remote editor receives the locally-handled commands, it executes them as usual, but it does not actually update the display. It only *pretends to itself* that it did. It updates all its tables of what is on the screen, just as it would if it were updating the display, but it does not output anything. This brings the remote editor's screen-records into agreement with the actual screen contents resulting from the local editing.

Local systems are recommended to output all accumulated locally handled commands every few seconds if there is no reason to do otherwise. Sending a few seconds worth of input commands all at once causes the remote editor to run for fewer long periods rather than many short periods. This can greatly reduce system overhead in timesharing systems on which process-switching is expensive.

3.6. Unexpected Output From Remote System

When the remote editor has permitted local editing, there should normally be no output from the remote system as long as the local system continues to handle characters locally. Only when an input character is sent on for remote handling will the remote editor produce further output.

Output can arrive at the local system while local editing is in effect only if the remote system does output which the remote editor was not responsible for. This might, for example, be a message received from another user. Since any output which was not from the editor means that the local system's screen record no longer reflects the text being edited, local editing must cease. The Local Editing Protocol says that resynchronization should cease entirely until the next Prepare-for-Local-Editing is received. The remote system should notify the editor to send another Prepare-for-Local-Editing.

There is also a special Stop-Local-Editing command which the remote system can send immediately, to reduce confusion, at any time when it becomes aware that the terminal is going to be put to other uses.

4. Local Handling of User-Written Editing Commands

Support for the Local Editing Protocol is actually part of the TECO editor programming system in which EMACS is written. This is because the low levels of I/O, and display updating, are all built into TECO. The definitions of editing commands within TECO are TECO programs supplied by EMACS. So TECO cannot have built-in knowledge of what editing command definitions correspond to what Define-Char function codes.

Therefore, TECO allows the editor written in TECO to supply a table which establishes a correspondence between TECO programs which are possible editing command definitions and Local Editing Protocol function codes. Whenever a command's definition must be transmitted to the local system in a Define-Char, TECO takes its current definition as a TECO program and searches the table for the corresponding function code. If the TECO program is not found in the table, function code 0 (no local handling) is used.

5. Evaluation of Local Editing

An editing session several hours long during the preparation of this paper was recorded so that the effectiveness of the Local Editing Protocol could be measured. The results are reported here. Numbers in this section are decimal.

The session involved approximately 6550 characters. Of these, 5550 were handled locally. 1000 characters were not. This gives an effectiveness of 85 per cent.

The 85 per cent figure actually measures the effectiveness of the particular implementation of the Local Editing Protocol, rather than the protocol per se. Each local system is free to choose which function codes to handle, and the test implementation does not handle vertical motion or line breaking. These two types of commands, together with their numeric arguments which the test implementation also does not handle, account for 440 characters. 120 other characters were not handled locally because they were typed too soon after a vertical motion command or a line breaking command, before resynchronization had taken place.

If the full set of Local Editing Protocol function codes were implemented by the local host, these 560 additional characters would have been handled locally, bringing the total to 6110, or 93 per cent. This figure more accurately represents the effectiveness of the protocol itself, as described in this paper.

The locally and remotely handled commands were far from randomly interspersed. They fell into large bursts. Frequently as many as sixty characters in a row were handled locally. Bursts would have been much longer if the line-breaking commands were implemented for local handling. This probably means that the protocol provides more satisfaction than the percentage figure might indicate; but this is somewhat subjective.

A question that now occurs is how much the effectiveness could be improved by addition of new, easily implemented function codes. Why were the remaining 440 characters not handled locally? The answer appears to be that most of them were commands that could not easily be handled locally, and the effectiveness of local editing could not easily be improved.

About 70 characters were commands such as Kill Line or to End of Line and Delete Whitespace around Cursor, together with their arguments. These commands would not be difficult to handle locally, and are not included in the protocol design because the precise details of the commands in EMACS were considered too idiosyncratic to merit inclusion. It appears that including them would not help very much; only 1 per cent.

About 100 characters were commands that are normally handled locally, but were not on particular instances, due to continuations or to operating across line boundaries. The figure of 100 also includes characters that were not handled locally because they were typed too soon after others not handled locally because of continuation lines; it is not easy to distinguish the two causes based on the records kept. With care, or perhaps with additional information provided about line boundaries or with a more editor-specific protocol, some of these cases could be handled locally.

But more than half of the commands that could not have been handled locally, about 240, were simply unsuitable. These included search commands (most of which went off-screen), file commands, sentence and paragraph commands, and scrolling-by-screenful commands. Sentence and paragraph commands are considered unsuitable because their definitions are complicated and depend on the language being edited. In principle, appropriate parameters could be defined and passed to the local system, but we decided that the restrictiveness and complexity was not worth while.

Scroll-by-lines commands and their arguments made up 12 characters only. These commands would have been handled locally if implemented and if anticipatory output, or memory of previous output scrolled off the screen, allowed the local system to know what data to scroll on. See below, under the Line Saving Protocol, for anticipatory output.

Another class of unhandleable commands were the close-parenthesis and close-bracket characters. These were not handled locally because they were defined to move the cursor to the matching open parenthesis or bracket, not merely to self insert. It would be possible to handle these locally if the syntax table used for parenthesis matching were transmitted to the terminal; but in this session there were only 16 such characters, plus a few more affected by ripple from them. This indicates that special effort for local handling of these characters would not be worth while. For editing Lisp code, the conclusion might be different!

After each remotely handled character, the Local Editing Protocol must wait for the pipelines to become empty before enabling local handling. This may not happen as long as the user is typing fast. Thus, failure to handle one character may prevent the local handling of many following characters. We counted the number of times this occurred due to a vertical motion or line breaking command, and how many following characters were affected each time.

The phenomenon occurred 30 times, affecting 120 characters in all. On 18 of the 30 occasions, only one following character was affected. The remaining events were evenly distributed from 2 through 11 characters, except that one event affected 14 characters, and one affected 25 characters. The conclusion is that the propagating inhibition phenomenon is real, and can potentially last a long time. But it is not a significant source of ineffectiveness of the protocol. In all, there were 440 line breaking and vertical motion commands, and they inhibited 120 other commands. If the number of characters inhibited is generally around 1/4 as many as those which potentially cause the inhibition, then inhibition increases ineffectiveness only by 1/4, which is not a serious problem. It must be noted that the amount of inhibition probably depends on the user's typing habits and speed.

The remaining cases of inhibition are included in the 70 characters affected by continuations and ends of lines. They too showed some tendency to form large clusters. But it is not possible to tell from the information available which were actually affected by inhibition and which by the continuations and ends of lines. This is why it is more accurate to exclude them. There were practically no cases of inhibition from other causes, possibly because I tend to pause to think or watch the screen after other sorts of editing commands.

The Line Saving Protocol

6. Motivation

The Local Editing Protocol was designed to improve response time. The Line Saving Protocol addresses a different problem: that of slow communication lines between the local and remote systems.

Slow lines have little effect on editing commands whose effects on the text are localized. Using typical modern terminals which support the insert/delete line/character operations, display updating for small insertions and deletions of text do not require a large number of characters to be output.

The inconvenience of slow lines comes for commands which move to a distant part of the text being edited. These usually require transmitting the entire screen. Commands to scroll the text on the screen also require transmitting several lines.

The Line Saving Protocol reduces the amount of output required in such situations by allowing text once displayed to be saved, and restored whenever the same text is to be displayed again. Moving to a part of the file which had been displayed at an earlier time no longer requires transmitting the text again.

The inconvenience of the slow line cannot be completely eliminated, because it is always necessary to transmit any piece of text at least once. However, the Line Saving Protocol allows the remote system to anticipate future needs by sending and saving text which is not to be displayed at all, so that it can be displayed quickly if it is needed later. This can be used for text surrounding that which is currently displayed, so that short-distance scrolling commands will be faster. For a small enough file and large enough memory in the terminal, it could be used for the entire file.

7. The Design of the Line Saving Protocol

The basic operations provided by the Line Saving Protocol are those of saving and restoring the contents of individual lines on the screen.

A line on the screen is saved for later use by giving it a label. The line remains unchanged on the screen by this operation, but a copy of the contents are saved. At a later time, the saved contents can be brought back onto the screen, on the same line or another one, by referring to the same label.

A new output command Save-Lines is defined, taking two arguments: the number of lines to save, and the label. Several consecutive lines are saved under consecutive labels, starting with the line the cursor is on and the specified label.

Another new output command is defined for restoring the saved text. Restore-Lines takes a number of lines and a label, just like Save-Lines, but it copies the saved text belonging to the labels back onto the screen. The labels are forgotten when they are retrieved. This is because the Line Saving Protocol is oriented toward local systems whose memory capacities are not enough to permit a strategy of saving all lines ever transmitted and never discarding any.

A label number is 14 bits wide, but terminals are not expected to handle that many labels. Each terminal specifies the largest label it can handle, as part of the terminal characteristics. The remote system should not try to use a larger label number. The terminal ought to be able to remember approximately that number of labels under normal circumstances, but it is permitted to discard any label at any time (because of a shortage of memory, perhaps).

Because some editors can use for text display a region of the screen which does not extend to the left and right margins, we define another new output command to specify which range of columns should be saved and restored. Set-Saving-Range *x1 x2* specifies that contents of lines should be saved and restored from column *x1* (inclusive) to column *x2* (exclusive).

Line contents should be saved and restored by actual copying. It might be possible to design other implementation techniques involving pointer manipulation which could be stretched to produce the same results, but there is no point in doing this. The Line Saving Protocol is aimed at situations where the slow speed of output is the limiting factor. There will be plenty of local CPU power available for copying the contents of lines.

8. Local Systems with Limited Memory

A local system with small memory capacity may reach a point where it can no longer keep track of all the saved lines. To provide for this, we permit the local system to forget labels at will. At times the remote system will attempt to refer to a label which the local system has forgotten. For these situations, we provide the local system with a way to tell the remote system about forgotten labels.

The local system can use the input command Label-Failure *n l* to say that *n* labels starting with *l* had been forgotten when an attempt was made to retrieve them. When the remote editor receives this command, it retransmits explicitly the contents of any screen line which was supposed to have been set up by restoring this label. The remote system need not attempt to predict which labels are still known in the local system. It can try to restore from a label; if this does not work, it will receive a Label-Failure command and should send the actual line contents.

9. Interaction between Local Editing and Line Saving

Some of the commands which the Local Editing Protocol provides for handling locally can push lines off of the screen. To provide for proper interaction between the two protocols, we define a new output command which tells the local system how to save lines which are removed by the screen by locally handled commands.

The output command Set-Local-Label *l* sets the label to be used for the next line to be pushed off the screen by local editing. Successive such lines decrement this parameter by 1. If several lines are pushed off the screen at once, they are processed lowest first. This tends to cause several consecutive lines which have been pushed off the screen to have consecutive labels, whether they were pushed off together or individually. That way, they can be restored with a single Restore-Lines command.

Because the remote system is always expected to know how locally handled commands have updated the display, the remote editor can always tell when a line has locally been given a label.

10. Implementation

Line saving was implemented as a test on the Lisp machine, but since the network connection in use is actually faster than character drawing, there was no visible change in performance. Only debugging tools could show that line saving and restoring was actually taking place! But it was demonstrated that line saving operated when it should have, and produced the correct display.

11. Anticipatory Output

The disadvantages of a slow connection can be overcome considerably by using time when the output stream is idle to transmit text to be displayed later. Such anticipatory output works with both line saving and local editing.

Using anticipatory output with line saving, one would transmit a line of text which would not appear on the screen, and then save it under a label to be restored under remote control.

Using anticipatory output with local editing, one would transmit a line of text near in the editor buffer to the text which is displayed, and identify it so that locally handled scrolling commands can move it onto the screen.

We define one command for labeling anticipatory output, which works with either line saving or local editing, whichever the local system supports, or with both.

To begin a line of anticipatory output, the remote editor sends a Move-Cursor-Invisible command, which is followed by a 14-bit signed number called the logical vertical position. The value of this number indicates the relationship between this line of output and the text on the screen, for the sake of local editing. After the output is done, the contents can be saved under a label, if the local system supports line saving. In any case, text output after the cursor has been moved with a Move-Cursor-Invisible should not appear on the screen.

The logical vertical position specifies the position of this line on an infinitely long screen which contains the actual lines of locally editable text at their actual positions. For example, if the current editing margins specify that lines 2 through 20 are used for local editing, then a line output at logical vertical position -3 contains what would appear five lines above the first displayed line of edited text, and a command to scroll down five lines would bring it onto the screen at line 2. A line might be displayed at logical vertical position 1; it would be invisible, but a command to scroll down one line would make it visible on line 2.

The Move-Cursor-Invisible command should not affect the cursor which is actually shown to the user on the screen. As long as output is going to an invisible line, the cursor on the screen should remain where it was just before the Move-Cursor-Invisible. There is no need to move the logical cursor back to a visible place until it is time to do visible output again.

When local editing is in progress, unexpected output which moves the cursor to an invisible line, or outputs characters to such a line, should not terminate local editing. Only unexpected output to the actual screen should do that. This is because anticipatory output comes only from the program that is using local editing or line saving. Moving the cursor to a visible point should also terminate local editing. Output which is actually unrelated to the editor ought to start with a Move-Cursor and will be detected by this test.

If the local system does not support local editing, the value used for the logical vertical position is immaterial; the only purpose of outputting the line is to save it under a label. In this case, the local system is not required to save multiple invisible lines according to logical vertical position. It may keep only the last one output. So each transmitted line should be saved under a label as soon as it is finished.

The remote operating system and the network are likely to have a large buffering capacity for output. Since anticipatory output is used primarily on slow connections, the remote editor could easily produce and buffer in a second an quantity of anticipatory output which will take many seconds to transmit. While the backlog lasts, the user would obtain no service for his explicit commands, except those handled locally. To prevent this, the remote editor ought to send anticipatory output in batches of no more than two or three lines' worth, and send these batches at an average rate less than the expected speed of the bottleneck of the connection. In between batches, it should check for input.

A new terminal-characteristics field is defined to say approximately how many lines of anticipatory output the local system can support for local editing use. The remote system should use this as a guide. It is free to send whatever it wants, and the local system is free to decide what to keep and what to throw away.

References

Arpanet

"Arpanet Protocol Handbook", Network Information Center, SRI International.

Chaosnet

Dave Moon, "Chaosnet", MIT Artificial Intelligence Lab memo 628, June 1981.

Echo Negotiation

Bernard S. Greenberg, "Multics Emacs: an Experiment in Computer Interaction" in Proceedings, Fourth International Honeywell Software Conference, Honeywell, Inc., March, 1980, Bloomington, Minn.

E

E.ALS[UP,DOC

, an on-line documentation file on the Stanford WAITS computer system, describes the editor E.

EMACS

Richard M. Stallman, "EMACS, the Extensible, CUstomizable, Self-Documenting Display Editor", Artificial Intelligence Lab memo 519a, April 1981.

IMEDIT

Jack Haverty, "IMEDIT—Editor Program for Use with the Imlac Terminals", MIT Programming Technology Division document "SYS.0E.01", August 1972.

Lisp Machine

Daniel Weinreb and Dave Moon, "Lisp Machine Manual", MIT Artificial Intelligence Lab, March 1981.

SUPDUP

Richard M. Stallman, "The SUPDUP Protocol", MIT Artificial Intelligence Lab memo 644, March 1982. This includes the full details of the Local Editing Protocol and Line Saving Protocol as used in SUPDUP.

TCP

Jon Postel, "DOD Standard Transmission Control Protocol", USC/Information Sciences Institute, IEN-129, RFC 761, NTIS ADA082609, January 1980. Appears in: Computer Communication Review, Special Interest Group on Data Communication, ACM, V.10, N.4, October 1980.

Z

Steven R. Wood, "Z, the 95% Program Editor", in the proceedings of the SIGPLAN conference on text manipulation, June 1981.]