Abstract

The SUPDUP protocol provides for login to a remote system over a network with terminal-independent output, so that only the local system need know how to handle the user's terminal. It offers facilities for graphics and for local assistance to remote text editors. This memo contains a complete description of the SUPDUP protocol in fullest possible detail.

@memo< **Keywords:** Communications, Display, 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 SUPDUP protocol is a superior replacement for the TELNET protocol. Its name means "Super Duper".

The basic purpose of the two protocols is the same: allow a user to log in to a remote system connected to his local system by a network, as if his terminal were directly connected to the remote system. Both protocols define a "virtual terminal" which the remote system is expected to use. The difference is that TELNET's "Network Virtual Terminal" is an obsolete teletype, whereas SUPDUP's virtual terminal is a display, capable (optionally) of powerful operations on text, drawing pictures, and invisibly sharing the work of a text editor running on the computer, all totally under the computer's control.

The cost associated with SUPDUP is that the remote system must be capable of communicating with SUPDUP's virtual terminal. Most operating systems understand nothing more than simple printing terminals, leaving everything else to user programs. The SUPDUP virtual display terminal is not a superset of the simple printing terminal, so these operating systems cannot serve as the remote system for SUPDUP. They can support user programs which make SUPDUP connections to other systems. Since these operating systems have the deficiency that every user program which does display must know the details of operation of each type of terminal to be used, they are inferior anyway; a modern operating system which provides terminal-independent display support will have no trouble supporting SUPDUP, or anything like SUPDUP.

SUPDUP can operate over any pair of streams which transmit 8-bit bytes.

The SUPDUP protocol was created initially by taking the formats of data in the terminal input and output buffers of ITS,[1] and the ITS terminal characteristics words. It has grown by layer on layer of extensions, and ought to be redesigned from scratch for simplicity. If you do not need to communicate with systems that support the existing SUPDUP protocol, you might as well design the low-level details anew. The higher level design decisions of SUPDUP are reasonably good to follow.

All numbers except number of bits are octal unless followed by a decimal point; thus, 12 and 10. are the same. Bits in a word are numbered in decimal from the sign bit (bit 0) to the least significant bit (bit 35, in a 36-bit word). Sizes of words and fields are also decimal.

Data words, and bits and fields within them, are referred to by the names that are conventional on ITS. The names of bits and fields are those provided for assembler language programs on ITS.

1 SUPDUP Initialization

A SUPDUP connection is created using the standard connection-creation mechanism for the network in use. For the Arpanet, this is the ICP. For the Chaosnet, this would be the exchange of an RFC, an OPN and a STS packet. The SUPDUP protocol itself becomes concerned only after this has been done.

The initialization of a SUPDUP connection involves sending terminal characteristics information to the remote system and a greeting message to the local system. The greeting message is ASCII text, one

[1]The Incompatible Timesharing System

character per byte, terminated with a %TDNOP code (see below). It should simply be printed on the console by the local system. The terminal characteristics information is made up of several 36-bit words, which are precisely the internal terminal characteristics words of ITS. Each word is broken up into 6 6-bit bytes, which are sent most significant first, each in one 8-bit byte. The words are

1. A count. The high 18 bits of this word contain minus the number of words that follow. This permits new words of terminal characteristics to be defined without requiring all local-system SUPDUP programs to be changed. The server will supply a default value for any characteristics words defined in the future and not mentioned below.
2. The TCTYP variable. This should always contain 7. If the server is an ITS, this tells it to assume that its terminal is a SUPDUP virtual terminal.
3. The TTYOPT variable. This contains many bits and fields.
4. The height of the screen.
5. The width of the screen minus one. The reason that one is subtracted is that if one column is used (as on ITS) to indicate line continuation, this is the effective width of the screen for the user's text.
6. The number of lines by which the terminal will scroll automatically if an attempt is made to Linefeed past the bottom of the screen. This value is under the control of the physical terminal being used, but the local system is expected to know the value by virtue of knowing what type of terminal is in use. It then informs the remote system by passing this value. For most terminals, this should be 1.
7. The TTYSMT variable. This contains bits and fields that describe the graphics and local editing capabilities of the terminal.

The TTYOPT word contains these bits:

%TOCLC (bit 2)
: The remote system should convert lower case characters to upper case. This is the initial value of a user option. If the local system has such an option as well, the local setting should be propagated. Otherwise, this bit should be zero.

%TOERS (bit 3)
: The terminal can erase selectively; that is, it supports the output commands %TDEOL, %TDDLF, and optionally %TDEOF (see below).

%TOMVB (bit 5)
: The terminal can move its cursor backwards. (When SUPDUP was invented, model 33 teletypes were still in use!)

%TOSAI (bit 6)
: The terminal can handle codes 000 through 037, and 177, as printing characters.

%TOSA1 (bit 7)
: The remote system should output codes 000 through 037, and 177, as themselves, expecting them to be printing characters. This is the initial value of a user-settable option, and it should not be set unless %TOSAI is also set.

%TOOVR (bit 8)
: The terminal can overprint. That is, if two printing characters are output at the same position with no erasing in between, they will both be visible. Any terminal which can move the cursor back and *cannot* overprint will be assumed to handle %TDDLF, since it can do so by means of the sequence Backspace Space Backspace.

%TOMVU (bit 9)
: The terminal can move its cursor upwards. That is, it is a display terminal with some form of cursor control.

%TOMOR (bit 10)
The remote system should pause, in whatever fashion it likes to do so, at the bottom of a screen of output. This is the initial value of a user option, and it should be set to *one*.

%TOROL (bit 11)
The remote system should scroll, as opposed to wrap around, when output goes past the bottom of the screen. This is the initial value of a user option, and can be set whichever way the local system expects its users to prefer. On ITS, the default is to wrap.

%TOLWR (bit 13)
The terminal can generate lower-case characters as input.

%TOFCI (bit 14)
The terminal keyboard has Control and Meta keys and can generate all the meaningful codes of the 12-bit character code (see below).

%TOLID (bit 16)
The terminal can do insert/delete line; it supports the %TDILP and %TDDLP commands.

%TOCID (bit 17)
The terminal can do insert/delete character; it supports the %TDICP and %TDDCP commands.

%TPCBS (bit 30)
The local system is sending 034-escape sequences on input. This bit *must be set*.

%TPORS (bit 32)
The local system should be notified with a %TDORS when the remote system attempts to abort output. This bit *should be set*.

%TPRSC (bit 33)
The local system implements the region-scrolling commands %TDRSU and %TDRSD, which scroll an arbitrary portion of the screen.

The TTYSMT variable contains these bits and fields:

%TQMCH (bits 0-2)
CPU type. 1 indicates a PDP-11. 2 indicates an Imlac PDS1. 3 indicates a PDS4. 0 indicates anything else. 4 through 7 are undefined.

%TQHGT (bits 3-7)
Character height in pixels, for the SUPDUP Graphics Protocol. This is the interline spacing for the font used for ordinary character output. The total usable screen height is this times the screen height in characters which applies to ordinary character output.

%TQWID (bits 8-11)
Character width in pixels, for the font used for ordinary character output. The total usable screen width in pixels is this parameter times the screen width in characters, as used for ordinary character output.

%TQVIR (bit 12)
The local system supports virtual coordinates in the SUPDUP Graphics Protocol.

%TQBNK (bit 13)
The local system supports blinking objects in the SUPDUP Graphics Protocol.

%TQXOR (bit 14)
The local system supports XOR mode in the SUPDUP Graphics Protocol.

%TQREC (bit 15)
The local system supports rectangle commands in the SUPDUP Graphics Protocol.

%TQSET (bit 16)
The local system supports multiple object sets in the SUPDUP Graphics Protocol.

%TQGRF (bit 17)
The local system supports the SUPDUP Graphics Protocol.

%TRGIN (bit 18)
The local system supports graphics input in the SUPDUP Graphics Protocol.

%TRGHC (bit 19)
The local system supports a hardcopy output device in the SUPDUP Graphics Protocol.

%TRLED (bit 20)
The local system supports the Local Editing Protocol.

%TRSCN (bit 21)
The local system supports scan-line output in the SUPDUP Graphics Protocol.

%TRLSV (bits 22-24)
If this value *n* is nonzero, the local system supports line saving. Furthermore, the largest allowed label is 4^n.

%TRANT (bits 25-27)
If this value *n* is nonzero, the local system supports approximately 4^n lines of anticipatory output.

2 SUPDUP Input

The SUPDUP input language provides for the transmission of MIT extended ASCII, described in detail below. This is a 12-bit character code, but not all possible 12-bit codes are assigned a meaning. Some of those that are not are used as special commands by the local editing protocol.

To transmit 12-bit data on an 8-bit connection, it must be encoded. The encoding used is that the 8-bit byte value 034 is an escape code. To transmit the value 034, send two 034's. Codes 200 and up are encoded into sequences starting with 034. Specifically, to encode *m**200+*n*, send 034, *m*+100, *n*. Other sequences starting with 034 serve other purposes, described below.

The %TOFCI bit in TTYOPT should be set if the terminal actually has the ability to generate all the meaningful codes of extended ASCII. If %TOFCI is zero, SUPDUP input is restricted to a subset which is essentially ASCII (but 034 must still be encoded as a sequence of two 034's). The Local Editing Protocol assumes that %TOFCI is one, and uses a 9-bit subset of the 12-bit character set for its editing commands.

A character in the 12-bit character set is composed of a *basic character*, which makes up the bottom 7 bits, and 5 *bucky bits*. The 2000 bit and the 1000 bit are currently unused. The 400 bit stands for the *Meta* shift key, and the 200 bit stands for the *Control* shift key. These two bits can be added to any character, and are significant in editing commands. The 4000 bit is used to extend the set of basic characters.

The 7-bit basic character is usually interpreted as an ASCII character, except that only these codes in the range 0 through 037 are allowed:

- Backspace (code 010)
- Tab (code 011)
- Linefeed (code 012)
- VT (code 013)
- Formfeed (code 014)
- Return (code 015)
- Call (code 032)

- Altmode (code 033)
- Backnext (code 037)

Backspace, Tab, Linefeed, Formfeed, and Return were derived from the ASCII formatting control characters with the same codes, and they will often echo like those ASCII characters.

The 4000 bit gives a non-ASCII interpretation to the basic character. For example, codes 4000 through 4037 are used for additional printing characters for symbols which are not in ASCII. They represent the characters produced, on output, by codes 0 through 37 (which, in the SUPDUP output language, must signify printing characters if they mean anything at all). Local system SUPDUP programs do not need to support these characters, and most do not (so we do not list their pictorial appearances). If the local-system SUPDUP program does provide for output of codes 0 through 37 as printing characters, the %TOSAI bit in TTYOPT should be one. Otherwise, it should not ever send codes 4000 through 4037 as input.

4177 is another additional printing character, which is echoed as code 177 (again, used only if %TOSAI is set). 4101, 4102, 4103 and 4110 are used for four special command keys, which are called *escape*, *break*, *clear* and *help*. Only *help* receives much use.

All other combinations of the 4000 bit with various basic characters are used for special purposes of the protocol, or are undefined. Codes 4105 (4000 plus E) and 4123 (4000 plus S) are used by the Local Editing Protocol. Code 4124 (4000 plus T) is used by the Line Saving Protocol. Codes 4130 (4000 plus X) and 4131 (4000 plus Y) are used by the SUPDUP Graphics Protocol.

A server system that has no application for MIT extended ASCII should convert it into ordinary ASCII as follows:

- Discard all but the low 8 bits of the 12-bit character.
- If the 200 bit is set, clear it, and the 100 bit and the 40 bit. Thus, control-bit plus A is converted into ASCII control-A.

The encoded SUPDUP input consists entirely of byte values less than 200. Byte values 200 and above are used for special commands that are not "terminal input". On ITS, these commands are processed by the network server program, whereas 034 escape sequences are processed by the system's terminal driver.

- The two byte sequence 300 301 says to log out the remote job and break the connection.
- The two byte sequence 300 302 introduces "console location" text. This text is meant for human consumption, and describes the location in some sense of the local system or the user's terminal or both. It is supplied to the remote system so that system can use it to respond to inquiries about the user's location. The console location text should be made up of ASCII printing characters, and is terminated with a zero byte.

3 SUPDUP Output

Once the SUPDUP connection is initialized, all output uses the SUPDUP output language. In this language, codes 0 through 177 are used only to represent printing characters which are expected to advance the cursor one position to the right. Most terminals support only codes 40 through 176, and only those codes will be sent. If the terminal supports other codes as printing characters, that fact is indicated by the %TOSAI bit in TTYOPT. In any case, codes 011 through 015 do *not* have their ASCII formatting significance.

All other output operations are represented by commands with codes from 200 to 377. Some of these commands are followed by extra argument bytes. The contents of an argument byte may be anything from 0 to 377, and its meaning is determined by the command it follows. The traditional names of these commands are six characters long and start with %TD. The commands are:

%TDMOV (code 200)
: Four arguments, the old vertical and horizontal position followed by the new vertical and horizontal position. Move the cursor to the new position, assuming that it was at the specified old position. The local system should forget its opinion of the old cursor position and use what the %TDMOV says. This would be absurd on a display terminal; %TDMOV is used only for printing terminals. For actual displays, %TDMV0 (see below) is used.

%TDEOF (code 202)
: No arguments. Erase to end of screen. The cursor does not move.

%TDEOL (code 203)
: No arguments. Erase to end of line. The cursor does not move.

%TDDLF (code 204)
: No arguments. Erase the character after the cursor. The cursor does not move.

%TDCRL (code 207)
: No arguments. Advance the cursor to the next line and clear it. At the bottom of the screen, this is expected to scroll the text on the screen by a fixed number of lines. The number of lines the terminal will scroll by is not expected to be under the control of the remote system. Instead, the TTYROL terminal characteristics word is used to tell the remote system how many lines the terminal *will* scroll.

%TDNOP (code 210)
: No arguments. Do nothing.

%TDORS (code 214)
: No arguments. This command marks the place in the data stream at which the remote system executed an abort-output operation. See the next section.

%TDQOT (code 215)
: One argument, which should be passed on directly to the terminal. This is used for transmitting 8-bit data, such as a program being downloaded into a terminal attached to the local system.

%TDFS (code 216)
: No arguments. Move the cursor right one position. This could be done just as well with a %TDMV0; %TDFS is used for data compression.

%TDMV0 (code 217)
: Two argument bytes: the vertical position and the horizontal position. Just move the cursor.

%TDCLR (code 220)
: No arguments. Clear the screen and move the cursor to the top left corner.

%TDBEL (code 221)
: No arguments. Ring the terminal's bell.

%TDINI (code 222)
: No arguments. Informs the terminal that the remote system has just been started. This is only sent to hard-wired terminals. Network connections will never receive this command, because no network connections exist when the system is started.

%TDILP (code 223)
: One argument, a number. Insert that many blank lines at the vertical position of the cursor, pushing the following lines down. The cursor does not move.

%TDDLP (code 224)

One argument, a number. Delete that many lines at the vertical position of the cursor and below, moving the following lines up, and creating blank lines at the bottom of the screen. The cursor does not move.

%TDICP (code 225)
: One argument, a number. Insert that many blank character positions after the cursor, pushing following text on the line to the right. The last few positions on the line are pushed off the right margin and discarded. Other text lines are not affected. The cursor does not move.

%TDDCP (code 226)
: One argument, a number. Delete that many character positions after the cursor, moving following text on the line left to fill them up, and creating blank positions at the right margin. Other lines are not affected, and the cursor does not move

%TDBOW (code 227)
: No arguments. Display printing characters that follow as white-on-black.

%TDRST (code 230)
: No arguments. Reset %TDBOW mode and any other such options as may be defined later.

%TDGRF (code 231)
: Variable number of arguments. This introduces a sequence of SUPDUP Graphics Protocol operations. The SUPDUP Graphics Protocol is documented in a separate section below.

%TDRSU (code 232)
: Two arguments, the number of lines in the region to be scrolled, and the number of lines to scroll it by. The region consisting of the specified number of lines, starting with the line that the cursor is on, is scrolled up by an amount specified by the second argument. Lines scrolled out of the top of the region are lost, and blank lines are created at the bottom of the region. This operation is equivalent to a %TDDLP at the top of the region followed by a %TDILP above the place where the blank lines should be created, above the bottom of the region. However, %TDRSU can avoid disturbing the lines below the region even momentarily, which looks better.

%TDRSD (code 233)
: Two arguments. Like %TDRSU but scrolls the region down instead of up.

The Local Editing Protocol introduces several additional %TD codes. See the section on the Local Editing Protocol for details on implementing them.

%TDSYN (code 240)
: Do-Local-Echoing. Two arguments, a resynchronize code from the last resynchronize received, and the number of input characters received by the remote system since that resynchronize. If the synchronization constraint is satisfied, the local system will begin local editing.

%TDECO (code 241)
: Prepare-for-Local-Editing. No arguments. Tells the local system to begin sending resynchronizes, which will enable the remote system to send a %TDSYN.

%TDEDF (code 242)
: Define-Char. Normally followed by two argument bytes which encode a 9-bit input character and a 5-bit function code; the function code specifies the editing action of the input character when the user types it. The bottom seven bits of each byte are taken, and the two are combined (first byte most significant) into a 14-bit number. The top 5 bits of this number are the function code. Function code 37 serves as an escape: a third argument byte follows, which contains the actual function code. This is how function codes larger than 37 are specified.

  The other 9 bits are normally the character whose local editing function is being defined. It is a 12-bit MIT extended ASCII character whose top three bits are taken as zero. This means that not all 12-bit codes can be specified; so the local editing protocol only applies to input characters with codes 777 and below. Any input character which contains the 4000 bit is always handled remotely. (The 1000 and 2000 bits are unused in the 12-bit character set.)

  If the terminal does not have Control and Meta keys, and transmits ASCII (%TOFCI is not set), %TDEDF will still speak in terms of 9-bit characters. Definitions specified for 9-bit characters in the range 300 through 337 should be applied to the ASCII control characters 000 through 037. Definitions for 9-bit characters in the range 040 through 177 should be applied to the ASCII characters with the same values.

A counterintuitive feature of MIT extended ASCII is that Control-A and Control-a are two different characters. Their codes are 301 and 341. Most programs interpret these two characters the same way, and most editors will want to tell the local system to interpret the two the same way. In the Local Editing Protocol this is done by defining the lower-case character with function code 22, which means "use the definition of a related character". For characters whose basic character is a lower case letter, the related character used is the corresponding upper case character. Thus, defining character 341 (Control-a) with code 22 tells the local system to use the definition of character 301 (Control-A). For a non-lower-case character with the Control bit (200) set, the related character is obtained by turning off bits 200 and 100. Thus, Control-I (311) turns into Tab (011, which is ASCII Control-I). Code 22 should be used only with lower case characters and control characters.

Some function codes are special, and do something other than define a character. With these, the 9 bits are used to transmit arguments associated with the operation being performed. Here is how.

Code 31
: Set Word Syntax. The syntax is specified for the ASCII character which is the low 7 bits of the accompanying 9-bit character. The 200 bit of the 9-bit character says what the syntax *is*: if it is 1, the character is a separator. If it is 0, the character is part of a word.

Code 32
: Set Insertion Mode. The 9 bits contain the insertion mode.

Code 33
: Initialize. The word syntax table is set so that only letters and digits are part of words. Insertion mode is set to 1. The fill column is set to 0. All character definitions are set to code 0 except these:

- Lower case letters, with or without Control or Meta, are set to code 22 (use the definition of the related upper case character). These are characters 140 through 172, 340 through 372, 540 through 572, and 740 through 772.
- All printing characters except lower case letters are set to code 7 (self-insert). These are characters 40 through 140, and 173 through 176.
- Digits, with either Control, Meta or both, are set to code 27 (specify repeat count). These are characters 260 through 271, 460 through 471, and 660 through 671.

Code 34
: Set Margin. The top 2 bits of the 9 say which margin to set: 0, 1, 2, 3 stand for left, top, right, bottom. The remaining 7 bits specify that margin, as a distance from the actual screen edge behind it.

Code 41
: Set Fill Column. The 9 bits contain the value of the fill column, or zero if there is to be no fill column.

%TDNLE (code 243)
: Stop-Local-Editing. No arguments.

%TDTSP (code 244)
: Space-for-Tab. No arguments. Used to output spaces which are part of the representation of an ASCII tab character in the text being edited.

%TDCTB (code 245)
: Line-Beginning-Continued. No arguments. States that the beginning of this screen line is not at the beginning of a line of the text being edited.

%TDCTE (code 246)
: Line-End-Continued. No arguments. States that the line of text which appears on this screen line extends beyond the end of the screen line.

%TDMLT (code 247)

Multi-Position-Char. Two arguments, the width of the character about to be displayed, and its character code. The width specifies the number of character positions, following the cursor, which belong to the representation of a single character of the text being edited. This command sequence should be followed immediately by output to fill up those positions.

The Line Saving Protocol introduces four additional commands:

%TDSVL (code 250)

Save-Lines. Three arguments: a number of lines *n*, followed by two 7-bit bytes of label number *l* (low byte first). *n* lines starting with the line the cursor is on have their contents saved under labels *l*, *l*+1, ... *l*+*n*-1.

%TDRSL (code 251)

Restore-Lines. Three arguments bytes, the same as for %TDSVL. The contents saved under *n* labels starting at label *l* are restored onto the screen on lines starting with the one the cursor is on. If any of the labels is not defined, one or more Label-Failure commands are sent to the remote system to report this.

%TDSSR (code 252)

Set-Saving-Range. Two arguments which specify the range of columns of line contents to save and restore in %TDSVL and %TDRSL. The first argument is the first column to save. The second is the last column to save, plus one.

%TDSLL (code 253)

Set-Local-Label. Two arguments, the low and high 7-bit bytes of a label number. This label number, and numbers immediately below it, will be used label the saved contents of any lines pushed off the screen by locally handled editing commands.

One more command is used together with either local editing or line saving for anticipatory output.

%TDMCI (code 254)

Move-Cursor-Invisible. One argument, a logical vertical position which is a 14-bit *signed* number transmitted low 7 bits first. Positions the cursor at an invisible line so that anticipatory text transmission can be done. Output to the invisible line continues until a cursor motion %TD command is received.

The specified vertical position signifies the relationship between the text on this line and that on the screen: if lines 2 through 20 are in use for editing (according to the editing margins), then an invisible line with vertical position 30 contains the text that would appear 10 lines below the last text actually displayed. This information is for the use of locally-handled scrolling commands.

The invisible lines displayed this way will be remembered by the local system for the use of locally handled scrolling commands, if the local system can handle any. Also, the contents can be saved under a label and then restored into actual visibility under the control of the remote editor.

4 Aborting Output in SUPDUP

When a SUPDUP server program is asked to abort (discard) output already transmitted, it sends a %TDORS character. At the same time, it sends an interrupt to the local system if the network between them permits this. The local system should keep a count of the number of such interrupts minus the number of %TDORSes received. When this count is positive, all output received must be discarded and only scanned for %TDORSes. It is legitimate for this count to become negative.

When the %TDORS is read which decrements the count back to zero, the local system must transmit the current cursor position to the remote system. The remote system is waiting for this, because it does not know how much output the local system actually discarded, and therefore it does not know where the terminal cursor ended up. The cursor position is transmitted using a four byte sequence: 034 020 *vpos hpos*. The remote system should hold all output and not transmit any more until this command is received.

If the network does not provide for interrupts (high-priority messages), the local system should send the cursor position whenever a %TDORS is received.

5 Non-Network Terminals

SUPDUP can also be used to communicate with directly wired or dial-up terminals, which we call non-network terminals. The key point of distinction is that "networks" provide buffering, flow control and error recovery, so that SUPDUP as used over networks need not be concerned with these things. When SUPDUP is used with non-network terminals, they must all be provided for.

The buffering is provided in the remote system’s terminal controller and in the terminal itself. The flow control, and associated error recovery, are provided by an extension of the SUPDUP protocol. This flow control mechanism was inspired by TCP [see TCP] and is immeasurably superior to the XON/XOFF mechanism by which many terminals deprive the user of the use of two valuable editing commands.

Flow control works by means of an *allocation*, stored in the remote system, which is the number of characters which the remote system can send to the terminal without overloading it. After this many have been sent, the remote system must cease transmission until it is sent an input command giving more allocation. The terminal sends this command when it has made room in its buffer by processing and removing characters from it.

To begin using flow control, the terminal should send the sequence 034 032, which initializes the allocation to zero. Then it should send the sequence 034 001 *n*, where *n* is the number of characters of buffer space available for output. This increments the allocation by *n*. Several 034 001 *n* sequences can be used to set the allocation to a value above 177.

The remote system should remember the allocation it has received and decrement it each time a character is transmitted. The terminal should count the number of output characters processed and removed from the terminal’s buffer for output; every so often, another 034 001 *n* sequence should be sent containing the number of characters processed since the last 034 001 *n*. For example, the terminal could send 034 001 030 each time it processes another 30 characters of output. By sending this sequence, it gives the remote system permission to fill up that part of the buffer again.

This flow control mechanism has the advantage that it introduces no delay if the buffering capacity does not become full, while never requiring the remote system to respond quickly to any particular input signal. An even greater advantage is that it does not impinge on the character set available for input from the terminal, since it uses the 034 escape mechanism, and 034 can be transmitted with the sequence 034 034.

Because line noise can cause allocate commands to be garbled, if the terminal receives no output for a considerable period of time, it should send another 034 032 followed by 034 001 *n*. This will make sure that the remote system and the terminal agree on the allocation. This is the only error recovery provided. To achieve error recovery for data as well, one must essentially implement a network protocol over the line. That is beyond the scope of SUPDUP.

ITS performs an additional encoding on SUPDUP output data over non-network lines. It encodes the data into 7-bit bytes so that it can be sent through old terminal controllers which cannot output 8-bit bytes. Other systems should not need to use this encoding. Codes 0 through 176 are transmitted as is. Code 177 is transmitted as 177 followed by 1. Code 2*nn* is transmitted as 177 followed by *nn*+2. All cursor positions are transmitted with 1 added to them to reduce the frequency with which bytes containing zero must be sent; this is also to avoid problems with old controllers.

The 300-sequences may not be used as input and the terminal should not send them. This is because they are normally processed by the network server, which is not in use for a non-network line.

SUPDUP Graphics Protocol

The SUPDUP Graphics Protocol allows pictures to be transmitted and updated across an ordinary SUPDUP connection, requiring minimal support from the remote operating system. Its features are

- It is easy to do simple things.
- Any program on the server host can at any time begin outputting pictures. No special preparations are needed.
- No additional network connections are needed. Graphics operations go through the normal text output connection.
- It does not require a network. It is suitable for use with locally connected intelligent display terminals, and by programs which need not know whether they are being used locally or remotely. It can be the universal means of expression of terminal graphics output, for whatever destination.
- Loss or interpolation of output, due to a "silence" command typed by the user or to the receipt of a message, does not leave the local system confused (though it may garble the interrupted picture itself).
- The local system is not required to remember the internal "semantic" structure of the picture being displayed, but just the lines and points, or even just bits in a bit matrix.
- Like the rest of the SUPDUP protocol, SUPDUP Graphics is terminal-independent.

The terminal capabilities associated with the SUPDUP Graphics Protocol are described by bits and fields in the TTYSMT variable, described above. One of these bits is %TQGRF, which indicates that the local system is able to handle the SUPDUP Graphics Protocol in the first place.

Graphics output is done with one special %TD command, %TDGRF. This command is followed by any number of graphics operations, which are characters in the range 0 through 177. These are just the ordinary printing characters, but in graphics mode they receive a special interpretation described below. Characters in their role as graphics operations have symbolic names starting with %GO, such as %GOMVA and %GOELA. Some graphics operations themselves take arguments, which follow the operations and are composed of more characters in the range 0 through 177.

Any character 200 or above leaves graphics mode and then is interpreted normally as a %TD command. This way, the amount of misinterpretation of output due to any interruption of a sequence of graphics operations is bounded. Normally, %TDNOP is used to terminate a sequence of graphics operations.

Some other %TD commands interact with SUPDUP Graphics. The %TDRST command should reset all graphics modes to their normal states, as described below. %TDCLR resets some graphics state information.

1 Graphics Coordinates

Graphics mode uses a cursor position which is remembered from one graphics operation to the next while in graphics mode. The graphics mode cursor is not the same one used by normal type-out: graphics protocol operations have no effect on the normal type-out cursor, and normal type-out has no effect on the graphics mode cursor. In addition, the graphics cursor's position is measured in pixels rather than in characters. The relationship between the two units (pixels and characters) is recorded by the %TQHGT and %TQWID fields of the TTYSMT variable of the terminal, which contain the height and width in pixels of the box occupied by a normal-size character. The size of the screen in either dimension is assumed to be the size of a character box times the number of characters in that direction on the screen. If the screen is actually bigger than that, the excess may or may not be part of the visible area; the remote system will not know that it exists, in any case.

Each coordinate of the cursor position is a 14-bit signed number, with zero at the center of the screen. Positive coordinates go up and to the left. If the screen dimension is even, the visible negative pixels extend one unit farther than the positive ones, in proper two's complement fashion. Excessively large values of the coordinates will be off the screen, but are still meaningful. These coordinates in units of pixels are called *physical coordinates*.

Local systems may optionally support *virtual coordinates* as well. A virtual coordinate is still a 14-bit signed number, but instead of being in units of physical pixels on the terminal, it is assumed that +4000 octal is the top of the screen or the right edge, while -4000 octal is the bottom of the screen or the left edge. The terminal is responsible for scaling virtual coordinates into units of pixels. The %TQVIR bit in the TTYSMT variable indicates that the local system supports virtual coordinates. When the remote system wants to use virtual coordinates, it should send a %GOVIR; to use physical coordinates again, it should send a %GOPHY. For robustness, every %TDGRF should be followed by a %GOVIR or %GOPHY right away so that following commands will be interpreted properly even if the last %GOVIR or %GOPHY was lost.

The virtual coordinates are based on a square. If the visible area on the terminal is not a square, then the standard virtual range should correspond to a square around the center of the screen, and the rest of the visible area should correspond to virtual coordinates just beyond the normally visible range.

Graphics protocol operations take two types of cursor position arguments, absolute ones and relative ones. Operations that take position arguments generally have two forms, one for relative position and one for absolute. A relative address consists of two offsets, delta-X and delta-Y, from the old cursor position. Each offset is a 7-bit two's complement number occupying one character. An absolute address consists of two coordinates, the X followed by the Y, each being 14 bits distributed among two characters, with the less significant 7 bits in the first character. Both types of address set the running cursor position which will be used by the next address, if it is relative.

Relative addresses are provided for the sake of data compression only. They do not specify a permanent constraint between points; the local system is not expected to have the power to remember objects and constraints. Once an object has been drawn, no record should be kept of how the cursor positions were specified.

Although the cursor position on entry to graphics mode remains set from the last exit, it is wise to reinitialize it with a %GOMVA operation before any long transfer, to limit the effects of lost output.

It is perfectly legitimate for parts of objects to go off the screen. What happens to them is not terribly important, as long as it is not disastrous, does not interfere with the reckoning of the cursor position, and does not cause later objects, drawn after the cursor moves back onto the screen, to be misdrawn.

2 Graphics Operations

All graphics mode operations have codes between 0 and 177, and symbolic names which start with %GO. Operations fall into three classes: draw operations, erase operations, and control operations. The draw operations have codes running from 100 to 137, while the erase operation codes run from 140 to 177. The control operations have codes below 100.

If an operation takes a cursor position argument, the 20 bit in the operation code usually says whether the cursor position should be relative or absolute. The 20 bit is one for an absolute address.

Operations to draw an object always have counterparts which erase the same object. On a bit matrix terminal, erasure and drawing are almost identical operations. On a display list terminal, erasure involves searching the display list for an object with the specified characteristics and deleting it from the list. Thus, on such terminals you can only count on erasing to work if you specify the object to be erased exactly the same as the object that was drawn. Any terminal whose %TOERS bit is set must be able to erase to at least this extent.

For example, there are four operations on lines. They all operate between the current graphics cursor position and the specified cursor position argument:

- %GODLR (code 101) draw line, relative address.
- %GODLA (code 121) draw line, absolute address.
- %GOELR (code 141) erase line, relative address.
- %GOELA (code 161) erase line, absolute address.

Here is how to clear the screen and draw one line.

```
%TDRST                  ;Reset all graphics modes.
%TDGRF                  ;Enter graphics.
%GOCLR                  ;Clear the screen.
%GOMVA xx yy            ;Set cursor.
%GODLA xx yy            ;Draw line from there.
%TDNOP                  ;Exit graphics.
```

Graphics often uses characters. The %GODCH operation is followed by a string of characters to be output, terminated by a zero. The characters must be single-position printing characters. On most terminals, this limits them to ASCII graphic characters. Terminals with %TOSAI set in the TTYOPT variable allow all characters 0-177. The characters are output at the current graphics cursor position (the lower left hand corner of the first character’s rectangle being placed there), which is moved as the characters are drawn. The normal type-out cursor is not relevant and its position is not changed. The cursor position at which the characters are drawn may be in between the lines and columns used for normal type-out. The %GOECH operation is similar to %GODCH but erases the characters specified in it. To clear out a row of character positions on a bit matrix terminal without having to respecify the text, a rectangle operation may be used.

3 Graphics Input

The %TRGIN bit in the right half of the TTYSMT variable indicates that the terminal can supply a graphic input in the form of a cursor position on request. Sending a %GOGIN operation to the terminal asks to read the cursor position. It should be followed by an argument character that will be included in the reply, and will serve to associate the reply with the particular request for input that elicited it. The reply should have the form of a Top-Y character (code 4131), followed by the reply code character as just described, followed by an absolute cursor position. Since Top-Y is not normally meaningful as input, %GOGIN replies can be distinguished reliably from keyboard input.

Unsolicited graphic input should be sent using a Top-X instead of a Top-Y, so that the program can distinguish the two. Instead of a reply code, for which there is no need, the terminal should send an encoding of the buttons pressed by the user on his input device. For a three-button mouse, the low two bits contain the number of the button (1, 2 or 3 from left to right), and the next two bits contain the number of times the button was clicked.

4 Sets

The local system may optionally provide the feature of grouping objects into sets. Then the remote system can blink or move all the objects in a set without redrawing them all. Sets are not easily implemented on bit matrix terminals, which should therefore ignore all set operations (except for a degenerate interpretation in connection with blinking, if that is implemented). The %TQSET bit in the TTYSMT variable of the terminal indicates that the terminal implements multiple sets of objects.

There are up to 200 different sets, each of which can contain any number of objects. At any time, one set is selected; objects drawn become part of that set, and objects erased are removed from it. An object in another set cannot be erased without selecting that set. %GOSET is used to select a set. It is followed by a character whose code is the number of the set to be selected.

A set can be made temporarily invisible, as a whole, using the %GOINV operation, without being erased or its contents being forgotten; and it can then be made instantly visible again with %GOVIS. %GOBNK makes the whole set blink. %GOCLS erases and forgets all the objects in the current set.

Also, a whole set can be moved with %GOMSR or %GOMSA. A set has at all times a point identified as its "center", and all objects in it are actually remembered relative to that center, which can be moved arbitrarily, thus moving all the objects in the set at once. Before beginning to use a set, therefore, one should "move" its center to some absolute location. Set center motion can easily cause objects in the set to move off screen. When this happens, it does not matter what happens temporarily to those objects, but their "positions" must not be forgotten, so that undoing the set center motion will restore them to visibility in their previous positions.

On a terminal which supports multiple sets, the %GOCLR operation should empty all sets and mark all sets "visible" (perform a %GOVIS on each one). So should a %TDCLR SUPDUP command. Thus, any program which starts by clearing the screen will not have to worry about initializing the states of all sets.

Probably only display list systems will support sets. A sufficiently intelligent bit matrix terminal can provide all the features of a display list terminal by remembering display lists which are redundant with the bit matrix, and using them to update the matrix when a %GOMSR or %GOVIS is done. However, most bit matrix terminals are not expected to go to such lengths.

5 Blinking

The %TQBNK bit in TTYSMT indicates that the terminal supports blinking on the screen. Usually blinking is requested in terms of sets: the operation %GOBNK means make the selected set blink. All objects in it already begin blinking, and any new objects drawn in that set also blink. %GOVIS or %GOINV cancels the effect of a %GOBNK, making the objects of the set permanently visible or invisible.

Implementing blinking in terms of sets is convenient, but causes a problem. It is not very hard for an intelligent bit matrix terminal to implement blinking for a few objects, if it is told told in advance, before the objects are drawn. Supporting the use of sets in general is much harder. To allow bit matrix terminals to support blinking without supporting sets fully, we provide a convention for the use of %GOBNK which works with a degenerate implementation of sets.

For the remote system, the convention is to do all non-blinking output in set 0 and all blinking output in set 1, and always send the %GOBNK *before* drawing an object which is to blink.

For the bit matrix terminal, which offers %TQBNK but not %TQSET, the convention is that %GOBNK is interpreted as meaning "objects drawn from now on should blink", and %GOSET is interpreted as meaning "objects drawn from now on should not blink". The argument of the %GOSET is ignored, since sets are not implemented.

When a remote system that is following the convention draws blinking objects, it will send a %GOSET to set 1 (which has no effect), a %GOBNK (which enters blinking mode), draw operations for the blinking objects, and finally a %GOSET to set 0 (which turns off blinking for objects drawn from now on). If the same sequence of commands is sent to a terminal which does fully support sets, the normal definitions of %GOBNK and %GOSET will produce the same behavior.

Erasing a blinking object should make it disappear, on any terminal which implements blinking. On bit matrix terminals, blinking *must* be done by XORing, so that the non-blinking background is not destroyed.

%GOCLS, on a terminal which supports blinking but not sets, should delete all blinking objects. Then, the convention for deleting all blinking objects is to select set 1, do a %GOCLS, and reselect set 0. This has the desired effect on all terminals. This definition of %GOCLS causes no trouble on non-set terminals, since %GOCLS would otherwise be meaningless to them.

To make blinking objects stop blinking but remain visible is possible with a %GOVIS on a terminal which supports sets. It might seem natural that %GOVIS on non-set terminals should just make all blinking objects become solid, but this does not work out cleanly. For example, what would the terminal do if another %GOBNK is output? To be compatible with terminals that implement sets, the formerly blinking objects would all have to be remembered. This is an undesirable burden for the simple blinking terminal, so we do not require it, and the remote system should use %GOVIS only if sets are actually supported.

6 Bit Map Displays: Rectangles, Scan Lines and XOR Mode

Bit matrix terminals have their own operations that display list terminals cannot duplicate. First of all, they have XOR mode, in which objects drawn cancel existing objects when they overlap. In this mode, drawing an object and erasing it are identical operations. All %GOD.. operations become equivalent to the corresponding %GOE..'s. XOR mode is entered with a %GOXOR and left with a %GOIOR. Display list terminals that do not implement XOR mode will ignore %GOXOR and %GOIOR; therefore, programs originating graphics output should continue to distinguish draw operations from erase operations even in XOR mode. %TQXOR indicates a terminal which implements XOR mode. XOR mode, when set, remains set even if graphics mode is left and re-entered. However, it is wise to re-specify it from time to time, in case output is lost.

Bit matrix terminals can also draw solid rectangles. They can thus implement the operations %GODRR, %GODRA, %GOERR, and %GOERA. A rectangle is specified by taking the current cursor position to be one corner, and providing the address of the opposite corner. That can be done with either a relative address or an absolute one. The %TQREC bit indicates that the terminal implements rectangle operations.

Finally, bit matrix terminals can handle data in the form of scan-lines of bits. The %TRSCN bit in TTYSMT indicates the presence of this capability. This is done with the operations %GODSC, %GOESC, %GODRN and %GOERN. The first two draw and erase given actual sequences of bits as arguments. The last two take run-length-encoded arguments which say "so many ones, then so many zeros", for data compression.

Scan line data for %GODSC and %GOESC is grouped into 16-bit units, and each 16-bit unit is broken into three argument characters for transmission. (Three characters are needed since seven bits at most can be used in each one). The first two argument characters transmit six bits each, and the third transmits four bits. The most significant bits of the 16-bit unit are transmitted first; the least significant four or six bits of the argument character are used. The end of the scan line data is indicated by 100 (octal) as an argument.

When drawing, a one bit means turn on the corresponding pixel, and a zero bit means leave the pixel alone. This is so that pictures output with scan lines can overstrike with other things. Similarly, when erasing, clear pixels for one-bits and do not clear them for zero-bits. In XOR mode, flip pixels for one-bits and leave alone the other pixels.

Operations %GODRN and %GOERN use run-length-encoded arguments instead of actual patterns of bits. Each argument character specifies a number of consecutive one bits or a number of consecutive zero bits. Codes 1 through 77 indicate a number of zero bits. Codes 101 through 177 indicate a number of one bits. An argument of zero terminates the %GODRN or %GOERN. As with the scan operations, a one bit means operate on the corresponding pixel, and a zero bit means do nothing to it.

The scan line operations do not usually go well with virtual coordinates; you cannot escape the fact that you are dealing with the pixel size of the display when you are controlling individual pixels.

7 Using Only Part of the Screen

It is sometimes desirable to use part of the screen for picture and part for text. Then one may wish to clear the picture without clearing the text. On display list terminals, %GOCLR should do this. On bit matrix terminals, however, %GOCLR can't tell which bits were set by graphics and which by text display. For their sake, the %GOLMT operation is provided. This operation takes two cursor positions as arguments, specifying a rectangle. It declares that graphics will be limited to that rectangle, so %GOCLR should clear only that part of the screen. %GOLMT need not do anything on a terminal which can remember graphics output as distinct from text output and clear the former selectively, although it would be a desirable feature to process it even on those terminals.

%GOLMT can be used to enable one of several processes which divide up the screen among themselves to clear only the picture that it has drawn, on a bit matrix terminal. By using both %GOLMT and distinct sets, it is possible to deal successfully with almost any terminal, since bit matrix terminals will implement %GOLMT and display list terminals almost always implement sets.

The %TDCLR operation should clear the whole screen, including graphics output, ignoring the graphics limits.

8 Graphics Output by Multiple Processes

It may be useful for multiple processes, or independent programs, to draw on the screen at the same time without interfering with each other.

If we define "input-stream state" information to be whatever information which can affect the action of any operation, other than what is contained in the operation, then each of the several processes must have its own set of input-stream state variables.

This is accomplished by providing the %GOPSH operation. The %GOPSH operation saves all such input-stream information, to be restored when graphics mode is exited. If the processes can arrange to output blocks of characters uninterruptibly, they can begin each block with a %GOPSH followed by operations to initialize the input-stream state information as they desire. Each block of graphics output should be ended by a %TDNOP, leaving the terminal in its "normal" state for all the other processes, and at the same time popping the what the %GOPSH pushed.

The input-stream state information consists of:

- The cursor position.
- the state of XOR mode (default is OFF).
- the selected set (default is 0).
- the coordinate unit in use (physical pixels, or virtual) (default is physical).
- whether output is going to the display screen or to a hardcopy device (default is to the screen).
- what portion of the screen is in use (see "Using Only Part of the Screen") (default is all).

Each unit of input-stream status has a default value for the sake of programs that do not know that the information exists; the exception is the cursor position, since all programs must know that it exists. A %TDINI or %TDRST command should set all of the variables to their default values.

The state of the current set (whether it is visible, and where its center is) is not part of the input-stream state information, since it would be hard to say what it would mean if it were. Besides, the current set number is part of the input-stream state information, so different processes can use different sets. The allocation of sets to processes is the server host's own business.

9 Errors in Graphics Operations

Errors in graphics operations should be ignored by the local system. This is because there is no simple way to report an error well enough to be useful. Since the output and input streams are not synchronized, the remote system would not be able to tell which of its operations caused the error. No report is better than a useless report.

Errors which are not the fault of any individual operation, such as running out of memory for display lists, should also be ignored as much as possible. This does *not* mean completely ignoring the operations that cannot be followed; it means following them as much as possible: moving the cursor, selecting sets, etc. as they specify, so that any subsequent operations which can be executed are executed as intended.

10 Storage of Graphics Operations in Files

Since graphics mode operations and their arguments are all in the range 0 through 177, it is certainly possible to store them in files. However, this is not as useful as one might think.

Any program for editing pictures of some sort probably wants to have a way of storing the pictures in files, but graphics mode operations are probably not best for the application. This is because the program presumably works with many kinds of information such as constraints which are only heuristically deduceable from actual appearance of the picture. A good representation must explicitly provide a place for this information. Inclusion of actual graphics operations in a file will be useful only when the sole purpose of the file is to be displayed.

11 Graphics Draw Operations

Note: the values of these operations are represented as 8-bit octal bytes. Arguments to the operations are in lower case inside angle brackets. *p* represents a cursor position (absolute or relative, according to the operation description).

%GODLR (code 101) *p*
: Draw line relative, from the cursor to *p*.

%GODPR (code 102) *p*
: Draw point relative, at *p*.

%GODRR (code 103) *p*
: Draw rectangle relative, corners at *p* and at the current cursor position. Only if %TQREC is set in TTYSMT.

%GODCH (code 104) *string* 0
: Display the chars of *string* starting at the current graphics cursor position.

%GODSC (code 105) *scan-bits scan-terminator*
: Draw scan bits starting at the current graphics cursor position. Only if %TRSCN is set in TTYSMT.

%GODRN (code 106) *run-length-encodings* 0
: Draw bits determined by *run-length-encodings* starting at the current graphics cursor position. Only if %TRSCN is set in TTYSMT.

%GODLA (code 121) *p* Draw line absolute, from the cursor to *p*.
This code does the same thing as %GODLR, except that the cursor position argument is absolute.

%GODPA (code 122) *p*
Draw point absolute, at *p*.

%GODRA (code 123) *p*
Draw rectangle absolute, corners at *p* and at the current cursor position. Only if %TQREC is set in TTYSMT.

12 Graphics Erase Operations

%GOELR (code 141) *p*
Erase line relative, from the cursor to *p*.

%GOEPR (code 142) *p*
Erase point relative, at *p*.

%GOERR (code 143) *p*
Erase rectangle relative, corners at *p* and at the current cursor position. Only if %TQREC is set in TTYSMT.

%GOECH (code 144) *string* 0
Erase the chars of *string* starting at the current graphics cursor position.

%GOESC (code 145) *scan-bits scan-terminator*
Erase scan bits starting at the current graphics cursor position. Only if %TRSCN is set in TTYSMT.

%GOERN (code 146) *run-length-encodings* 0
Erase bits determined by *run-length-encodings* starting at the current graphics cursor position. Only if %TRSCN is set in TTYSMT.

%GOELA (code 161) *p*
Erase line absolute, from the cursor to *p*.

%GOEPA (code 162) *p*
Erase point absolute, at *p*.

%GOERA (code 163) *p*
Erase rectangle absolute, corners at *p* and at the current cursor position. Only if %TQREC is set in TTYSMT.

13 Graphics Control Operations

%GOMVR (code 001) *p*
Move cursor to point *p*

%GOMVA (code 021) *p*
Move cursor to point *p*, absolute address.

%GOXOR (code 002)
Turn on XOR mode. Only if %TQXOR is set in TTYSMT.

%GOIOR (code 022)
Turn off XOR mode. Only if %TQXOR is set in TTYSMT.

%GOSET (code 003) *n*
Select set. *n* is a 1-character set number, 0 - 177. Only if %TQSET is set in TTYSMT, except for a degenerate interpretation when %TQBNK is set and %TQSET is not.

%GOMSR (code 004) *p*
Move set origin to *p*. Only if %TQSET is set in TTYSMT.

%GOMSA (code 024) *p*

Move set origin to *p*, absolute address. Only if %TQSET is set in TTYSMT.

%GOINV (code 006)
Make current set invisible. Only if %TQSET is set in TTYSMT.

%GOVIS (code 026)
Make current set visible. Only if %TQSET is set in TTYSMT.

%GOBNK (code 007)
Make current set blink. Canceled by %GOINV or %GOVIS. Only if %TQBNK is set in TTYSMT.

%GOCLR (code 010)
Erase the graphics portion of the screen. If possible, erase all graphics output but not normal output.

%GOCLS (code 030)
Erase entire current set. Only if %TQSET is set in TTYSMT, except for a degenerate interpretation if %TQBNK is set and %TQSET is not.

%GOPSH (code 011)
Push all input-stream status information, to be restored when graphics mode is exited.

%GOVIR (code 012)
Start using virtual coordinates. Only if %TQVIR is set in TTYSMT.

%GOPHY (code 032)
Resume giving coordinates in units of pixels.

%GOHRD (code 013) *n*
Divert output to output subdevice *n*. *n*=0 reselects the main display screen. Only if %TQGHC is set in TTYSMT.

%GOGIN (code 014) *n*
Request graphics input (mouse, tablet, etc). *n* is the reply code to include in the answer. Only if %TQGIN is set in TTYSMT.

%GOLMT (code 015) *p1 p2*
Limits graphics to a subrectangle of the screen. %GOCLR will clear only that area. The cursor positions are both absolute!

Local Editing Protocol

The purpose of the Local Editing Protocol is to improve the response time in running a display text editor on the remote system through SUPDUP. It 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.

Bits in the TTYSMT word tell the remote system that the local system supports one or both of these subprotocols of SUPDUP. It is up to the remote system to use them or not.

Any local editing protocol must accomplish these things:

1. 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.
2. 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.
3. The local system must be able to verify for certain that all the input it has sent to the remote system has been processed, and all the resulting output has been received by the local system. We call this process "synchronization". To attempt local editing if this condition is not met would cause timing errors and paradoxical results.
4. The local system must be able to tell the remote system in some fashion about any editing that has been performed locally.
5. 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.

0.1 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 %TD commands plus some conventions as to how the existing %TD commands are used by remote editors are enough to allow the contents of the edited text to be deduced.

The local system must not only obey these commands but also keep a record of the characters on the screen suitable for executing the user's 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. Some parts of the screen may be reserved for such things as status information, command echoing, or other editing windows. 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 a output command Space-for-Tab (%TDTSP). 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, %TDMLT *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: %TDMLT 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. Text lines that run over the ends of screen lines are said to be *continued*. To deal with continued lines, we define two new output commands, %TDCTB and %TDCTE 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, %TDCTE on one line would always accompany %TDCTB on the next, but this might not be appropriate for other editors. Both of these commands must set bits that accompany the lines when they are moved up or down on the screen. The %TDCTE bit should be cleared by anything which erases the end of the line, and by %TDDCP within the line. The %TDCTB bit should be cleared by anything which erases the entire line. EMACS always outputs a %TDCTB for the first line on the screen, since EMACS does not attempt to think about the text before what appears on that 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 kind of %TDEDF (see below).

5. Space characters at the end of a line are distinguished by means of conventions for the use of the erasing commands %TDEOL, %TDCLR and %TDDLF. The convention is that %TDDLF 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 %TDEOL or %TDCLR 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 %TDICP 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 %TDDCP 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 %TDCTE).

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. (Note that this can only be relied upon to be the end of a line of text if %TDCTE has not been done on this screen line).

0.2 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. In SUPDUP local editing, %TDEDF is used for this. 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.

A %TDEDF 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 %TDEDF commands. On subsequent occasions, %TDEDF commands need only be used for characters whose meanings have changed.

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, and the insertion mode.

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 %TDEDF command (code 31) is used to specify the word syntax bit for one text character. See below.

EMACS has an optional mode in which lines are broken automatically, at spaces, when they get too long. In such a feature, certain characters[2] 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* (code 40). The fill column is a parameter whose purpose is to specify where on the line these characters should cease to be handled locally. A special form of %TDEDF (code 41) sets the value of the fill column.

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 %TDEDF command (code 32) 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

[2] In EMACS, the Space character

mode can be handled locally, though we have not actually defined any such function.

Another special form of %TDEDF (code 34) is used to set the editing margins, which say what part of the screen is used for local editing.

Another special form of %TDEDF command (code 33) initializes the definitions of all user input characters, all the word syntax table bits, the fill column 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 %TDEDF commands needed by first using code 33 to initialize all characters and then fixing up those command definitions and parameters whose standard initial states are not right.

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

0.2.1 %TDEDF Function Codes

Here is a list of function codes (all in octal) and their meanings, plus implementation directions.

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.

4 Delete the previous character. Do not handle the command locally at the beginning of 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.

11 Similar, but move down instead of up. Do not handle locally if either the line the cursor starts on, or the following line, is continued at the beginning.

12 Like code 10, but treat tabs just like so many spaces.

13 Like code 11, but move down.

14 Move cursor to beginning of previous line. Must not be handled locally if either the current line or the previous line is continued at the beginning.

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

16 Insert a line-break at the cursor and move the cursor after it. Should not be handled locally if the cursor horizontal position is greater than or equal to the fill column (plus the left editing margin).

17 Insert a line-break at the cursor but do not move the cursor. Does not pay attention to the fill column either.

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

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

22 Use the definition of another character whose code is determined from this character. The details and purpose of this function are explained in the section on .

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.

24 Move cursor to the beginning of the previous word. Do not handle locally if the word appears to begin at the beginning of the line and the line is marked as continued at the beginning.

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

26 Delete from cursor to the beginning of the previous word. Do not handle locally on a line marked as continued at the end, or when code 24 would not be handled locally.

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.

30 Introduces digits which make up an argument. The following characters, as long as they are digits (or a leading minus sign), should be saved and used as a repeat count for the first character which is not a digit or minus sign.

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. See the section on SUPDUP output, under %TDEDF, for further information.

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.

33 This function code ignores the accompanying character and initializes the definitions of all command characters, as well as the word syntax table and insertion mode. See the section on SUPDUP output, under %TDEDF, for further information.

34 Set margin. This function code specifies one of the editing margins. See the section on SUPDUP output, under %TDEDF, for further information.

35 Move cursor up vertically, ignoring continuation. Like code 10, but handle locally regardless of whether lines are continued.

36 Move cursor down vertically, ignoring continuations. Like code 35 but move down.

40 Self-insert, within the 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. Otherwise, certain function codes are not handled when the cursor horizontal position is greater than the fill column (plus the current left editing margin). Function codes affected are 16, 40 and 42.

42 Insert line-break, moving over it, except that if the cursor is at the end of a line which is not continued, and a blank line follows, move the cursor onto the blank line. Should not be handled locally if the current cursor column is greater than or equal to the fill column parameter (plus the left editing margin).

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 or anticipatory output.

44 Scroll down. Similar to 43 but scrolls down instead of up.

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.

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.

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.

0.3 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 %TDECO (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 %TDEDF commands as necessary, it sends a %TDSYN (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 %TDSYN 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 editors 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.

The first %TDSYN after a %TDECO is usually preceded by many %TDEDF commands to initialize all the editing commands. Later %TDSYN commands are preceded by %TDEDFs only for commands whose meanings have changed.

Since the character count in a %TDSYN 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 %TDECO.

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 %TDSYN.

0.4 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, preceded by a special marker which identifies them to the remote system as locally handled. The special marker two characters long; it consists of the extended ASCII character 4105 (4000 plus E) followed by a byte containing the number of locally handled editing characters that follow. Thus, a locally handled sequence X Z Control-B Y would be sent as 4105 4 130 132 302 131.

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.

0.5 Unexpected Output From Remote System

If output is received from the remote system while local editing is going on, the local system should stop doing local editing, and should not send any more resynchs until another %TDECO is received. If the remote system generates output that the text editor does not know about, it should notify the text editor to send another %TDECO.

The remote system can send the command %TDNLE (Stop-Local-Editing) as a clean way of stopping local editing. It has the same effect.

The Line Saving Protocol

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 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.

1 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 label number is 14 bits wide (transmitted as two bytes of 7 bits, low bits first), but local systems are not required to be able to handle 2^{14} different labels. The %TRLSV field in the TTYSMT variable tells the remote system approximately how many different labels the local system can remember. The remote system should not try to use a larger label number. The local system 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).

Lines are saved with the command %TDSVL (Save-Lines). It takes three bytes of arguments: one for the number of lines to save, and two for the label. Several consecutive lines are saved under consecutive labels, starting with the line the cursor is on and the specified label, then moving downward and incrementing the label number.

To restore lines, the remote system should send a %TDRSL (Restore-Lines) command. %TDRSL takes arguments just like %TDSVL, but it copies the saved text belonging to the labels back onto the screen. The meanings of the labels are forgotten when the labels are restored.

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

Line contents should be saved and restored by actual copying; or by some other technique with equivalent results.

If the local system is asked to restore a label for which there is no record, it should send to the remote system the sequence 4124 (4000 plus T) followed by the number of lines (one byte) and starting label (two bytes). This is called a Label-Failure sequence. The remote system, on receiving this, should record that those labels are no longer available and reoutput the contents of those lines from scratch. In order to implement this, the remote system must remember which lines it attempted to restore from which labels.

2 Interaction between Local Editing and Line Saving

Some local editing commands can push lines off the screen. If Line Saving is in use as well, the remote system can ask for such lines to be saved with labels in case they are useful later.

The command %TDSLL (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 %TDRSL command.

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

@memo<

>

3 Anticipatory Output

The remote system can use the time when the user is idle to send and label lines which are not needed on the screen now but might be useful later. Such *anticipatory output* can be brought onto the screen by a %TDRSL command from the remote system or by a locally-handled scrolling command.

To begin a line of anticipatory output, the remote editor sends a %TDMCI (Move-Cursor-Invisible) command, which is followed by a 14-bit signed number called the logical vertical position (transmitted as two 7-bit bytes, low bits first). 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 %TDMCI should not appear on the screen. The terminal's actual cursor should remain where it was before the %TDMCI.

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, or moving the cursor onto the screen, should do that. Output which is actually unrelated to the editor ought to start with a %TDMV0 and will be detected by this test.

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.

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.

The %TRANT field of the TTYSMT word tells the remote system 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.

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

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.

Table of Contents