HONEYWELL

MULTICS TEXT EDITOR (TED) REFERENCE MANUAL

SOFTWARE

MULTICS TEXT EDITOR (TED) REFERENCE MANUAL

SUBJECT

Introduction and Description of the Multics Ted Text Editor

SPECIAL INSTRUCTIONS

This manual presupposes some basic knowledge of the Multics system provided by the 2-volume set, *New Users' Introduction to Multics* – Part I, (Order No. CH24) and Part II, (Order No. CH25), referred to in text as New Users' Introduction.

SOFTWARE SUPPORTED

Multics Software Release 9.0

ORDER NUMBER

CP50-00

December 1981

Honeywell

PREFACE

Throughout this document, the convention used in discussing the editor is as follows:

TED when used on title pages, covers, and when used for section title.

Ted when referring to the "subsystem" invoked by the command (i.e., Ted refers to the Text EDitor).

ted when referring to the command that invokes the editor.

This manual provides a detailed description of Ted including all the necessary information to edit text and programs online, describe text processing, arithmetic manipulation, evaluation processing, etc.

Until such time as a Ted Text Editor Users' Guide is available, refer to the Qedx Text Editor Users' Guide, Order No. CG40, for qedx compatible request usage within the Ted subsystem. The most basic manual available (for qedx) that presupposes no user knowledge of either Multics or a text editor subsystem is the Guide to Multics WORDPRO for New Users, Order No. DJ18.

The only aspect of word processing on Multics covered in this document is the use of the ted command; other aspects are described in the WORDPRO Reference Manual, Order No. AZ98.

File No.: 1L13 CP50-00B

This manual contains references to the Multics Commands and Active Functions, Order No. AG92 referred to as the Commands and the Subroutines and Input/Output Modules, Order No. AG93 referred to as the Subroutines.

Often, user typed lines and lines displayed by Multics are shown together in the same examples. To differentiate between these lines, an exclamation mark (!) precedes user-typed text. This is done only to distinguish user text from system-generated text, it is not to be included as part of the input line. Also, a "carriage return" (moving the display mechanism to the first column of the next line, called a newline or NL on Multics) is implied at the end of every user-typed line.

Note: Because of page constraints in this document, certain character strings of data used in examples may not match exactly the information as seen on a user's terminal. That is, the character strings in examples may be folded or multiple lined, whereas the actual interactive (live) session may display the same information on a single line or multiple lines with different line breaks than shown.

SIGNIFICANT CHANGES IN THIS ADDENDUM

Ted now accepts multiple -request control arguments (see Appendix A). Other documentation changes are associated with trouble reports and customer orginated Tech Pub Remark Forms.

CONTENTS

SECTION 1

GENERAL INFORMATION

INTRODUCTION

Ted is an extended version of the Qedx text editor. Although the primary purpose of an editor is to provide for interactive creation and modification of textual material, Ted additionally provides for the execution of text editing requests which are stored in a Multics segment instead of being typed from a user terminal. What makes Ted special is that its request set provides for all of the functions generally available in a programming language such as branching, assignment, arithmetic manipulation, and subroutine invocation.

ABBREVIATIONS, METASYMBOLS, AND SPECIAL CHARACTERS

A number of abbreviations, metasymbols, and special characters are used throughout this document. For user convenience, all of these cases have been consolidated in Appendix C.

BASIC OPERATION

To create a new ASCII segment, a user might perform the following steps:

1. Invoke ted and enter input mode by typing one of the input requests (e.g., append) as the first ted request.
   a. Enter ASCII text lines into the buffer from the terminal.
   b. Leave input mode by typing the appropriate escape character sequence as the first characters of a new line.
2. Inspect the contents of the buffer and make any necessary corrections using edit or input requests.
3. Write the contents of the buffer into a segment using the write request.
4. Exit from the editor using the quit request.

To edit an existing ASCII segment, a user might perform the following steps:

1. Invoke ted and read the segment into the buffer by giving a read request as the first ted request.
2. Edit the contents of the buffer using edit and input requests as necessary. (The editor makes all changes on a copy of the segment, not on the original. Only when the user issues a write request does the editor overwrite the original segment with the edited version.)
3. Using the write request, write the contents of the modified buffer either back into the original segment or, perhaps, into a segment with a different name.
4. Exit from the editor using the quit request.

The user can create and edit any number of segments with a single invocation of the editor as long as the contents of the buffer are deleted before work is started on each new segment or a new buffer is selected in which to work.

Note: While most users interact with ted through a terminal, the editor is designed to accept input through the user_input I/O switch and transmit output through the user_output I/O switch. These switches can be controlled (using the iox_ subroutine described in the Subroutines or the io_call or file_output commands described in the Commands) to interface with other devices/files in addition to the user's terminal. For convenience, this description assumes that the user's input/output device is a terminal.

Responses from the Editor

In general, the editor does not respond with output on the terminal unless explicitly requested to do so (e.g., with a print or print line number request). The editor does not comment when the user enters or exits from the editor or changes to and from input and edit modes. The use of frequent print requests is recommended for new users of Ted.

ADDRESSING CONVENTIONS

Lines in the buffer can be addressed by three basic means (also see the description under "Buffers" below, specifically "Absolute Buffer Referencing," as regards addressing the current or auxiliary buffers):

- Absolute addressing
- Relative addressing (i.e., relative to the "current line")
- Contextual addressing

In addition, a line address can be formed using a combination of the above techniques.

An address consists of two major parts: the line part and the byte part (both parts are optional). The general format is:

<line-address> (<byte-address>)

The line address references a line within a buffer. The byte address references a byte within a line. (However, for an exception to this, refer to "Mode Change" described below.) A byte address is generally of the same form as a line address except that it is enclosed in parentheses.

When two addresses occur to form a range, they can be separated in one of two ways:

,

a comma as an address separator means that the following (or right half of the) address is to be searched for from the same starting point as the past (or left half of the) address.

;

a semicolon as an address separator means that the following (or right half of the) address is to be searched for from the location found by the past (or left half of the) address.

A special shorthand address form is available in certain circumstances. The following three forms are identical in meaning, with the last being the shorthand version.

```
/abc/(2);.(10)
/abc/(2),/abc/(10)
/abc/(2,10)
```

In other words, when the line part is the same on both addresses, then the byte portions may be combined within the same parentheses.

Absolute Addressing

Each line in the buffer can be addressed by a decimal integer which indicates the current position of the line within the buffer. The first line in the buffer is line 1, the second line is 2, etc. The last line in the buffer can be addressed either by line number or by using the "$" character, which is interpreted to mean "the last line currently in the buffer." In certain cases it is possible to address the (fictitious) line preceding line 1 in the buffer by addressing line 0. This technique is used to insert text before the first line in a buffer using the 0a (append) request.

As lines are added to or deleted from the buffer, the line numbers of all lines that follow the added or deleted lines are changed accordingly. For example, if line 15 is deleted from the buffer, line 16 becomes line 15, 17 becomes 16, and so on.

If an attempt is made to address a line not contained in the buffer, an error message is printed by the editor. No address is considered valid if the buffer is currently empty.

When the first portion of a given address is a number (e.g., 1, 50, 266) then the reference is said to be absolute. The address "0" is a special case of an absolute address. It refers to the beginning of the buffer. 0(n) refers to the n'th character of the buffer. Ted, depending on the use of line mode or string mode (described in Section 1), permits two versions of absolute addressing (either by line number or character number within a line). In the following examples, there are several cases where an address can be given in more than one format. The standard format is always shown on the first line, followed by the short or abbreviated format on the second line.

For example:

```
(1),(3)d   (standard format)
(1,3)d     (short format)
```

describe the same address.

```
3,5p
```

is an absolute line range in the buffer. (This example instructs ted to print lines 3 through 5 of the current buffer, whether in line or string mode.)

```
(1),(3)d
(1,3)d
```

is an absolute character number reference within the current line in the buffer. (This example instructs ted to delete the first, second, and third characters of the current line.) After execution, and if in string mode -- entered via "\s", the line pointer retains character orientation.

If the above request is followed with a p (print) request, the display produced is the next (or fourth) character only, not the entire line contents.

If the request had been executed in line mode, then the print request would have resulted in a display of the remainder of the line (i.e., characters remaining in the line after deletion of the first three characters).

```
0(5),0(45)p
```

prints the 5th through 45th character of the buffer without any consideration of how many lines it may be.

When the "$" character is included in an address it refers to the "end-of" the buffer or line. The "$" in a byte address explicitly references the NL (or, in the case of the last line of a file which does not end in NL, "$" references the (fictitious) byte which would contain the NL if it were present). For example:

5,$d
deletes lines of the buffer (from line 5 to the end).

(6,$) d
deletes a group of characters in the current line (deletes character positions 6 through the last character of the line including the NL character, if present). This has the effect of inserting characters 1 through 5 of the current line at the beginning of the next line since the last line delimiter (i.e., NL) was deleted.

And finally, when the "." character is included in an address, it refers to the current position, either line-in-buffer or byte-in-line. For example:

.p
prints current line of the buffer.

*
Relative Addressing

Ted maintains the notion of a "current line" that is specified by using the character "." (period). Normally, the current line is the last line addressed by an edit request or the last line entered from the terminal by an input request.

Addressing may be done relative to the current position (in a forward or reverse direction) by utilizing signed integers (either "+" or "-") in the address. The "+" character is optional when not the first character of the address. Thus, ".+4" is the same as ".4", which is the same as "+4". Some examples are:

$-5P
displays the fifth line preceding the last line ($) of the buffer. (This request also displays the line number of the selected line.)

($-5) , ($) d
deletes the last five characters of the current line and the NL character (if one exists). An error message is provided in the event that the line contains less than five characters.

15 ($) d
deletes the NL of line 15. If the last character is a NL character, then the NL is deleted. (This is one case where referencing something which does not exist does not result in a ted response.) Additionally, if a line 16 exists, then line 16 is merged with line 15.

.,.+5p
.,.5p
displays six lines of the buffer, the current line and the following five lines.

(20) ; (.+6) p
displays seven characters of the current line starting with the 20th character and continuing with the following six characters.

Contextual Addressing

Addressing may be accomplished using an <RE> (regular expression) search within a buffer. An <RE> search is accomplished by starting on the line following the current line and searching through to the end of the buffer if necessary. If the <RE> is not found, the search continues using the wraparound feature, to the beginning of the buffer and searches forward from this point up to and including the current line (or original starting point). The "/" character is the required delimiter in this use of an <RE> (e.g., /abc/-4).

An <RE> search within a line is accomplished starting with character position (.+1) except when included after a line address. (Conceptually the line address locates the 0'th byte of the selected line so the <RE> search within the line begins at character position (0+1)). The "/" character is also the required delimiter in this use of an <RE> (e.g., "(13/somestring/)").

A backup search capability also exists by utilizing "</<RE>/". This search looks for an expression by moving backward one line at a time (the current line is not searched -- search starts at location ".-1"). If after examining the buffer contents no match is located, then the backup search fails (i.e., unlike the forward search that wraps around from the end-of-buffer to the beginning-of-buffer, the backup search does not wraparound). The "/" character need not be used as a delimiter in this case (e.g., $<"end;" is a valid <RE> that uses quotes as a delimiter). Some examples of the backup search follow.

Example 1

Buffer contents:

```
        one
        two
"."-> three
        four
        five
```

Request:

```
<!four! =
```

Terminal output:

```
backup search failed.  "!four!"
```

Example 2

Buffer contents:

(Same as Example 1, "." is also the same)

Request:

```
</one/ =
```

Terminal output:

```
1
```

meaning that the <RE> was found in line 1 of the buffer (the = request says to print the linenumber).

Example 3

Assume the current buffer contains a text segment and you are interested in determining in reverse order how many lines within the file contain the string "then", further assume that each find is to be displayed with linenumber.

Request:

```
$n                      (set pointer to last-line of buffer)
*/then/ P               (test the last line, since the next request will not)
</then/ P >-0
```

results in a backup search through the buffer, displaying each line containing "then", with associated line number, and then repeats the request line with the ">-0" request (see the > request for additional information).

The above searching procedure may not always be what is needed. The range to be searched may be limited by placing an address enclosed in brackets ([]) immediately before the search specifier. For example:

```
[5,12]/abc/     (searches forward for "abc" from lines 5 through 12 only)
[5,12]</abc/    (searches in reverse for "abc" beginning on line 12, then 11, on down to 5)
```

The address within the brackets can use any means of specification (other than buffer selection).

An <RE> is a description of a string. Most characters in an <RE> are taken literally. The following have a special meaning:

^

as the first character of an <RE>, it matches the (fictitious) character preceding the first character on a line.

$

as the last character of an <RE>, it matches the NL at the end of the line. However, the NL is not considered part of the matching string. The matching string is referenced by & in the <REPL> part of an s (substitute) request. (Also see "String Mode" described in Section 1.) This fails on the last line of a buffer if it does not end with a NL.

*

matches ZERO or more occurrences of the character which immediately precedes the "*" in the <RE>.

.

matches any character.

/

normally used to delimit an <RE> used as an address. If any of the special characters (^$*.) are used as a delimiter then their corresponding special function cannot be used in the <RE>.

\[N]x

matches N occurrences of character x.

\[N].

matches N occurrences of any character.

\[N]\c.

matches N occurrences of "." (period).

The following list of examples is a small sampling of addressing by context. The examples include an <RE> with a description of the editor action.

/a/

matches the letter "a" anywhere on a line.

/abc/

matches the string "abc" anywhere on a line.

/ab*c/

matches "ac", "abc", "abbc", "abbbc", etc. anywhere on a line.

`/in..to/`
matches "in" followed by any two characters followed by "to" anywhere on a line.

`/in.*to/`
matches "in" followed by any number of any characters (including none) followed by "to" anywhere on a line.

`/^abc/`
matches a line beginning with "abc".

`/abc$/`
matches a line ending with "abc".

`/^abc.*def$/`
matches a line beginning with "abc" and ending with "def".

`/^.*$/`
matches any line.

`/^$/`
matches an empty line (a line containing only a NL character).

`/abc/-5`
references the fifth line preceding the line containing string "abc" (the <RE> in this case).

`(/abc/+2)`
references the "c" of the first occurrence of "abc" on the current line.

`0/J/(/k/)`
references the first "k" on the first line containing a "J" (starting the search on line 1 of the buffer).

`(/a.*z/)d`
deletes the string beginning with "a" and ending with "z" on the current line.

`\s (/abc/);(+5)d \l`
deletes a string beginning with "abc" and the 3 characters following "abc" (i.e., 6 characters deleted in all). This string may be anywhere in the buffer after the current line since string mode is specified.

Using "?" Prefix

A "?" character preceding an address is used for testing the validity of the address. This is known as an address prefix. The address prefix has the same form as an address and results in the same address processing, but does not act as an address to the request. An address prefix can only be present before the first address.

Syntax

Format 1	Format 2	Format 3
`?ADR, ...`	`?ADR; ...`	`?ADR <request>`

Format 1 tests for validity and then has no further effect. Format 2 tests for validity and then leaves the effective "." at the location found by the address. Format 3 appears to <request> as though no address had been present.

If any part of an address cannot be found, all processing of the request line stops. (No notification or message of this occurrence is provided to the user; ted merely ignores the rest of the request line and continues.) However, since all address processing is completed, the user may want to be at that location. This is accomplished by separating the prefix from the following address with a ";". For example,

```
?/Charly/;.;.+1d >-0
```

finds each line in the buffer containing "Charly" (starting at "."), deletes each line containing the <RE> which in this case is "Charly" and the following line. Actually this is not fail-safe. To cover the possibility of the match beginning on the last line of the buffer (important when writing a ted_com where control must be retained), the user should enter:

```
?/Charly/+1;.-1;.+1d >-0
```

In the above two examples, ">-0" is a transfer of control request (re-execute current line).

An example utilizing Format 3 is:

```
?1 t/Data left over/ 1
```

Compound Addresses

An address can be formed using a combination of the techniques described above. The following rules are a general guide in the formation of these compound addresses.

1. If an absolute part is to appear in an address, it must be the first component of the address. (A "." or "$" cannot appear anywhere except first.)
2. A relative line number can appear anywhere in a compound address.
3. An <RE> can appear anywhere in a compound address.
   a. An absolute line number can be followed by an <RE>. This construct is used to begin the <RE> search after a specific line number. For example, the address "10/abc/" starts the search for "/abc/" immediately after line 10.
   b. An <RE> can follow, or be followed, by an address specified by a relative line number. For example, the address "-8/abc/" starts the search eight lines before the current line, while "/abc/+8" addresses the line eight lines after the first occurrence of "abc".
   c. An <RE> can be followed by another <RE>. For example, the address "/abc//def/" matches the first line containing "def" appearing after the first line containing "abc". As mentioned earlier, an <RE> can be followed by a decimal integer. For example, the address "/abc/-10/def/+5" starts the search for "/def/" 10 lines before the first line to match "/abc/" and if "/def/" is matched, the value of the compound address is the fifth line following the line containing the match for "/def/".

      Note: The second search could wraparound and find a match which is actually before the first search match.

Addressing Errors

Refer to "Error Messages" in Appendix D for a complete list of all ted error messages.

Mode Change

Characteristics of addressing functions may give different results (i.e., other than expected) depending on the use of line or string mode. Changing between line mode and string mode (invoked by \l and \s) may be accomplished at any time within an invocation of the ted command except when in input mode.

CONCEALING SPECIAL CHARACTERS

The special meanings of "/", "*", "$", "^", and "." within an <RE> can be removed by preceding the special character with the escape sequence \c (i.e., the special character is taken literally).

Concealing may be done in any <RE> in the replace portion of an s (substitute) request, and in input mode. At any time you may conceal \b, \{, \r, or <NL>. In order to input the sequence \f into text while in input mode, the user would have to enter:

```
\c\f
```

otherwise the \f by itself would terminate the input mode and switch ted to edit mode.

```
/\c/\c*/
```

matches the string "/*" anywhere on a line.

Within an invocation of ted, the last <RE> used in any context, with the exception of g*, is remembered. The user can reinvoke the last used <RE> by using a null <RE> (i.e., //). In addition, an <RE> can be followed by a signed decimal integer in the same manner as when addressing relative to the current line number. For example, the addresses "/abc/+5-3", "/abc/+2" or "/abc/2" all address the second line following a line containing "abc".

The two uses of "." and "$" as line numbers and as special characters in an <RE> are distinguished by context.

EDITOR REQUEST FORMAT

Ted requests normally take one of the following three general formats:

```
<request>

ADR<request>

ADR1,ADR2<request> -or- ADR1;ADR2<request>
```

where ADR is a one-line address, ADR1 and ADR2 are the first and second components of an address range (address prefixes may precede any of these ADR forms), and <request> is a ted request. When addressing a series of lines for a specific request (ADR1,ADR2<request>), any of the three address types (absolute, contextual, or relative) can be used for either ADR1 or ADR2.

For example, if line 1 is the current line and the buffer contains:

```
b:procedure;
a=r;
c=s;
k=t;
end b;
```

you could display lines 2 through 4 by executing a p (print) request preceded by one of several address combinations; a few of the possible address combinations are:

```
2,4p
2,+3p
2,/^k/p
+1,/^k/p
/a=/,+3p
/a=/,/^k/p
```

Note: If you attempt to try more than one of the above combinations (for test purposes), be sure to relocate the pointer to line 1 by doing an n (nothing) request. (This is accomplished by executing a "1n" request prior to each new request.)

Some editor requests require no address, some require a single address, and others require a pair of addresses. In most cases, however, the user can issue a request omitting one or both of the required addresses and let the editor provide the missing address information by default. The following general rules apply to the use of addresses.

1. If a request requiring an address pair is issued with the second address missing, the (missing) second address is assumed to be the same as the first. For example,

   ADR<request>

   is interpreted as:

   ADR,ADR<request> or equivalently, ADR;.<request>

   and addresses a single line in the buffer (i.e., the line addressed by ADR).

2. If a request requiring an address pair is issued with both addresses missing, one of the following address pairs is assumed depending on the request issued.

   .,.<request> for all editor requests except write, global, exclude, zif, and evaluate

   1,$<request> for write, global, and exclude

   <request> for zif and evaulate (there is no default)

3. If a request requiring a single address is issued with no address specified, one of the following addresses is assumed depending on the request issued.

   .<request> for all editor requests except read

   $<request> for read requests

4. If a request is given with more addresses than the request will accept, an error message is displayed.

VALUE OF "."

1. There is a current line pointer for each buffer.
2. Its name is "."
3. All editor requests that alter the contents of the buffer, or print from the buffer, change the value of "." (i.e., the current line). Usually, the value of "." is set to the last address specified (either explicitly or by default) in the editor request. The one major exception to this rule is the delete request, which sets "." to the line AFTER the last line deleted. (If the line deleted was the last line in the buffer, then "." is set to "$+1" (beyond end of the buffer). Negative relative addressing can be used at this point, but any other references to "." result in an "undefined" message.)

MULTIPLE REQUESTS ON A LINE

In general, any number of editor requests can be issued in a single input line. However, the following requests must be terminated with an NL character, (i.e., each must appear on a line by itself or be at the end of a line containing multiple editor requests.

r	read
w	write
q	quit with buffer check
e	execute command line

Whereas the following must appear at the beginning of a line and also must appear on a line by itself.

..	execute command line without input function processing
\?	Where am I?
help	online information

SPACING

The following rules govern the use of spaces in editor requests.

1. Spaces are taken as literal text when appearing inside of regular expressions. Thus, /the<SP>n/ is not the same as /then/.
2. Spaces cannot appear in numbers (e.g., if 13 is written as 1<SP>3, it is an error), pathnames (e.g., if foobar or foo_bar is written as "foo<SP>bar", it is an error), or between a function name and the "(" (e.g., if fka(..." is written as "fka<SP>(...", it is an error).
3. Spaces within addresses except as indicated above are ignored.
4. The treatment of spaces in the body of an editor request depends on the nature of the request. For example:

   ```
   t!ab<SP><SP>de!
   ```

   results in two literal spaces between "ab" and "de" in the typed line. Also refer to "Input Mode" below, concerning spaces following a request.

MODES OF OPERATION

Ted functions in one of five basic modes: input, bulk, read, break or edit. Each mode is discussed individually and includes examples.

Input Mode

Ted can be placed in input mode with one of the three input requests: a, c, or i (append, change, and insert). If the input request is followed immediately by a SP or NL, that character (SP or NL) is discarded. If in blank mode, anything following the request other than SP or NL is an error (refer to Appendix B, "o" request, for a description of the blank mode). If not in blank mode, then the character immediately following the request (other than SP or NL) is part of the input data or literal text. If the user wants the first character of the text (or line) to be a space, then two spaces must be input (e.g., "a<SP><SP>text..."). The first space is discarded as noted above, leaving the second space intact. The literal text can contain any number of text lines. To exit from input mode and terminate the input request, the escape sequence \f (or \F) is typed, usually as the first characters of a new line. The \f escape sequence can be followed immediately with more editor requests on the same line. The usual form of an input request is as follows:

```
ADR <input request>
TEXT
  .
  .
  .
\f
```

except for the c request which may take an address pair to replace several lines by the new TEXT.

It is important to remember to terminate the input request with the \f escape sequence before typing another request. Otherwise, the editor remains in input mode, and the (would be) editor request is regarded as input and included in the text rather than executed as a request.

Upon leaving input mode, the value of "." is set to point to the last line input from the terminal.

The \{}, \b, and \r are input functions. In general, ted input is obtained a line at a time. Input comes from user_input and from invoked buffers. Each time ted needs another line it calls the input routine. This routine takes from the input until an NL is encountered. Along the way it looks for input functions and processes them.

All input functions are processed while in input mode, however, they may be entered as literal data by means of the conceal escape sequence (\c). (All of these sequences are described later in this section under "Input Functions.")

Ted knows nothing about any input functions which may have existed. The input routine knows nothing about what will be done with the line retrieved. The result of this is that all input functions on a line are processed before any requests in the line. You might consider the input functions as having higher priority than requests. This means that with a line typed as:

```
2,4k(x) $a \bx\f
```

you might think that you are moving data to buffer x and then appending it to the end of the current buffer. However, since the \b has higher priority, it is done first and the data is not in buffer x yet. If you are lucky and b(x) has not been used, then you will get an error message

indicating that the buffer does not exist. Otherwise it uses the contents of b(x) and continues. In this case it could be some time before you find that it did not do as expected. (Also refer to "Input Functions" described below.)

Bulk Mode

Ted can be placed in bulk input mode with the use of one of three bulk input requests: !a, !c, or !i (bulk append, bulk change, and bulk insert). These requests accept lines of data without processing of any kind. The only thing being looked for is the terminate sequence (a line consisting only of a "."). In this mode, what you see is what you get (i.e., \b, \r, \{}, \c, and \f are just treated as data). This is very handy for entering ted_coms because you can forget about how much concealing an input function needs; you simply type.

Bulk requests differ from input requests in that all bulk data is literal whereas input data is examined for possible input functions.

Requests may follow on the same line as a bulk request. These are not input data; they are requests executed after completion of the bulk input request. For example,

```
!   10!a 11p                    -bulk append after line 10 in current
                                 buffer.
!   data1
!   data2
!   data3
!   .                           -terminate bulk append and switch to edit
                                 mode.
    data1                       -displayed by the "11p" request above.
```

Read Mode

Ted can be placed in read mode with the \r input function. This gets a line from the terminal. When this function is encountered, a line is read from user_input and used as if it replaced the \r. The NL character on the line read is not included with the replacement data and the line is not scanned for input functions. Several examples follow.

```
!   ted                                  -invoke ted command
!   a Now is the time                    -append data
!   TEST
!   to the aid of their country.
!   \f                                   -terminate input mode.
!   2s/TEST/\r/ P                        -substitute, replacing with user_input
!   for all good men to come             -user_input replaces "TEST"
        2    for all good men to come
                                         -print with line number--second request
                                          on substitute request line above.
!   1,$P
        1    Now is the time
        2    for all good men to come
        3    to the aid of their country.
```

The request line:

```
1,$s/abc/\r/ s/def/\r/
```

requires two user responses before the line is executed. Ted then replaces all occurrences of "abc" in the buffer with whatever is entered from the terminal (user_input) first, and then replaces all occurrences of "def" (last line only as the second request does not contain an address) with whatever is entered second.

```
!   ted                                  -invoke ted
!   a                                    -append
!   abc Now is the time
!   def for all good men
!   abc to come to the aid
!   def of their country.abcdef.
!   \f
!   1,$s/abc/\r/ s/def/\r/
!   TEST1                                -user_input for first substitute
!   TEST2                                -user_input for second substitute
!   1,$P                                 -print buffer contents
        1    TEST1 Now is the time
        2    def for all good men
        3    TEST1 to come to the aid
        4    TEST2 of their country.TEST1TEST2.
```

The substitute for "def" did not touch line 2 because the second substitute request (s/def/...) did not contain an address pair. The "." pointer was set to $ (last line of buffer) after execution of the first substitute therefore only the last line was acted upon. (Also refer to "Input Functions" described below.)

Note: If a \f is given in response to a \r it causes the request line to be aborted. (If in input mode, it is properly terminated.) The action is then like "search-fail", see "Ted Execution" below. The request line:

```
/\r/+1 s/xxx/\r/ >-0
```

loops indefinitely, getting two responses each time the request line is executed. If either response is \f the processing immediately stops. Also refer to \r (input function) below for additional detail.

Break Mode

This mode is entered by encountering a breakpoint. It has all the functionality of edit mode except that input mode, if entered, must be exited on the same line. Refer to "Using Breakpoints" in Section 3.

Edit Mode

All requests other than a, c, i, !a, !c, and !i function in the edit mode. The functions described as "line mode" and "string mode" (below) provide a powerful tool to manipulate data.

LINE MODE

Line mode treats a file as a series of lines (separated by the NL character). The current location in this mode is a line in the file. Line mode is entered during address processing by \l. (Default)

STRING MODE

String mode treats a file as a character string. The current location in this mode is a character in the file. String mode is entered during address processing by \s. String mode interacts slightly with line and byte addressing and with string search. The following table identifies the differences between line and string modes.

	Line Mode	String Mode
\s	Goes to string mode.	Stays in string mode.
\l	Stays in line mode.	Goes to line mode.

Note: The \l and \s are considered address parts. The mode can change any number of times during an address.

/a.*b/	The file is searched for "a" starting on the line following the current line, proceeding to the end of the buffer, then from the beginning of the buffer to the current line. If "a" is found, then "b" must follow	Same as line mode as regards the search for "a". If "a" is found, then "b" must follow "a" in the buffer (i.e., the "." can match an NL).

	"a" on the same line (i.e., "." cannot match an NL).	
(/a.*b/)	"a" is looked for in the current line, starting at the current byte (first byte of line, unless a previous byte address part has been given in the address).	"a" is looked for in the rest of the buffer, starting at the current byte.
12($+1)	Displays the error message: "Addr- after line end".	Addresses the first character of line 13.
12(-1)	Displays the error message "Addr- before line begin".	Addresses the NL character of line 11.
2(3)= ·	Displays the line-byte address: 2(3)	Displays both the file address and line-byte address: 0(9) 2(3)
2(3)= =	The byte portion of the current location IS NOT maintained from the first request to the second. 2(3) 2	The byte portion of the current location IS maintained from the first request to the second. 0(9) 2(3) 0(9) 2(3)
1,s/c/X/	Buffer before: 1 abc 2 crb Buffer after: 1 abX 2 crb	Buffer before: 1 abc 2 crb Buffer after: 1 abXcrb
	The NL is used in finding the string but IS NOT included in it.	The NL is used in finding the string and IS included in it. Note: This function may be accomplished in line mode by a conceal of the NL as: 1,$s/c\c\012/X/

Example Using String Mode

Buffer contents:

```
Now is the time for all good
men to ... This request may
also be given as xxx. come to
the aid of their country.
```

Assume that ted is in line mode and you want to do a search for the following <RE>:

```
/This request may also be given as xxx\c./
```

The search fails and ted responds with the message:

```
Line search failed. "/This request.../"
```

The only way to find a line of this format (when you do not know how much of the expression is on a common line) is to enter string mode (\s), and then make the search.

```
\s /This.request.may.also.be.given.as.xxx\c./
```

ted responds with:

```
men to ... This request may
also be given as xxx. come to
```

The dot "." is used between each word to match the possible NL character that could appear after any of the search words.

TED EXECUTION

When you inadvertently request a large amount of terminal output from the editor and wish to abort the output without abandoning all previous editing, simply issue the QUIT signal (by pressing the proper key on the terminal, e.g., BRK, ATTN, INTERRUPT), and, after the QUIT response, reenter the editor by invoking the program_interrupt (pi) command (described in the Commands manual). This action causes the editor to abandon its printout, but leaves the value of "." as if the printout had gone to completion. Execution continues with what follows the print request.

To ABORT anything, issue the QUIT signal and then type "ted -reset" (see Appendix A). This acts as if an error has just occurred and unwinds execution back to ted request level. All buffers are as they were at the time the interrupted request began (except for uppercase and lowercase requests (u and U) which work in place). That is to say, no change to buffer data is effective until a request successfully completes.

If a ted_com (refer to Section 2) is running and it appears to be in a loop, execution can be suspended to allow examination of conditions. Issue the QUIT signal and then type "ted -pause". The next time a line of input is requested in edit mode it acts as if a breakpoint had been encountered (refer to Section 3).

If an error is encountered by ted, an error message (refer to Appendix D) is printed on the user's terminal and any further editor requests on the same line (i.e., multiple requests on one line) are discarded.

It is important to note when using multiple requests on a single line that only the requests following an error condition are discarded; the requests preceding the request that fails are executed. For example,

```
!   ted                                  -invoke ted command
!   a                                    -append -- enter input mode
!   1                                    -data entered
!   2
!   3
!   PROGRESS
!   4
!   5
!   END
!   \f                                   -terminate input mode and enter edit mode
!   1,$P                                 -print request
        1    1
        2    2
        3    3
        4    PROGRESS
        5    4
        6    5
        7    END
!   4s/PROGRESS/TEST/ P 1,3d 7s/END/something/ P
        4    TEST
    Addr- after buffer
!   1,$P
        1    TEST
        2    4
        3    5
        4    END
```

The substitute request (4s/.../), the print (P), and the deletion of lines 1-3 (1-3d) above are accomplished. This results in a buffer of four lines. Thus the error condition is related to the substitution (7s/.../). This substitution and the following print request (P) are discarded.

If a user gets out of ted by issuing the QUIT signal, and subsequently invokes ted in the same process without first using the release or program_interrupt command (at Multics command level), ted assumes the user intends to work on another file or wants to produce a new file. The original, or first invocation of ted "quit from" is not lost to the process (unless the user logs out) and can be returned to at anytime the user wishes. For example,

```
!   ted                              -first invocation of ted
!   r test_1                         -read segment "test_1"
!   20                               -print line 20
    something                        -data in line 20
!   QUIT                             -invoke QUIT by pressing BRK, etc.
    r 10:23 0.339 22 level 2         -Multics ready message

!   ted                              -second invocation of ted
!   r test_2                         -read segment "test_2"
!   20                               -print line 20
    xyz123                           -data on line 20
       .                             -other editing requests
       .
       .
!   \?                               -"Where am I?"
    ted(2.5c) EDIT MODE[02]
!   w                                -write modified file back to original
                                      location. "test_2"
!   q                                -normal quit request -- user is done
    r 10:24 0.625 60 level 2

!   pi                               -program_interrupt -- Multics command
                                      returns to previous invocation of ted. (In
                                      this case, "start" could also have been
                                      used.)
!   \?                               -"Where am I?"
    ted(2.5c) EDIT MODE[01]
!   20
    something
!   s/something/anything/            -substitute request
       .
       .
       .
!   w                                -write modified file back to original
                                      location, "test_1"
!   q                                -normal quit request
    r 10:25 0.360 19                 -Multics ready message
```

Another possibility is to use the e (execute) request and call for another invocation (up to a maximum of 14) of ted. For example,

```
!   ted                              -first invocation of ted
!   r test_4
!   1,5P
        1    .fif
```

```
       2    .pdw 72
       3    .pdl 66
       4    .sr fw "015"
       5    .alc
!  1d
!  e ted                                  -second invocation of ted
!  r test_16
!  1,5P
       1    .pdw 79
       2    80-03-13     MULTICS MANUALS COMPLETED -- PAGE 1
       3    .spb
       4    .fif
       5    FW       ORDER NO.
!  e ted                                  -third invocation of ted
!  r test_2
!  1,5P
       1    command_line off
       2    ab
       3    &goto &2
       4    &label interactive
       5    long_date
!  1d
!  \?                                     -"Where am I?"
   ted(2.6) EDIT MODE[03]
!  q
   Modified buffers exist:
   -> (0)     >user_dir_dir>Multics>test_2
!  ted: Do you still wish to quit?  no
                                          -write data back to "test_2"
!  w
!  q                                      -normal quit request -- returns process to
                                           second invocation of ted
!  1,5P                                   -print request used to verify second
                                           invocation of ted
       1    .pdw 79
       2    80-03-13     MULTICS MANUALS COMPLETED -- PAGE 1
       3    .spb
       4    .fif
       5    FW       ORDER NO.
                                          -normal quit request -- returns process
!  q                                       back to first invocation of ted
!  1,5P                                   -print request used to verify first
                                           invocation of ted
       1    .pdw 72
       2    .pdl 66
       3    .sr fw "015"
       4    .alc
       5    .ur PROJECT STATUS REPORT -- FW%fw%
!  q
   Modified buffers exist:
   ->(0)     >user_dir_dir>Multics>test_4
```

```
!   ted: Do you still wish to quit? no
!   w                                   -write data back to "test_4"
!   q
    r 10:14 1.620 128
```

BUFFERS

A buffer is a named set-of-data consisting of text, its length, a remembered address, and assorted flags. Buffer requests allow the user to create buffers, move text from one buffer to another, and check on the status of all buffers currently in use. A buffer may be used to hold the text actually being edited, a ted_com to be executed, pieces of the text being edited which have been moved or copied, arguments from the Multics call of the ted command, or other miscellaneous values. The buffer currently being edited is called the current (or working) buffer. Actually, ted supports a virtually unlimited number of buffers. One buffer at a time is designated as the "current buffer"; any other buffers are referred to as "auxiliary buffers." Any named buffer may be made the current buffer by invoking the b( ) or !b( ) request.

Each buffer is given a symbolic name of 1 to 16 ASCII characters not including the parentheses (). In addition, the buffer name cannot contain the characters "," or "." When ted is invoked, a single buffer (buffer 0) is created and designated as the current buffer. Additional buffers can be created merely by changing to, moving data to, or copying data to a previously undefined buffer name.

Buffer names of more than one character must always be enclosed in parentheses; for example, the buffer name Fred is typed as (Fred). A buffer name consisting of a single character can be typed with or without the enclosing parentheses (e.g., "y" is taken to be "(y)"). In this document, (X) represents a general buffer name of 1 to 16 characters. A buffer name may also be represented as (X,ADR1,ADR2) to define an address range. Use of this form indicates that some part of a buffer is to be referenced. A buffer name in an M or K request may also be represented by (X,ADR) to define a location. If a range is given, the end of the range is the location used. In both cases of (X,ADR1,ADR2) and (X,ADR) the parentheses are required. Buffers cannot be referenced with addresses unless they already exist.

Buffers exist for only the current invocation of ted. That is, if the user creates several buffers, issues the q (quit) request and then invokes ted again, the buffers created earlier are gone (unrecoverable). This should not be confused with the situation where a user has multiple invocations of ted in process. All current invocations of ted (up to a maximum of 14) are recallable with buffers intact.

Use of Buffers

Perhaps the most common use of buffers is for moving text from one part of a segment to another. A typical pattern is to place the text to be moved into an auxiliary buffer (see below) with an m (move) request. For example,

```
1,5m(temp)
```

moves lines 1 through 5 of the current buffer into the auxiliary buffer (temp) replacing its former contents, if any, and deletes lines 1 through 5 from the current buffer. Once the lines are moved

to an auxiliary buffer, they can be used as literal text in conjunction with an input request. For example, to insert the lines in buffer (temp) immediately before the last line in the current buffer, one of the following two sequences might be used.

```
$i
\b(temp)\f
```

In this case, the literal text in buffer (temp) replaces the \b input function in the input line. Notice that the \f immediately follows the \b. If the \f is put on the line following the \b, a blank line follows the literal text of buffer (temp) that was just inserted in the current buffer. (The blank line is caused by the two successive NL characters: one is the last character in buffer (temp) and the other is inserted between the \b and the \f.)

```
$-1r(temp)
```

also puts the data in the same place. However, there is a difference between these two methods. The \b method scans the data for input functions; the r() method does not. If there are no input functions in the data, it is cheaper to use the r(). If there are functions there but you do not want them to be expanded, you must use r() to avoid it.

Ted requests may be placed in a buffer and then called. Assume the current buffer is b(a) and a call is made to b(name) by typing the call request %(name). The current buffer remains b(a) and b(name) simply is the execution buffer. The ~ request stops execution in the execution buffer (assumed in this case to be b(name)) and reverts to the previous level which is request level.

Each buffer is implemented as a separate segment in the user's process directory (however, see -safe control argument in Appendix A) and, thus, is capable of holding any segment. Any buffer which has an associated pathname is assumed to contain meaningful data, otherwise it is considered as temporary data which just disappears when ted is exited. The buffer "b(0)" is considered meaningful even if there is no pathname. Whenever such a buffer is modified, it is flagged. When the buffer is written to permanent storage, or if all the data is deleted, the flag is turned off. This is a flag which is tested by the q request.

Whenever data is moved to a buffer, it no longer exists in its old location. The new buffer is the only place where the data exists. It is therefore flagged as "not-pasted" (from cut/paste terminology). This flag is turned off by either doing a read, "r(x)", by invoking the whole buffer, "\b(x)", or by deleting the entire buffer. This is a flag which is tested by the q request.

The active function ted_buffer is used to reference data in ted buffers. It returns the pathname of the segment containing the data in b(x). The bitcount of this segment gets set to properly reflect the amount of data present. After returning from the "e" request, the bitcount of the segment is used to determine the new length of the buffer data. For example:

```
e ds [ted_buffer 0].
```

dumps the segment which contains the data of b(0).

Absolute Buffer Referencing

The "current buffer," described above, is the buffer normally used when executing ted requests such as p or 1d. However, any buffer may be referenced at any time in an "absolute" sense (i.e., regardless of which buffer is "current"). The presence of the absolute buffer name is indicated by the @ character followed by a buffer name and a comma separator if address parts are included. The normal use of the @ character in both Multics and ted is to delete the entire

contents of the current line being built. Therefore, in order to use this feature, and to prevent the default erase character from accomplishing its normal "kill" function, it must be preceded with a backslash (\) character at the time of entry (typing) when it is intended as reference to an absolute buffer. For example:

```
!   ted
!   p
    Buffer empty.
!   \@3p
    Buffer empty. (in b(3))
!   \@3a forst line                    intentional typo
!   \f
!   x                                  verifies current buffer is "0"
          0     (0)
          1     (3)
!   \@3,(2)c i\f                       corrects typo noted above
!   \@(3),1                            ()s included but not required in this case.
    first line
```

If an error occurs while working in a buffer which is not the current buffer, the error message contains a trailer to identify the buffer where the error occurred (see example above).

Repeated Editor Sequences

Another common use for buffers is for the definition of frequently used editing sequences. For example, the user is faced with the task of adding the same source code sequence in several places in a program, but instead elects to type the editing sequence into a buffer only once and then invoke the contents of the buffer as many times as necessary. In the following example, buffer NEW contains the necessary editor requests and literal text to append four lines of text at any point in the current buffer. Assume the contents of buffer NEW is:

```
a
if   code ^=0 then do;
     call error (code);
     return;
     end;
\f
.-4,.lp
w
```

and the user invoked request is:

```
ADR\b(NEW)
```

This results in ADR becoming the address of the append request in buffer NEW and specifies the point at which the literal text is to be appended to the current buffer. The four lines of text in buffer NEW (lines 2-5) are appended to the current buffer; the \f terminates the append request;

and the print request prints the line preceding the appended lines, the four appended lines, and the line following the appended lines in the current buffer.

INPUT FUNCTIONS

Ted input can be viewed as a stream of ASCII characters. Depending on the context, some of these characters are interpreted as editor requests and others are interpreted as literal text. The following are recognized by ted, in either context, as directives to alter the input character stream in some fashion.

Note: The special meanings of \b, \r, and \{} can be suppressed by preceding the input function with a \c escape sequence.

\b(X)
\b(X,ADR1,ADR2)

Sequence used to redirect the editor input stream to read subsequent input from buffer (X). That is, replace the invoke sequence \b(X) with the contents of the specified buffer or with the selected portion of the contents of the invoked buffer. This is similar to the % request except no arguments can be passed. Also conditional execution is completely different.

Conceptually, the content of the buffers replace the invoke sequence. In actuality, the contents is used one line at a time and is scanned, looking for input functions that may result in nesting. In input mode the lines are used in sequence (assuming no \f therein). The \f sequence results in a return to EDIT mode. In edit mode the lines are used in sequence unless a goto (>) request is encountered. If another \b is encountered while accepting input from buffer (X), the stream is again redirected and the newly encountered input sequence is also replaced by the contents of the named buffer. The editor allows the recursive replacement of \b functions by the contents of named buffers to a recursion depth of 500.

The \b(X,ADR1,ADR2) form can be used to select a portion (or window) of buffer (X).

The buffer to which the input stream is redirected can contain editor requests, literal text or both. If the editor is executing a request obtained from a buffer (rather than from the terminal) and the request specifies a regular expression search for which no match is found, the usual error comment is suppressed and the remaining contents of the buffer are skipped. If one thinks of the input function \b(X) as a subroutine call statement, the failure to match a regular expression specified by some request in buffer (X) can be thought of as a return "statement".

Reading the contents of buffer (X) does not delete or alter in any way the contents of the buffer. The content of buffer (X) remains constant. Thus, ted can read input from buffer (X) many times (see "Repeated Editor Sequences" above).

Syntax

Format 1: `\b(X)`

Format 2: `\b(X,ADR1,ADR2)`

Format 1 invokes all of buffer X. "All" here is true only if no imbedded return requests are encountered. Also, use of the goto request can make portions be reused. Format 2 invokes part (window) of buffer X. However, if the portion invoked

executes a goto (>) request, then the window is "broken" and any portion may be accessed.

This function may also be entered as "\B...".

Example 1

An invoke sequence may look like:

```
\bx \b(XXX) \b(XXX,5;/abc/)
```

but the sequence cannot contain any input functions such as "\b(\ba)".

Example 2

Request:

```
\bx                    -or-          \b(x)
```

invokes buffer x. Whereas:

```
\b(x,1)
```

invokes just the first line of buffer x, and:

```
\b(x,0(1),0(20))
```

invokes the first 20 characters of buffer x.

Notes

Suppose a ted_com is executing and an error occurs on line 147 because one delimiter was missing on a substitute. You can go to the buffer containing the ted_com, fix line 147, return to the buffer being operated upon, and continue execution (in most every case). You would continue in this fashion:

```
\b(exec,147,$)
```

The ",$" is very important, otherwise only line 147 would be executed.

\r

Sequence used to temporarily redirect the input stream to read a single line from the user's terminal. It is normally used when executing editor requests contained in a buffer. That is, replace the read sequence (\r) with a line read from user_input. The NL character from the line read is not included (unless running in ted$qedx mode -- see Appendix A). The line is not scanned for input functions. If the \f sequence is entered from user_input, the line being built is aborted and an error message is displayed if at console level. In a ted_com, there is no error message; this condition causes the executing buffer to be exited and the invoking buffer to continue. In the string that replaces the \r sequence, additional \r or \b sequences have no effect. (Also refer to "Read Mode" above.)

Syntax

```
\r
```

This sequence may also be entered as "\R"

Example

Request:

```
s/abc/\r/ s/def/\r/
```

requires two responses from user_input before the line is executed. First, replace all occurrences of "abc" (current line only as no address is given) with the response entered for the first \r request, and second, replace all occurrences of "def" with the response entered for the second \r request. If either response contains "\f" then the line is not executed.

\{}

Sequence used to temporarily redirect the input stream to read any value "returned" as a result of an evaluation. That is, replace the evaluation sequence (\{}) with the value returned from the evaluation. If the evaluation returns nothing, then a null value is assumed. This input function cannot contain any input functions (i.e., no \b or \r).

Syntax

```
\{<eval>}
```

Note

Refer to Section 4 for more details regarding <eval>.

Example

The multiline request sequence:

```
{ln:=0}
a \{ln:-ln+10; LN:fak (ln,"00000"); LN: LN} \r\012\f >-1
```

accepts lines of input, automatically prefixing each one with a 5-digit line number which is incremented by 10. The loop is terminated by a user response of "\f". (The \r terminates execution of the buffer when the \f is input.) (Also refer to Section 4, "Arithmetic Factor" for an alternate form of the above example.)

Commentary to the above example is:

line 1
Initialize the counter.

line 2
Has two input functions (\{} and \r). The first functions as follows:

`ln:=ln+10`
increments the counter.

`LN:=fak(ln,"00000")`
converts the number to a character string with leading zeros and assigns it to LN.

`LN:`
displays the converted number without a NL.

`LN`
returns this same number for use in the request line.

The second function is the \r request which accepts the line the user types.

line 3
Terminates input mode and loops back to get the next line.

CALLING TED AS AN ACTIVE FUNCTION

Ted can be called as an active function. This means that if a function does not exist in the system to do what is desired, then the user can get ted to do it. For example, you want to use the information "how many lines are in a segment?". The status active function provides bit count, or maximum length, or current length, or records used, but no line count. Following is a function which does just that. Line numbers are shown on the example for purposes of commentary immediately following the example.

```
1   !   ted
2   !   !a
3   !   zif {ag^=1} !t|Usage: [ted lines pathname]| !1 Q
4   !   r \b(arg1)
5   !   {lines:=fln(be)}
6   !   b(argn) a \{lines}\f Q
7   !   .
8   !   w lines.ted
9   !   q
10      r 15:00 143.804 9013
11
12  !   ioa_ "File lines.ted contains ^a lines." [ted lines lines.ted]
13      File lines.ted contains 4 lines.
14      r 15:01 0.901 152
```

line 1
Invoke ted

line 2-9
Build an active function ted_com "lines.ted"

line 3
The first line of the ted_com, complains if it is not called with exactly 1 argument.

line 4
The second line of the ted_com, reads the specified file.

line 5
The third line of the ted_com, sets the variable "lines" to the value of the function linenumber (end-of-buffer).

line 6
The fourth line of the ted_com, places the value of the variable "lines" into b(argn) and quits (Q).

line 12
Actually uses the active-function and prints the result which is displayed on line 13.

line 13
The display requested. The ted_com is contained in four lines of the buffer (i.e., the data which was entered on lines 3-6 of the example).

When ted is called as an active function the following control arguments are not allowed:

-Jshow
-Jset
-pause
-reset
-restart
-status

TED REQUEST GROUPING

The following list is intended to acquaint the user with associated groupings of requests (all of which are described in detail in Appendix B). Many of the requests can be invoked in more than one way (e.g., gP or g!p, !p or P).

Basic Requests

\?	Where am I?
d	delete
g=	global linenumber
gd	global delete
gp	global print
g!p	global print with linenumber
help	online information
o	options
!p	print with linenumber
p	print
!q	quit-force
q	quit with buffer check
r	read
v	exclusive global--inverse action to the g requests
w	write
=	print linenumber

Input Requests

!a	bulk append
a	append
!c	bulk change
c	change (replace) text
!i	bulk insert
i	insert

Intermediate Requests

b	change buffer
!e	execute a command line after printing it
e	execute command line
h	process out pseudo-tabs

j	sort
!j	special sort
k	kopy (or copy)
m	move
n	nothing, sets "." to address
!r	read with abbrev expansion of pathname
!s	no-fail substitute
s	substitute
u	translate to lowercase
!u	translate to uppercase
!w	write with abbrev expansion of pathname
wm	write all modified buffers to their respective segments (blank mode only)
!x	status of named or current buffer
x	status of all buffers
xm	status, modified buffers only (blank mode only)
y	tabin, process in horizontal tab characters
zdump	dump octal/ascii
z.fi.ad	fill/adjust
z.fi.na	fill/no-adjust
..	execute command line without input function processing, beginning-of-line only
|xxx	external call to ted_xxx_

Advanced Requests

!b	remember current buffer, then change
b()	change current buffer, going to a remembered one
^b	delete buffer
!f	fileout to a buffer, no auto reversion
f	fileout into a buffer, auto reversion
g*	global-if
!k	kopy-append
!l	linefeed to error_output
l	linefeed to user_output
!m	move-append
qhold	exit ted while maintaining the -safe environment
^r	force pathname
!t	type string to error_output
t	type string to user_output
"	comment, ignore rest of line
{...}	evaluate

Flow-Of-Control Requests

#	if line
^#	if-not line
%	call buffer with optional arguments
*	if expression found
^*	if expression not-found
:	label define, must be at beginning-of-line
>	transfer of control (goto)
^>	error exit

```
zif {...}    conditional test
~            return from buffer
```

USING THE HELP FACILITIES

Three options are available to the user:

1. Multics System "help" Facility
2. "Where am I" Ted Facility
3. Ted "help" Facility

Multics System "help" Facility

This facility displays information in the form of online documentation (called info segments) which include descriptions of system commands (such as the ted command), active functions, subroutines, miscellaneous information about system status, system changes, and general information. For example, if you are in Ted and wish to ask a question about a system command, simply invoke the e request as:

```
e help <REST>
```

to display the online documentation describing the system command <REST>. For a detailed description of the Multics help command, refer to the Commands manual.

"Where am I" Ted Facility

This facility is invoked within ted by typing "\?" (see Appendix B) on a line by itself. It is a means of reorienting the user to the current invocation of ted. The feature can be used by persons working on more than one invocation of ted, who may be interrupted, and simply want to know "Where am I?" in order to get back to the point where left off.

It can also be used when an expected response is not forthcoming (i.e.. it could tell you that you are in input mode. Then you would know why there has been no response to entered requests).

Ted "help" Facility

This facility is called within ted. It is invoked by typing "help" and is a means of accessing information segments about many subjects concerning the ted command. This facility should not be confused with the Multics system "help" facility described above.

SECTION 2

TED_COMS

A ted_com is an editor request sequence (commonly referred to as "macro" in Qedx). They may be placed in auxiliary buffers and used by ted as an interpretive programming language. In this context, it is useful to regard the executable editor requests in a buffer as a subroutine and to view the \b (input function) or the % (call buffer) request as a call statement when using auxiliary buffers. (Also refer to "Generating a ted_com" in Section 3.)

In the example discussed below, a ted_com is implemented to read ASCII text from the terminal until an input terminating sequence, a line consisting only of ".", is typed. When the terminating sequence is typed, the ted_com asks the user for a name under which the input is filed and exits from ted. The ted_com is implemented with an executable buffer (subroutine) named read and is invoked by diverting the input stream to the read buffer (i.e., by calling the read subroutine).

The content of buffer read (excluding line numbers) is:

```
1     t|Type| 1
2     :a $a
3     \r
4     \f
5     ^*/^\c.$/ >a
6     d
7     t|Give me a segment name: |
8     w \r
9     q
```

line 1
The first request (t) prints the message "Type" on the user's terminal. The second request (1) adds an NL.

line 2
The colon (:) defines a label referred to in line 5 request ">a". The second request ($a) places ted in input mode to append text to the end of the current buffer.

line 3-4
One line is read from the user's terminal with the \r input function on line 3 and the append request is terminated with the \f on line 4.

line 5
If the current line does not match the expression "^\c.$" (which checks for a "." on a line by itself), then go back to label "a".

line 6
The terminating sequence is deleted with the d request if the test from line 5 is satisfied.

line 7
The ted_com asks the user for a segment name in which the input lines

line 8
The content of the current buffer containing the input lines is written into a segment with the w (write) request, the name of which is read from the terminal by the \r (input function).

line 9
The ted_com exits from ted with the q (quit) request. If the quit request were not included, ted would expect further instructions from the user's terminal.

INITIALIZATION OF TED_COMS

Ted provides a means through which a ted_com can be initiated directly from command level. Ted can be invoked in the following manner:

```
ted ted_com
```

The above command is equivalent to entering ted with the simple command:

```
ted
```

and immediately executing a series of requests such as:

```
b (exec)
r ted_com.ted
b0
\b (exec)
```

This request sequence reads the initial ted_com segment into buffer exec, changes the current buffer back to buffer 0, and executes the contents of buffer exec.

The ted command can also be invoked with more than one argument. Thus, the command line:

```
ted get path
```

is the equivalent of:

```
ted
b (exec)
r get.ted
b (args)
a
path
\f
b (arg1)
a path\f
b0
\b (exec)
```

The above is only a partial equivalence since neither b(args) nor b(arg1) are created until first referenced. Because an argument is present, these buffers have a "right" to exist, but are not created unless needed. The presence of these buffers is not apparent to the user (i.e., the buffers are not displayed upon execution of an x request) until they have been used.

If the contents of get.ted is:

```
r \b(args)
```

then the contents of the exec and args buffers become:

exec	args
r \b(args)	path

and the request \b(exec) reads the segment path into buffer 0. The editor then waits for further commands from the user.

With the same contents of get.ted, the invocation:

```
ted get path 1,$s/x/y/ w q
```

enters the following into the exec and args buffers:

exec	args
r \b(args)	path 1,$s/x/y/ w q

This causes ted to read the segment path into buffer 0, substitute the character y for every occurrence of x, write out the buffer to the segment path, and then quit and return to command level.

NOTES ON TED_COM USE

There is no safeguard to prevent ted from modifying a ted_com buffer which is currently executing (i.e., a self-modifying ted_com). If this is attempted, havoc can well be the result, unless the line which does the modifying ends with a > (goto) request.

Refer to Section 4 and the % (call buffer with optional arguments) request in Appendix B for other examples of ted_coms.

When a ted_com is running and a search-fail condition is encountered (either substitute or address search), this is not considered an error. It causes the current level of execution to be terminated and execution continues where the ted_com was called from. This could result in ted sitting there waiting for the user to enter something and the user sitting there waiting for ted to finish. (Refer to the "-abort" argument in Appendix A for additional information on this situation.)

CONDITIONAL EXECUTION

A number of different ted requests test for various conditions and then execute according to the specific request. Each of the following are fully described in Appendix B.

`# <REST>`
: test current buffer, and if it contains data, execute <REST>. (See "Note" below.)

`ADR# <REST>`
: test current line, and if it is ADR, execute <REST>. (See "Note" below.)

`ADR1,ADR2# <REST>`
test current line, and if it is in the range of ADR1 through ADR2, execute <REST>. (See "Note" below.)

`ADR1,ADR2*/<RE>/ <REST>`
search the addressed text for the <RE>, and if found, execute <REST>. (See "Note" below.)

`ADR1,ADR2zif {<eval>} <REST>`
test <eval>, and if other than "0" or "false" is returned, execute <REST>.

`ADR1,ADR2gX/<RE>/`
global-inclusive, goes through the buffer a line at a time searching for the <RE> and does X to the line if a match is found. X can be =, p, !p, P, or d.

`ADR1,ADR2vX/<RE>/`
global-exclusive, goes through the buffer as above and does X to the line if a match is not found. X can be =, p, !p, P, or d.

`ADR1,ADR2g*`
global-if, goes through the buffer as above testing for logical conditions made up of <RE>s and <eval>s; if a "true" results, then a number of operations may be done.

Note: The "'" and "^" characters may be used in the above case to invert the sense of the request (e.g., # or '# or ^#).

EXTERNAL FUNCTIONS

Values of active functions may be obtained while in ted. Active functions are described in the New Users' Introduction.

Executing "ted_act" from within ted results in args being placed into b(act) of the current invocation of ted. These args may contain abbreviations and/or active functions. If multiple arguments are received by ted_act, they are placed into b(act) separated by a vertical tab (\013). The user may then change this to whatever is wanted. For example,

```
!   ted                                   -invoke ted
!   x                                     -display the status of buffers
        0 ->      (0)
!   e ted_act [date_time]                 -invoke ted_act
!   x
        0 ->      (0)
        1         (act)
!   t!\b(act) is today! 1                 -type string to user_output
    05/23/80 1408.7 mst Fri is today
!   t|Or should I say \b(act,1(1,8)), perhaps?| 1
    Or should I say 05/23/80, perhaps?
!   b(act)                                -set current buffer
!   1P                                    -print
        1    05/23/80  1408.7 mst Fri<no NL>
!   U/./ p                                -uppercase and print
    05/23/80  1408.7 MST FRI<NL>
                                          -NL required only for purpose of this
                                           example
!   x
        0         (0)
        1 ->      (act)
!   bi x                                  -change current buffer and status
        0         (0)
        1         (act)
        0 ->      (1)
!   a                                     -append
!   \b(act) will be on a friday.
!   \f                                    -terminate input
!   p
    05/23/80  1408.7 MST FRI will be on a friday.
!   x
        0         (0)
        1         (act)
        1 ->      (1)
!   f(ACT) e date_time                    -fileout into buffer and execute
!   x
        0         (0)
        1         (act)
        1 ->      (1)
        1         (ACT)
!   b(ACT) 1P
        1    05/23/80  1414.0 mst Fri
    " there is a NL here, while b(act) had none
!   q
    r 14:23 13.8 386
```

SECTION 3

USING DEBUG FEATURES

There are several debugging features which make the checking efforts of a string processing program much easier. They are:

```
-trace          -break        -label
-no_trace       -no_break     -no_label
-trace_edit     -pause
-trace_input
```

If any of the tracing control_args are supplied as part of the ted invocation, the current state of tracing is saved, the specified state is set, and ted is executed. When leaving ted, the saved state is restored. If none are specified, the execution is under whatever state is current.

GENERATING A TED_COM

The following ted_com provides a simple question and answer session. It asks for the square root of 49 and expects a user response of "7". Any response other than "7" results in the ted_com looping until the right answer is supplied. The ted_com shown below is purposely designed to force a looping sequence for testing purposes. (Looping starts and continues indefinitely if the first user response is not equal to 7.) The missing element in the ted_com which forces the looping is associated with line 5. Had the additional request "1,$d" been provided between the "l" and ">" requests (to clear the buffer after each user-supplied response), then the program would terminate upon any response of "7".

Bulk input mode (!a) is used to enter the ted_com as it is easier to enter the input functions (\b and \r). Several of the lines below include line numbers which are included for explanation purposes (i.e., references are made to these line numbers in the next example's discussion).

```
    !   ted
    !   !a
    !   :(loop)
    !   t|Enter the square root of 49.| 1
    !   a \r\f
    !   zif {\b(0)=7} t|That's right!| 1 q
    !   t|Sorry, \b0 is not right. Try again.| 1 >(loop)
    !   .
    !   1,$P
1           1    :(loop)
2           2    t|Enter the square root of 49.| 1
3           3    a \r\f
4           4    zif {\b(0)=7} t|That's right!| 1 q
5           5    t|Sorry, \b0 is not right. Try again.| 1 >(loop)
    !   w math.ted
    !   q
        r 09:07 1.315 256
```

USING TRACE

The -trace control_arg turns on the trace mechanism so that when a ted_com is invoked, each line of the program is displayed (including the "mode in" where applicable) before the line is executed. The following example uses the ted_com generated and stored in "math.ted" above. Line numbers are shown on the example for purposes of commentary immediately following the example.

```
6   !   ted -trace math
7       **EDIT**  b(exec)  r math.ted
8       **EDIT**  b0 :(loop)
9       **EDIT**  t|Enter the square root of 49.| 1
10      Enter the square root of 49.
11  !   99
12      **EDIT**  a 99\f
13      **EDIT**  zif {99=7} t|That's right!| 1 q
14      **EDIT**  t|Sorry, 99 is not right. Try again.| 1 >(loop)
15      Sorry, 99 is not right. Try again.
16      **EDIT**  :(loop)
17      **EDIT**  t|Enter the square root of 49.| 1
18      Enter the square root of 49.
19  !   7
20      **EDIT**  a 7\f
21      **EDIT**  zif {997=7} t|That's right!| 1 q
22      **EDIT**  t|Sorry, 997 is not right. Try again.| 1 >(loop)
23      Sorry, 997 is not right. Try again.
24      **EDIT**  :(loop)
25      **EDIT**  t|Enter the square root of 49.| 1
26      Enter the square root of 49.
27  !   \f
28      **EDIT**
29  !   o ^trace
30      **EDIT**  o ^trace
31  !   b(exec) 4
32      zif {\b(0)=7} t|That's right!|1 q
33  !   !i
34  !   \037
35  !   .
36  !   1,$P w
37          1    :(loop)
38          2    t|Enter the square root of 49.| 1
39          3    a \r\f
40          4    \037
41          5    zif {\b(0)=7} t|That's right!| 1 q
42          6    t|Sorry, \b0 is not right. Try again.| 1 >(loop)
43  !   q
44      Modified buffers exist:
45        (0)
46
47  !   ted: Do you still wish to quit?    yes
48      r 09:08 4.366 408
```

line 6
Run the ted_com with the input and edit trace options on.

line 7-10
Lines are displayed one-at-a-time before use.

line 11
First user response.

line 12-18
Lines are displayed one-at-a-time.

line 23
It should be obvious at this point that the buffer was not cleared before testing (line 4 of the ted_com "zif" request), but for the sake of the example we will debug it "as is."

line 27
The \f input, in reply to the \r request (see line 3 of example), stops execution of the buffer when it appears on a line by itself.

line 29
Turn off (stop) tracing mechanism. From this point (lines 27 through line 49) the program is being modified to be set up for breakpoints.

line 31
Change to b(exec), and examine line 4 (previous example), the one that you would expect to make the first breakpoint on.

line 32
Verifies the contents of line 4 of the ted_com.

line 33
Enter a breakpoint before line 4 using "!i". This is accomplished in bulk input mode to ensure that it actually gets input. A \037 is discarded if in EDIT mode with break mode off (see "Using Breakpoints" below). You cannot set a breakpoint with:

```
4s/^/\037/
```

but you can with:

```
4s/^/\r/      -or-     4s/^/\c\037/
\037
```

line 35
Terminate the insert request (!i) on line 33.

line 36
Verify that the breakpoint is in place, then write the buffer back to the file. In the "breakpoint" example below it is now set up to stop immediately after the user response to the math ted_com request for input.

line 43
The intent here is to quit, but because the ted_com caused b0 to be modified, the ted request "q" complains with a response (lines 44 through 47).

line 47
The "yes" reply (end-of-line) informs ted that the user wishes to quit.

USING BREAKPOINTS

The breakpoint is used for stopping a program for examination (when embedded within the ted_com text). When the \037 is included (at the user's option) in a ted_com, and the break mode is enabled (i.e., the break option is set), the program stops at each \037, displays the break

message, and waits for user intervention. The following message is displayed when a breakpoint is encountered in the break mode:

```
BREAK:  b(X), line n, level m[dd]
```

where:

X	is the buffer being executed
n	is the line being executed
m	is the buffer invocation depth
dd	is the ted recursion number

From this point the user may enter most ted requests. However, if input mode is entered, it must also be exited on the same line. For example, "$a q\012\f" adds a line to the end of the current buffer consisting of a "q". Note that the input mode begins and ends on the same line. To continue after a break, enter "><NL>" which is the "goto" request (i.e., restart at point of interruption).

If \? is entered while at a BREAK, ted redisplays the break message followed by the line on which the break occurred with the point of break being indicated. Therefore it is best to place a break on a line with a request instead of on a line by itself.

To turn the breakpoint mechanism on, enter:

```
ted -break path
```

To turn it off, enter:

```
ted -no_break path
```

The breakpoint facility can also be set/reset using the o request (refer to Appendix B for a description of the break and ^break options).

The break character (\037) is recognized only in edit mode and functions as follows:

nobreak mode
: all occurrences of \037 are ignored.

break mode
: all occurrences of \037 are treated as NLs (i.e., the current request line is terminated at this point and is then executed). When the next input line is asked for, the break message is displayed. The data following the \037 begins a new request line after continuation from the breakpoint. This means that a breakpoint should not be set after a conditional request. If a breakpoint is placed in the fashion:

```
zif {1>2} +1d \037 -2d
```

then the following would occur:

1. get request line "zif {1>2} +1d"
2. execute it. Since the condition is false, the "+1d" is ignored.
3. BREAK...
4. continue from break
5. get request line "-2d"
6. execute it

As you can see, without the breakpoint, neither "d" request would be executed, but with the breakpoint, the second "d" request is executed.

Note: A literal \037 may be entered by concealing it with a \c.

The following is a continuation of the "trace" example which was set up (by the requests in line 29 through 35 above) to stop at a breakpoint immediately after the user response to the math ted_com request for input. Commentary associated with the example immediately follows, and relates to the line numbers shown.

```
1   !   ted -break math
2       Enter the square root of 49.
3   !   99
4       BREAK: b(exec), line 4, level 1[01]
5   !   1,$p
6   !   99  x
7           1 -> mod (0)
8           6         (exec) >udd>Multics>math.ted
9   !   >
10      Sorry, 99 is not right. Try again.
11      Enter the square root of 49.
12  !   7
13      BREAK: b(exec), line 4, level 1[01]
14  !   1,$p
15  !   997  Q
16      quit during BREAK
17      r 09:09 2.764 227
```

line 1
Run math ted_com with breakpoints enabled.

line 4
The breakpoint is encountered.

line 5
Examine the current buffer contents.

line 6
The "x" request (end-of-line), invoked by the user, requests ted to display buffer status (i.e., a pseudo "where-are-we" request).

line 9
Continue the program at the point of interruption.

line 13
Breakpoint encountered again.

line 14
Examine the current buffer contents. At this point it should be obvious that the "old data" is still in the buffer.

line 15
The "Q" request (end-of-line), invoked by the user, informs ted that the user wishes to quit-force from ted and return to Multics command level. Quit-force is used since the source of the problem is known. At this point the ted_com writer may re-edit the math

ted_com, make the necessary correction alluded to in "Generate a ted_com" above (insert the "1,$d" request), and proceed.

Another example of using breakpoints follows. The ted_com "breaking.ted" consists of:

```
t!first! l
t!second ! \037 t|third| l t!4th...! \037 t|5th| l
\037 t:last---: l q
```

Now we run it with and without breaks enabled.

```
1   !   ted -break breaking
2       first
3       second BREAK: b(exec), line 2, level 1[01]
4   !   \?
5       BREAK: b(exec), line 2, level 1[01]
6       2 t!second ! \037 <BREAK> t|third| l t!4th...! \037 t|5th| l
7
8   !   >
9       third
10      4th...BREAK: b(exec), line 2, level 1[01]
11  !   \?
12      BREAK: b(exec), line 2 level 1[01]
13      2 t!second ! \037 t|third| l t!4th...! \037 <BREAK> t|5th| l
14
15  !   >
16      5th
17      BREAK: b(exec), line 3, level 1[01]
18  !   \?
19      BREAK: b(exec), line 3, level 1[01]
20      3   \037   <BREAK>   t:last---: l q
21
22  !   >
23      last---
24      r 07:58 1.831 97
25
26  !   ted -no_break breaking
27      first
28      second third
29      4th...5th
30      last---
31      r 07:59 0.543 28
```

line 1
Run the ted_com, with breaks enabled.

line 2
The result of executing the first line of the ted_com.

line 3
The result of executing the first part of line two (second) in the ted_com. The response following "second" from ted indicates a breakpoint has been encountered.

line 4
Now we don't remember what is on line 2, so we ask where we are.

line 5
This tells the same thing as before.

line 6
This shows what line 2 of the ted_com looks like, with the marker "<BREAK>" (with an HT on either side) showing which break you are currently at.

line 7
Finishes up the response.

line 8
We tell ted to continue.

line 9
The next part of the line gets executed.

line 10
The next part executed (4th...) followed by another break message.

line 11
Again we ask where we are.

line 13
This is just like before, except that the second breakpoint is marked.

line 15
Continue again.

line 16
Next execution.

line 20
Now we see that we are at the beginning of line 3. This example shows why it is not advisable to put breaks on separate lines (e.g., the "\?" would not be able to give you any clues as to where you are).

line 23
Finish up the execution.

line 26
Now let's run it straight through.

line 27-30
The uninterrupted execution output.

USING PAUSE

The pause mechanism can be invoked at any time while a ted_com is being executed to cause a pause at the next request time. Any ted edit requests can then be executed except \b and \r.

The "><NL>" request restarts the program at the point of interruption. For example, assume that a program is running that is suspected of looping. The program can be stopped by pressing the QUIT key and entering the command:

```
ted -pause
```

Note the line being executed, enter "><NL>" (to restart), and repeat the sequence several times to verify that looping is actually occurring.

The following is a "do-nothing" example used to describe the use of the pause mechanism. Commentary associated with the example immediately follows.

```
1   !   ted
2   !   a
3   !   !a
4   !   nothing but the best,
5   !   my dear!
6   !   .
7   !   1n
8   !   :(loop)
9   !   s/n/n/
10  !   >(loop)
11  !   \f w loopforever.ted
12  !   q
13      r 09:10 31.121 45
14
15  !   ted loopforever
16  !   <(hit QUIT)>
17      QUIT
18      r 09:10 1.121 42 level 2
19
20  !   ted -pause
21      BREAK: b(exec), line 7 level 1[01]
22  !   x
23           2 -> mod (0)
24           8        (exec) >udd>Multics>loopforever.ted
25  !   b(exec) 1,$P
26           1    !a
27           2    nothing but the best,
28           3    my dear!
29           4    .
30           5    1n
31           6    :(loop)
32           7    s/n/n/
33           8    >(loop)
34  !   Q
35      quit during BREAK
36      r 09:11 1.400 194
```

lines 1-14
These lines are associated with the generation of the ted_com.

line 15
Invoke ted_com loopforever.

line 16
This is not a ted response, but a user-initiated action of "pressing the QUIT key on the terminal."

line 20
tells ted a break-in is requested.

line 21
Ted responds as if a breakpoint had just been encountered.

line 22
Display buffer status (i.e., a pseudo "where-are-we" request).

line 25
Display the contents of b(exec)—the ted_com being executed.

line 34
Execute Quit-force (the problem is identified). The user could fix the ted_com at this point before quitting and save system overhead by not reinvoking ted a second time.

SECTION 4

EVALUATIONS

Evaluations are a process of doing arithmetic and/or string operations within ted. The user may place numbers or strings into variables and then use these variables at a later time, but within the same invocation of the ted command (i.e., variables are destroyed when the ted invocation is exited). This facility allows the user to implement counters, parse strings, build strings, etc. Braces "{}" must always be used to delimit evaluations normally referred to in this manual as "<eval>".

Refer to Appendix C for a description of literals used within evaluations. Metasymbols used throughout this section include:

`<aexp>`
: Arithmetic expression within an <eval>.

`<assign>`
: Assignment within an <eval>.

`<cat>`
: Concatenation within an <eval>.

`<data>`
: Data elements within an <eval>.

`<eval>`
: Evaluation.

`<factor>`
: Arithmetic factor within an <eval>.

`<lexp>`
: Logical expression within an <eval>.

`<lpart>`
: Last part within an <eval>.

`<mask>`
: Fak conversion options within an <eval>.

`<number>`
: Any numeric digit or combination of digits made up of 0-9.

`<part>`
: Part within an <eval>.

`<term>`
: Arithmetic term within an <eval>.

`<ucat>`
: Unary-operator expression within an <eval>.

USE OF EVALUATIONS

Following is a brief description of the use of evaluations.

Requests

The evaluation request is not expected to return any value. It may be given an address, or an address pair (to define a range), however it does not have a default address. This lack of a default address allows evaluations within an empty buffer.

`{ <eval> }`

Does not expect <eval> to return anything; its purpose is to just "do" something.

`zif {<eval>}`

Does expect a result so it can test for true/false.

Note: Neither of these may reference the buffer-related built-ins (lb, sb, se, le) unless an address is present.

"Not expecting a value" means that ted prints a warning message if a value is returned. "Expecting a value" does not mean that ted complains if nothing is returned, it merely uses a null value in whatever context it is. (Refer to Appendix B for additional detail regarding the above ted requests.)

Input Function

`\{ <eval> }`

Does expect a value to be returned; this value is used as if it replaced the \{<eval>}.

However, no value need be returned. A way to fill in a form is:

```
a
\{"Name: ":}Name: \r
\{"Addr: ":}Addr: \r
\{"Age:  ":}Age:  \r
\f
```

This is using the "print" functionality of evaluation (a string in quotes followed by a colon) to supply a prompt for the "\r"s embedded in the ted_com.

Editing Function

Can be used to reference the value of the string matched by the "g*" requests.

`\g{ <eval> }`

Does expect a value to be returned.

"Not expecting a value" means that ted prints a warning message if a value is returned. "Expecting a value" does not mean that ted complains if nothing is returned, it merely uses a null value in whatever context it is. (Refer to Appendix B for additional detail regarding the above ted requests.)

EVALUATION DESCRIPTION

Following is a BNF-like description of the evaluation syntax. The meta language does not use "|" to mean OR.

<eval> An Evaluation

{ <part>... <lpart> }

An evaluation may be a list of <part>s followed by an <lpart>.

where "parts" are used-up by either assigning or printing. An lpart is used-up by either assigning or returning. If lpart was assigned, null is returned. If no lpart, null is returned. The contexts vary as appropriate for the type. Note that spaces may be included between tokens in an <eval>; space can be before or after ":=" but not within.

```
zif {i := i+3;  j<i} >a

{i:=i+2}

\{fs("Now is the time for", i,7)}

\{}
```

<part> Evaluation Part

<cat> ;

The value of <cat> is printed followed by an NL.

<lexp> ;

The value of <lexp> is printed as either "true" or "false" followed by an NL.

<cat> :

The value of <cat> is printed without an NL.

<lexp> :

The value of <lexp> is printed as either "true" or "false" without an NL.

<assign> ;

The value is "used up" by being assigned, so there is no further action.

d(<cat>) ;

This requests a dump of variables. <cat> must result in a string consisting of only the characters "akKv". If "a" is present, all non-zero a[*] are shown. If "k" is present, all non-null k[*] are shown. If "K" is present, all non-null K[*] are shown. If "v" is present, all user-defined variables are shown regardless of content (type). (Refer to "Variable" below for a description of a [*], k[*], and K[*] variables.)

<lpart> Evaluation Last Part

<cat>

The value of <cat> is returned as a character string.

<lexp>

The value of <lexp> is returned as either "true" or "false".

<assign>

A null value is returned.

If no <lpart>, a null is returned.

<lexp> Logical Expression

`<cat> <rel> <cat>`

This compares two values. The result is always either "true" or "false". If the <cat>'s are not of the same type, conversion is done to the "higher" of the two types. The order, low to high is:

arithmetic
logical
string

`<rel>`

Is any of these operators: =, ^=, >=, <=, <, or >.

<cat> Concatenate Operation

`<cat> || <aexp>`

This means a series of expressions may be concatenated. If any are not strings, they are converted.

`<aexp>`

There may be no concatenation. In this case, no conversion is done.

`<cat> <aexp>`

This is an obsolete form, present only for compatability. It has the same meaning as <cat>||<aexp>.

<assign> Assignment

`a[ <cat_1> ] := <cat_2>`

The value of <cat_2> is assigned to an element of the "a" numeric array. <cat_2> and <cat_1> must both be convertible to a number.

`k[ <cat_1> ] := <cat_2>`

The value of <cat_2> is assigned to an element of the "k" short string array. <cat_1> must be convertible to a number. <cat_2> is converted to a string.

`K[ <cat_1> ] := <cat_2>`

The value of <cat_2> is assigned to an element of the "K" long string array. <cat_1> must be convertible to a number. <cat_2> is converted to a string.

`<var> := <cat>`

The value of <cat> is assigned to the variable <var>. The type of <var> is determined by the type of <cat> and may change from assignment to assignment.

An example of each type:

```
a[10]:= p[2]+1
k[a[10]]:= p[3]
K[fs(p[3],1,1)]:= p[1]||"-"||p[3]+a[10]
i:=0
```

A variable name can be from 1 to 16 characters, beginning with a letter and consisting of letter, digits, and underscore. A few names are reserved because they are the names of built-ins (i.e., bn, dn, and cs). There is no conflict with function references because they end with "(" (i.e., fs(, fln(, if().

<aexp> Arithmetic Expression

`<aexp> + <term>`
Terms may be added. They must be convertible to numbers.

`<aexp> - <term>`
Terms may be subtracted. They must be convertible to numbers.

`<term>`
A term may be left unchanged.

<term> Arithmetic Term

`<term> * <factor>`
Factors may be multiplied. They must be convertible to numbers.

`<term> / <factor>`
Factors may be (integer) divided. They must be convertible to numbers.

`<term> | <factor>`
Factors may be mod'ed (remainder when dividing). They must be convertible to numbers.

`<factor>`
A factor may be left unchanged.

<factor> Unary-Operator Expression

`+ <fact>`
A unary plus may precede a <fact> which must be convertible to a number.

`- <fact>`
A unary minus may precede a <fact> which must be convertible to a number.

`<fact>`
There may be no unary-operator at all. The type may then be anything.

<fact> Arithmetic Factor

`<data>`
A factor may be one of several types of data, either literal, built-in, variable, or function.

`( <cat> )`
It may also be an expression in parentheses. There is no implied conversion by the fact that it is used in this fashion.

`( <lexp> )`
A factor can be an <lexp>.

`( <assign> )`
The specified assignment is done and the value assigned is also "passed on" for subsequent use in the context where it occurs.

```
\{(a:=a+1)}
```

has the same effect as

```
\{a:=a+1;a}
```

Following is an alternate example to that given in Section 2, "Input Functions" for the \{} sequence.

```
{ln:=0}
a \{(LN:=fak((ln:=ln+10),"00000")): LN} \r
\f >-1
```

<data> Data Elements

Data elements consist of a number of items which are categorized as:

literal
variable
buffer
addressed
addressed functions
string
parameters
string functions
arithmetic functions

These items have a further breakdown and are listed below. Each is prefixed with a reference number. Following the "Arithmetic Functions" grouping below, is an alphabetized list of all items with a cross reference to this number.

LITERAL

1. <integer>
   Numeric data may be in decimal. ALL numeric data is carried as fixed bin(35,0).
2. "..."o
   Numeric data may be given as an octal integer string. The string must contain only the characters "01234567" (e.g., "377"o).
3. "..."x
   Numeric data may be given as a hexadecimal integer string. The string must be composed of only these characters, "012345678abcdefABCDEF" (e.g., "1F72"x).
4. "..."
   String data may be entered as a quoted string. Any internal quotes must be doubled (e.g., ""smartly""). A string has no inherent max length. It must fit in a segment with all other data which exists (including internal temporaries) at any given time.

VARIABLE

5. <var>
   Reference to a user-defined variable. The variable must have previously been assigned some value. Its type is dependent upon the last assignment.
6. a[<cat>]
   A reference to an element of the arithmetic array. <cat> must be convertible to a number in the range -200:200.

7. k[<cat>]
   A reference to an element of the short-character array. Each element is restricted to 32 characters. <cat> must be convertible to a number in the range -200:200.

8. K[<cat>]
   A reference to an element of the long-character array. Each element is restricted to 500 characters. <cat> must be convertible to a number in the range -10:10.

BUFFER

These buffer built-ins do not give correct results when used in \{} context. For example:

```
a \{en}.pll
```

does not work. However,

```
{e:=en}
a \{e}.pll
\f
```

does work.

9. bn
   The name of the current buffer.

10. dn
    The directory name of the file associated with the current buffer. Null if no pathname on the buffer.

11. en
    The entry name of the file associated with this buffer. Null if no pathname on the buffer.

12. sk
    The subfile type of the file associated with this buffer.

    ":" archive component
    "|" superfile component (obsolete)
    " " no subfile

13. sn
    The archive component name (subfile name) of the file associated with this buffer. Null if no archive component name associated with buffer.

ADDRESSED DATA

These data are available in { } and zif { }, but only if an address is present. For example:

```
/abc/{a:=sb}
0(\{a}) p
```

prints "a" and

```
/abc/{fs(Kb,sb,1);}
```

also prints "a".

14. Ks
    The string addressed. If no byte-address is given, then this is identical to Kl.

15. `Kl`
    The line addressed. Or more precisely, it begins at the beginning of the line where Ks begins and ends at the end of the line where Ks ends. It is possible for this to span lines; it is all dependent on the address given.

16. `Kb`
    The contents of the whole buffer.

17. `lb`
    The byte index in the buffer of the place where Kl starts. (The first character of a buffer is index 1.)

18. `sb`
    The byte index in the buffer of the place where Ks starts.

19. `se`
    The byte index in the buffer of the place where Ks ends.

20. `le`
    The byte index in the buffer of the place where Kl ends.

21. `be`
    The byte index in the buffer of the place where the buffer ends (i.e., the number of characters in the buffer).

ADDRESSED FUNCTIONS

There can be no spaces between the function name and the "(". This is true of all functions.

22. `fln( lb )`
    Yields the line number of where lb is.

23. `fln( le )`
    Yields the line number of where le is.

24. `fln( be )`
    Yields the line number of where be is (i.e., the number of lines in the buffer).

STRING

25. `cs`
    A string containing the ASCII collating sequence (PL/I collate9()).

PARAMETERS

26. `ag`
    The number of arguments which were given when ted was called.

27. `p[ <cat> ]`
    A reference to an element in the parameter array. <cat> must be convertible to a number in the range 0:pn. p[0] is a special case which represents the whole string following the buffer name in the % request. The first character of this string is the delimiter of the parameters.

28. pn

    Specifies how many buffer parameters are present. Parameters are present only when a buffer is called via the % request. When a buffer is invoked via \b, this value is 0.

29. em

    The full text of the last error message, whether printed or not. Messages have the form:

    ```
    Xxx) text
    ```

    where Xxx is the message identifier and text is the part which is normally printed.

30. emt()

    Returns the message text, as described under em.

STRING FUNCTIONS

31. fs(<cat_1> , <cat_2>)

    Yields a substring of <cat_1> which is converted to string, if necessary. <cat_2> is converted to a number which specifies where to begin taking the string. If negative, it means to count from the end of the string. The string returned is everything after the beginning point (inclusive).

32. fs(<cat_1> , <cat_2> , <cat_3>)

    Yields a string of a specified length. <cat_1> and <cat_2> are as described above. <cat_3> is converted to a number which specifies the length of the string to return. If there are not sufficient characters in <cat_1>, then the result is padded to the required size with spaces. If <cat_3> is negative, and padding is needed, it is done on the left.

33. fs(<cat_1> , <cat_2> : <cat_3)

    Similar to the form above, except that <cat_3> specifies the ending POSITION of the string to take. If <cat_3> is negative it means to count from the right end of the string.

34. frs(<cat_1> , <cat_2> , <cat_3>)

    Rearranges a string. First an index of <cat_2> in <cat_1> is done. This breaks the <cat_1> into parts which may then be selected by the contents of <cat_3> which must consist of only these characters:

    "b" give the before part. If no match, this is <cat_1>.
    "m" give the matched part (this is null if no match).
    "a" give the after part.
    "s" give the value of <cat_2>

    This means that:

    ```
    frs(name,".pll","bs")
    ```

    yields a string with the suffix ".pll", taking into account the presence of one to start with.

35. if(<lexp> , <cat_1> , <cat_2>)

    Yields <cat_1> if <lexp> is TRUE. Otherwise, it yields <cat_2>.

36. if(<lexp> , <cat>)

    Yields <cat> if <lexp> is TRUE. Otherwise, it yields nothing.

ARITHMETIC FUNCTIONS

37. `fka( <cat> )`
    Converts a string to arithmetic. It is included for compatability only.

38. `fl( <cat> )`
    Yields the length of <cat>. If <cat> is not a string, it is converted.

39. `fi( <cat_1> , <cat_2> )`
    The index function. Both arguments are converted to strings. The value returned is the position in <cat_1> where string <cat_2> is found. The result is 0 if it is not there (PL/I index).

40. `fir( <cat_1> , <cat_2> )`
    Does an index function on the reverse of <cat_1>.

41. `ff( <cat_1> , <cat_2> )`
    Finds a character of <cat_2> in <cat_1>. If none exist, 0 is returned (PL/I search()).

42. `ffr( <cat_1> , <cat_2> )`
    Finds, except it looks from the end of the string (PL/I search(reverse())).

43. `fv( <cat_1> , <cat_2> )`
    Verifies the presence of some character in <cat_2> in <cat_1>. If all match, 0 is returned (PL/I verify).

44. `fvr( <cat_1> , <cat_2> )`
    Verifies except it looks from the end of the string (PL/I verify(reverse())).

45. `ff( <cat> , S(xxx) )`
    Finds a member of a specified set of characters which is specified by the value of xxx, which is made up of one or more of the selectors:

    "n" numeric
    "u" uppercase letter
    "l" lowercase letter
    "a" letters plus "_"
    "o" octal
    "x" hexadecimal
    "g" graphic, \041->\176
    "m" motion, BSP HT NL VT FF SP

46. `ffr( <cat> , S(xxx) )`
    Finds, but from the end of <cat>.

47. `fv( <cat> , S(xxx) )`
    Verifies, with a set.

48. `fvr( <cat> , S(xxx) )`
    Verifies, but from the end of the string.

49. `fak( <cat_1> , <mask> )`
    Converts a number, <cat_1>, to a string under the control of a mask, <mask> (see below).

50. `fmx( <cat...> )`
    Returns the maximum of all the arguments given to it. The arguments are separated by commas.

51. fmn(<cat...>)
    Returns the minimum of all the arguments given to it.

52. mct()
    Returns the number of matches which occurred on the last non-g* substitute executed. Is -1 if no substitute has been done.

Following is an alphabetical listing (in ascii collating sequence) of all data elements shown above. The number preceding each item identifies the associated numbered item in text.

```
 4  "..."
 2  "..."o
 3  "..."x
 1  <integer>
 5  <var>
 8  K[ <cat> ]
16  Kb
15  Kl
14  Ks
 6  a[ <cat> ]
26  ag
21  be
 9  bn
25  cs
10  dn
29  em
30  emt()
11  en
49  fak( <cat_1> , <mask> )
45  ff( <cat> , S(xxx) )
41  ff( <cat_1> , <cat_2> )
46  ffr( <cat> , S(xxx) )
42  ffr( <cat_1> , <cat_2> )
39  fi( <cat_1> , <cat_2> )
40  fir( <cat_1> , <cat_2> )
37  fka( <cat> )
38  fl( <cat> )
24  fln( be )
22  fln( lb )
23  fln( le )
51  fmn( <cat...> )
50  fmx( <cat...> )
34  frs( <cat_1> , <cat_2> , <cat_3> )
31  fs( <cat_1> , <cat_2> )
32  fs( <cat_1> , <cat_2> , <cat_3> )
33  fs( <cat_1> , <cat_2> : <cat_3 )
47  fv( <cat> , S(xxx) )
43  fv( <cat_1> , <cat_2> )
48  fvr( <cat> , S(xxx) )
44  fvr( <cat_1> , <cat_2> )
36  if( <lexp> , <cat> )
35  if( <lexp> , <cat_1> , <cat_2> )
 7  k[ <cat> ]
17  lb
```

```
20  le
52  mct()
27  p[ <cat> ]
28  pn
18  sb
19  se
12  sk
13  sn
```

<mask> fak Conversion Options

The result of the conversion is exactly as long as the <mask>. The significant digits of the number and the <mask> are processed from right to left. All numeric, alphabetic, and special characters may appear in <mask>. However, some of the characters have special meanings:

"~"
: causes a space in the results.

"0"
: this (zero) is replaced by a digit if one is left; otherwise, it remains in the result.

" "
: is replaced by a digit if one is left; otherwise it is replaced by a fill character. After being used a fill character becomes a " " if it is not "*". All zeros should occur to the right of the spaces in the mask. The rightmost space or zero is the units position.

"-"
: in the units position, signals floating minus sign. The fill character is set to "-" or <SP>, as appropriate.

"-"
: in the leftmost position, signals a fixed minus sign. It is set to either "-" or <SP>, as appropriate.

"$"
: in the units position, signals floating dollar sign. The fill character is set to "$".

"*"
: in the units position, signals asterisk protection. The fill character is set to "*".

"o"
: in the units position, signals octal conversion is to be done on the value. The result is a string, not a signed value.

"x"
: in the units position, signals hexadecimal conversion is to be done on the value. The result is a string, not a signed value.

","
: is replaced by a fill character if no digits are left. Otherwise, it remains in the result.

"."
: is replaced by a fill character if no digits are left. Otherwise, it remains in the result.

Any character not having a special meaning remains in the result.

EVALUATION EXAMPLES

```
" examples of all kinds of evaluations
    50 ->     (exec) >udd>m>jaf>install>examples.ted|eval
          This is the data addressed in most of the following things:
          (the address will not be displayed each time, however)
   0(63),0(75)p ->
xyz
"12345678
   {var1:="123abc456"}
          Things which have arithmetic type
   {ff(var1,"bc");} -> 5
   {ffr(var1,"bc");} -> 4
   {fi("abcdedcba","c");} -> 3
   {fir("acbdefdeccba","c");} -> 3
   {fl(Ks);} -> 13
   {fln(lb);} -> 2
   {fln(le);} -> 3
   {fln(be);} -> 50
   {fv("3241","12468");} -> 1
   {fvr("3241","12468");} -> 4
   {fvr("3241","02468");} -> 1
   {lb;} -> 40
   {le;} -> 77
   {sb;} -> 63
   {se;} -> 75
   {be;} -> 2338
          Things which have character type
   0(63),0(75){Ks;} -> xyz
"12345678
          The line above actually prints without a NL at the end
   {fak(1234,"00,000");} -> 01,234
   {k[2]:="abcdefgGG"}
   {"+"||   k[2]       ||"+";} -> +abcdefgGG+
   {"+"||fs(k[2], 2   )||"+";} -> +bcdefgGG+
   {"+"||fs(k[2],-2   )||"+";} -> +GG+
   {"+"||fs(k[2], 4   )||"+";} -> +defgGG+
   {"+"||fs(k[2], 2, 4)||"+";} -> +bcde+
   {"+"||fs(k[2], 2,-4)||"+";} -> +bcde+
   {"+"||fs(k[2],-2, 4)||"+";} -> +GG  +
   {"+"||fs(k[2],-2,-4)||"+";} -> +  GG+
   {"+"||fs(k[2], 2: 4)||"+";} -> +bcd+
   {"+"||fs(k[2], 2:-4)||"+";} -> +bcdef+
   {bn   ;} -> exec
   {dn   ;} -> >udd>m>Falksenj>install
   {en   ;} -> examples.ted
   {em   ;} -> Xrq) Invalid request \011.
   {emt();} -> Invalid request \011.
   {sn   ;} -> eval
   {sk   ;} -> |
```

Following is an excerpt from the actual execution of an annotated ted_com. Each line of commentary is marked with:

"***"

The basic format is 1) a ted request, 2) the result of the request, and 3) some notes on what happened. This list illustrates just evaluations.

```
{ 34: "<-->": 19; "result"}
34<-->19
{ has result "result".
   "***" A value followed by a ":" is printed without a NL.  A
   "***" value followed by a ";" is printed with a NL.  A value
   "***" followed by a "}" is returned.  In this circumstance it
   "***" is an error.

{ a[1] := 1}
   "***" An arithmetic array is available, and is named "a".  You
   "***" may assign numeric values to it.

{ a[2]:= "0002"}
{ a[-40]:= (7=3)}
   "***" The values, however, need not be numbers, they just need
   "***" to be convertible to numbers.  The logical value of "(7=3)" is
   "***" converted to a "0" in order to be stored in a [-40]

{ a[3]:= "invalid"}
Bad decimal digit. "invalid"
   "***" It doesn't like it if it is not convertible.

{ a[400]:= 400}
Subscript not in a[-200:200].
   "***" There are not an infinite number of elements there.

{ k[1] := "anything at all"}
   "***" There is also a short character array, "k".

{ k[2] := 5280}
{ k[-40]:= (3=7)}
{ k[3] := "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"}
   "***" This array has room for 32 characters per element, so
   "***" this last one gets truncated.  The logical value of "(3=7)" is
   "***" converted to "false" in order to be stored in k[-40]

{ k[-225] := 1}
Subscript not in k[-200:200].
   "***" This also is not infinite in size.

{ K[8] := "something"}
   "***" There is an array of long character strings, "K".  It
   "***" has 500 character elements.
```

```
{ K[9] := k[1]}
{ K[3] := "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"}
   "***" So this doesn't get truncated.

{ K[100]:="" }
Subscript not in K[-10:10].
   "***" There are much fewer of these available.

{ a := 1; bf := "able to run"; collie:= (3>4)}
   "***" There are also arbitrarily named variables.  The names
   "***" are up to 16 characters in length.  The data type of one
   "***" of these is determined by the type of the last datum to
   "***" be assigned to it.  A variable is not defined until it
   "***" has been assigned a value.  A variable may not be
   "***" referenced until it is defined.

{em:= "zilch"}
Syntax- eval. "{em:= "
   "***" These names cannot conflict with the few reserved names
   "***" which exist.  If you try to assign a value to one of
   "***" these, it complains.

{fmn:=fmn(4,1,6)}
   "***" None of the function names are reserved, because the "("
   "***" IMMEDIATELY following (no whitespace) shows the
   "***" difference.

{d("a");}
a[   1] = 1
a[   2] = 2
   "***" You may request a display of the "a" array (all non-zero
   "***" elements are displayed)
{d("kK");}
k[ -40] = "false"
k[   1] = "anything at all"
k[   2] = "5280"
k[   3] = "abcdefghijklmnopqrstuvwxyzabcdef"
K[   3] = "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"
K[   8] = "something"
K[   9] = "anything at all"
   "***" You may request a display of the "k" and "K" arrays

{d("v");}
fmn = 1
collie = false
bf = "able to run"
a = 1
   "***" You may request a display of the user variables.  In
   "***" fact the quoted string which is an argument to the
   "***" display function is any combination of the characters
   "***" "akKv".
{ a:= bf ; bf:= "1234"; collie:=1234}
{d("v");}
```

```
fmn = i
collie = 1234
bf = "1234"
a = "able to run"
   "***" Note that the data types are now different.

bx a Now is the time for all good men
to come to the aid of their...
\f
   "***" Now we change to another buffer.

x
     4    mod (0)
     2 ->     (x)
   "***" This shows the status of all the buffers.
X
     2 ->     (x)
   "***" This shows the status of only the current buffer.


(10,25) { "|": Ks: "|"; }
|o the aid of the|
   "***" The built-in Ks references the string addressed.

(10,25) { "|": Kl: "|"; }
|to come to the aid of their...
|
   "***" The built-in Kl references the line addressed.

(10,25) { "|": Kb: "|"; }
|Now is the time for all good men
to come to the aid of their...
|
   "***" The built-in Kb references all of the current buffer.

{ fak( 135, "000.0");}
013.5
   "***" The function fak converts arithmetic to character under
   "***" the control of a mask.

{ fs( "123456789", 3);}
3456789
{ fs( "123456789", 3, 6);}
345678
{ fs( "123456789",-3);}
789
   "***" The function fs is like PL/I substr with an extension to
   "***" handle negative starting values (they count from the end
   "***" of the sting).
{ fs( "123456789", 3: 6);}
3456
{ fs( "123456789", 3:-3);}
3456
```

```
   "***" Also, there is this extension.  In this form, the 2nd
   "***" number represents the character position in the string
   "***" to end the string instead of the length. Again you can
   "***" count from the end of the string.

{ frs( "copydumplist","dump","ab");}
listcopy
   "***" The frs function "re-arranges" a string.  The first
   "***" argument is divided into 3 parts by indexing into it
   "***" with the second argument.  This then defines parts:
   "***"     before, match, after
   "***" The third argument must then be a string made of of only
   "***" the characters "bmas".  (s means "searched", in other
   "***" words, the value of the 2nd argument.  This 3rd argument
   "***" tells what pieces you want an in which order you want
   "***" them.  In this example, the after and then before is
   "***" selected.

{ if( 3>4, "zebra");}

{ if( 3<4, "zebra");}
zebra
{ if( 3>4, "zebra","lion mane");}
lion mane
{ if( 3<4, "zebra","lion mane");}
zebra
   "***" The if function returns its 2nd argument if the 1st
   "***" argument is TRUE otherwise it returns its 3rd argument,
   "***" if present.

{fs( cs,60:70);}
;<=>?@ABCDE
   "***" The function cs gives the value of PL/I collate().

{ bn;}
x
   "***" The built-in bn is the name of the current buffer.

{ dn;}

   "***" The built-in dn is the directory portion of the pathname
   "***" associated with the current buffer, if any.

{ en;}

   "***" The built-in en is the entry portion of the pathname
   "***" associated with the current buffer, if any.
{ sn;}

   "***" The built-in sn is the subfile/component portion of the
   "***" pathname associated with the current buffer, if any.

{ sk;}
```

```
   "***" The built-in sk is the subfile/component kind associated
   "***" with the current buffer, if any.
   "***" ":"-> archive component  "|"-> subfile  " "-> segment

{ em;}
Vxx) Syntax- eval. "{em:= "

   "***" The built-in em gives the current error message.  It has
   "***" this form:
   "***"     Xxx) text...

{ emt( );}
Syntax- eval. "{em:= "
   "***" This function returns the error message text, the
   "***" text...  part of the error message.

{ + 3*4;}
12
{ - 3*4;}
-12
{ - 5 + 1;}
-4
   "***" Unary +,- are available for forming expressions.  If the
   "***" expression following is not numeric, a conversion is
   "***" attempted.

{"abc"||"def";}
abcdef
   "***" The || operator is used for concatenation.

{"stmt"||13||" in error";}
stmt13 in error
   "***" If the arguments for concatenation are not character
   "***" strings, they are converted.
{ "abc" <"jaf";}
true
{  0013 < 9 ;}
false
{ "0013"<"9";}
true
{   "12"< 13;}
true
   "***" Logical expressions are available, also.  The data types
   "***" may be different.

{ 4+6;}
10
   "***" Addition
{ 4-6;}
-2
   "***" Subtraction
{ 4*6;}
```

```
24
   "***" Multiplication
{ 4/6;}
0
   "***" Division
{ 4|6;}
4
   "***" Modulo

{ 4* (3+7);}
40
   "***" Expressions may be enclosed in parentheses.

{ 4* (3<7);}
4
{ 4* (3>7);}
0
   "***" If a logical expression is used in an arithmetic context
   "***" it will assume the value 0 for false and 1 for true.

{i:=10;i;}
10
{ 4* (i:=i-5);i;}
20
5
   "***" An assignment may also be enclosed in parentheses.  The
   "***" value of it is the value which was assigned.  Note that
   "***" i=10 to begin with and i=5 after the expression is
   "***" calculated.
{ a[ 137/2 ];}
0
   "***" Any expression may be used for a subscript value.  The
   "***" subscript expression is that enclosed within the []'s.
   "***" Unset arithmetic array values default to 0.

{ ag;}
0
   "***" This built-in gives the number of arguments which ted was
   "***" called with.

{ pn;}
0
   "***" This tells the number of parameters present in the
   "***" current buffer invocation.  If the buffer were called by
   "*"  , then this is >=0.  Otherwise it is =0.

{p[0];}

   "***" This represents the whole parameter string present on a
   "*"  then this
   "***" is a null string.  The first character of this string is
   "*" the delimiter for this   call.
```

```
   "***" The values p[1] - p[pn] represent the various parameters
   "***" available to the current buffer invocation.  These are
   "***" "local" values and are not known to any buffers nested
   "***" from this one.

{ mct( );}
1
   "***" This function returns the number of matches found on the
   "***" last substitute done.  The value is -1 if no substitute
   "***" has been done yet in this invocation of ted.

 { lb;}
Not defined- lb "{ lb"
   "***" This and the next 3 built-ins are not defined when no
   "***" address is given.

.{ lb;}
34
   "***" This built-in gives the character count in the current
   "***" buffer of where the addressed line begins.

.{ sb;}
34
   "***" This built-in gives the character count in the current
   "***" buffer of where the addressed string begins.
.{ se;}
64
   "***" This built-in gives the character count in the current
   "***" buffer of where the addressed string ends.

.{ le;}
64
   "***" This built-in gives the character count in the current
   "***" buffer of where the addressed line ends.

{ be;}
64
   "***" This built-in gives the character count in the current
   "***" buffer of where the buffer ends, i.e.  the length of the
   "***" buffer.

{ fmx(1,10,-3,6);}
10
   "***" The function fmx is the max function.
{ fmn(1,10,-3,6);}
-3
   "***" The function fmn is the min function.
{ fmn(fmx(1,10),fmx(-3,6));}
6
   "***" They may be nested.

{bf := "Smithsonian Institute"}
{ fl( bf );}
```

```
21
   "**** The function fl gives the length of its argument.

{ ff( bf , "ti" );}
3
   "**** The function ff is the "find" function, the PL/I search
   "**** built-in.

{ ffr( bf , "ti" );}
2
   "**** Function ffr is find-reverse.

{ fi( bf , "ti" );}
16
   "**** Function fi is the PL/I index function.

{ fir( bf , "ti" );}
4
   "**** Function fir is index-reverse.
{ fv( bf , "Smite" );}
5
   "**** Function fv is the PL/I verify function.

{ fvr( bf , "Smite" );}
3
   "**** Function fvr is verify-reverse.

{ ff( bf , S(l) );}
2
{ ffr( bf , S(u) );}
9
{ fv( bf , S(ul) );}
12
{ fvr( bf , S(ul) );}
10
   "**** These 4 forms are like mentioned above except that the
   "**** set to look for is specified generically instead of
   "**** literally.  Within the S() may be these selectors:
   "****   u - upper case        l - lower case
   "****   a - ul plus "_"       n - decimal digits
   "****   o - octal digits      x - hex digits
   "****   m - motion character  g - graphic character

.{ fln( lb );}
2
   "**** This function gives the line number where the beginning
   "**** addressed line is.

.{ fln( le );}
2
   "**** This gives the line number where the ending addressed
   "**** line is.
```

```
{ fln( be );}
2
   "***" This gives the line number of the end of the buffer.
```

MASK EXAMPLES

```
" examples of masks in {fak(n,mask)}
a[   1] = 12345
a[   2] = -654321
a[   3] = -9


=====

Given a value of:       12345

If mask is      "    ,   . $"
then result is      $123.45

If mask is      "    ,   . *"
then result is  *****123.45

If mask is      "    ,   . -"
then result is       123.45

If mask is      "    ,   . 0"
then result is       123.45

If mask is      "    , 00.0*"
then result is  *****123.45

If mask is      "    , 00.0-"
then result is       123.45

If mask is      ""
then result is  12345

If mask is      "*f0f0f0f0f0"
then result is  *f1f2f3f4f5

If mask is      "-   ,   .  "
then result is       123.45

If mask is      "-   , 00.0*"
then result is   ****123.45

If mask is      "-   , 00.00"
then result is       123.45

=====
```

```
Given a value of:      -654321

If mask is      "    ,    . $"
then result is     $6,543.21

If mask is      "    ,    . *"
then result is   ***6,543.21

If mask is      "    ,    . -"
then result is     -6,543.21

If mask is      "    ,    . 0"
then result is      6,543.21

If mask is      "    , 00.0*"
then result is   ***6,543.21

If mask is      "    , 00.0-"
then result is     -6,543.21

If mask is      ""
then result is   -654321

If mask is      "*f0f0f0f0f0"
then result is   *f5f4f3f2f1

If mask is      "-   ,    .  "
then result is   -  6,543.21

If mask is      "-   , 00.0*"
then result is   -**6,543.21

If mask is      "-   , 00.00"
then result is   -  6,543.21

=====

Given a value of:           -9

If mask is      "    ,    . $"
then result is            $9

If mask is      "    ,    . *"
then result is   **********9

If mask is      "    ,    . -"
then result is            -9

If mask is      "    ,    . 0"
then result is             9

If mask is      "    , 00.0*"
then result is   ******00.09
```

```
If mask is      "    , 00.0-"
then result is       -00.09

If mask is      ""
then result is  -9

If mask is      "*f0f0f0f0f0"
then result is  *f0f0f0f0f9

If mask is      "-   ,   .  "
then result is  -         9

If mask is      "-   , 00.0*"
then result is  -*****00.09

If mask is      "-   , 00.00"
then result is  -     00.09

=====

Given a value of:           0

If mask is      "    ,   . $"
then result is           $0

If mask is      "    ,   . *"
then result is  **********0

If mask is      "    ,   . -"
then result is            0

If mask is      "    ,   . 0"
then result is            0

If mask is      "    , 00.0*"
then result is  ******00.00

If mask is      "    , 00.0-"
then result is        00.00

If mask is      ""
then result is  0

If mask is      "*f0f0f0f0f0"
then result is  *f0f0f0f0f0

If mask is      "-   ,   .  "
then result is

If mask is      "-   , 00.0*"
then result is   *****00.00
```

```
If mask is      "-    , 00.00"
then result is        00.00
```

APPENDIX A

TED COMMAND

This section describes the Multics ted command and its associated requests. The command description contains the name of the command, discusses its purpose, and shows the correct usage. Notes and examples are included as necessary for clarity.

Name: ted

The ted command is used to create and edit ASCII segments and is used to perform many kinds of text processing. It can be called as an active function (see "External Functions" in Section 2) and be invoked recursively to a depth of 14.

SYNTAX AS A COMMAND

```
ted {ted_com} {-control_args}
```

ARGUMENTS

ted_com
: reads the contents of the named segment into buffer (exec) and executes it. The ted suffix is supplied if not present. When the contents of buffer (exec) is exhausted, request lines are read from user_input unless the -abort argument (below) is present.

CONTROL ARGUMENTS

See "Notes" below as regards the execution sequence of arguments, and list of undocumented control segments.

-abort
: exit the program if an error occurs while executing a ted_com instead of returning to ted request level. (For historical compatibility, "-com" may be used for this function.)

-arguments args, -ag args
: makes the argument list args available during the processing of a ted_com. The args may be referenced either in buffers (arg1), (arg2), etc., or line 1, line 2, etc. of buffer (args). None of these buffers show up on an x request list until they have been referenced.

-debug, -db
: instructs ted to display "Edit." before accepting a request line from user_input. This control is useful for detecting when a ted_com "falls" back to request level. The -abort argument (above) detects when a ted_com "falls" back to request level, but does not leave you in a position to examine the environment.

-Jset {X...}
: this control argument is obsolete. An old usage of:

```
e ted -Jset X...
```

should become

```
j/s=X.../
```

Refer to Appendix B, sort request, for a description of the preferred usage.

-Jshow
: this control argument is obsolete. An old usage of:

```
e ted -Jshow
```

should become

```
j/?/
```

Refer to Appendix B, sort request, for a description of the preferred usage.

-option STR1{,...STRn}, -opt STR1{,...STRn}
sets the options specified by STRi before anything else is accomplished during ted startup. STR is any option setting which may be given to the "o" request (see Appendix B). Multiple options must be separated by either a comma (,) or a space ( ) character. If space is utilized as the separator, then the whole string must be in quotes.

-pathname path, -pn path
begin ted execution by reading segment path into buffer(0). (See -request control argument for an example of use.)

-pause
interrupts execution of a ted_com (to ted, this simulates encountering a breakpoint at the beginning of the next line in EDIT mode). The next time a request line is fetched, the input routine enters the break sequence (refer to Section 3). This argument is valid only after having interrupted a ted execution. (This argument is mutually exclusive with all other control arguments.)

-request STR, -rq STR
executes STR as a ted request line before entering the request loop. For example:

```
ted -rq gd/something/ ted_com
```

deletes all instances of "something" before executing the ted_com.

```
ted -pn foo -rq "gd/abc/ w" -rq "q"
```

deletes all instances of "abc" in the foo segment, writes it back, and then quits (q).

-reset {level}
breaks out of a ted_com loop and returns to ted request level. The optional "level" is a two-digit number specifying the ted recursion level to be returned to. If level is not specified, then the most recent invocation is assumed. This argument is valid only after having interrupted a ted execution. (This argument is mutually exclusive with all other control arguments.)

-restart
restarts a -safe environment which was left due to a system crash, process interruption, or overt user action (see Appendix B, qhold request). If only one environment exists, it is restarted. If more than one exists, all are listed and the user is asked which invocation to restart.

```
!   ted -restart
    More than one environment exists.
     #     Started, by whom, as what
     1   07/13/82  1706.5 mst Tues  Able.Multics (ted[0])
     2   07/13/82  1706.8 mst Tues  Able.Multics (ted[0])
               comment=weekly report
    Which one do you want?
!   x1
        0 ->     (0)
    Which one do you want?
!   2
    Restarting session of 07/13/82  1707.1 mst Tues.
```

Answers to the above query, "Which one do you want?", may be "?" for info on what you can answer, "N" to restart number N, "xN" to do a pseudox request on N, "dN" to cleanup N, and "q" to exit without restarting. (This argument is mutually exclusive with all other control arguments.)

-safe
places the active ted work segments in the user's default working directory so that the environment is saved in case of a loss such as a system crash. It also displays a warning if there are any existing environments. For example:

```
! ted -safe
ted: 2 environments already saved.
```

(Refer to -restart above.)

-status, -st
generates a list of existing "safe" environments. (This argument is mutually exclusive with all other control arguments.) For example:

```
!   ted -status
     #     Started, by whom, as what
     1   07/13/82  1706.5 mst Tue  Able.Multics (ted[0])
     2   07/13/82  1706.8 mst Tue  Able.Multics (ted[0])
     3   07/13/82  1707.1 mst Tue  Able.Multics (ted[0])
```

The [0] tells the recursion level of the environment.

```
!   ..ted -st
     #     Started, by whom, as what
     1   07/13/82  1706.8 mst Tue  Able.Multics (ted[0])
     2*  07/13/82  1707.1 mst Tue  Able.Multics (ted[1])
    (*=now active)
```

-temp_dir path, -td path
saved environments are normally kept in the home directory. In some cases there may not be enough quota for the work to be done. This argument allows an environment to be placed elsewhere. When this control argument is given, a link is created in the home_dir to the data base for the environment, so that -status and -restart know what exists. The link is removed in cleaning up an environment. The -status listing indicates where any remote environments actually are.

NOTES

Several of the arguments cause a portion of a "script" to be created for ted to follow. The sequence in which arguments contribute to the script, regardless of the sequence used in the command line (top to lowest priority) is:

-option, -pathname, -request, ted_com, -abort, and -debug

For example, if the command is entered as:

ted ted_com -debug -request -option

the sequence of execution by ted is: -option, -request, ted_com, and -debug.

The following control arguments have been deleted from the command description in this issue. They are fully supported as control arguments but will no longer be documented. The intent is to specify options using the -option control argument. (Refer to the "o" request in Appendix B for a complete description of all options.)

OLD USE	NEW USE	DESCRIPTION
-blank	blank	requires blank character between multiple requests on a line.
-no_blank	^blank	opposite to blank.
-break	break	enables tracing of labels.
-no_break	^break	opposite to break.
-trace_edit	edit	display each ted request before execution.
-trace_input	input	display each line of data before insertion.
-label	label	enables tracing of labels.
-no_label	^label	opposite to label.
-read	read	turn read option ON. (Default)
-no_read	^read	turn read option OFF.
-trace	trace	equivalent to (combination of) edit and input.
-no_trace	^trace	equivalent to ^edit ^input.
-part_blank	(na)	this control argument is obsolete.

An example of "old" usage is:

```
ted -blank -label -no_read
```

whereas the "new" usage is:

```
ted -option blank,label,^read
```

TED IN QEDX MODE

Ted may be run in "qedx mode" and is invoked as:

```
ted$qedx
```

-or-

`qedx` (when ted has been initiated with the reference name qedx)

To use ted in "qedx mode" by typing qedx, you must issue the command:

```
initiate [where ted] qedx
```

once in each process before typing the command:

```
qedx
```

The initiate command could be placed in your start_up.ec.

The differences when running in qedx mode are:

1. The suffix ".qedx" is supplied on a ted_com name. In either mode, a suffix of either ".ted" or ".qedx" is accepted. A suffix is added only in the absence of both.
2. The \r sequence includes the NL (and therefore terminates the line being fetched for execution). This means that you cannot use the construct:

   ```
   s/abc/\r/
   ```

3. A .\f in response to a \r input function is treated as data.
4. The "not-pasted" flag is not set on the m request.
5. Certain error messages are prefixed by "qedx" instead of "ted."
6. There is no processing of archive pathnames (names containing "::" in the final entryname) in read request pathnames.
7. If a request requiring a single address is supplied with an address pair, one of the addresses is ignored and a warning is provided telling which address was ignored. Likewise any request provided with any address where none is wanted, also results in a warning message.

All other qedx requests have identical actions as ted requests.

LIST OF TED REQUESTS

The requests flagged below with an "X" in the qedx column identify those ted requests that are compatible with the qedx command requests of the same name. These requests were maintained intact to minimize user confusion when changing over from Qedx to Ted usage. (Refer to "Ted in Qedx Mode" above.) Following is a list of all ted requests. Many of these requests can be invoked in more than one way (e.g., \g{} or \G{}, h or H).

qedx	ted Requests	Description
Regular Requests		
	\?	Where am I?
X	"	comment, ignore rest of line
X	=	print linenumber
	{...}	evaluate
	|xxx	external request call to ted_xxx_
X	a	append
X	b	change buffer
	b()	change current buffer, going to a remembered one
X	c	change (replace) text
X	d	delete
X	e	execute command line
	..	execute command line without input function processing (beginning-of-line only)
	f	fileout into a buffer, auto reversion
X	g	(see "Global Requests" below)
	h	process out pseudo-tabs
	help	online information
X	i	insert
	j	sort
	k	kopy (or copy)
	l	linefeed to user_output
X	m	move
X	n	nothing, sets "." to address
	o	option
X	p	print
X	q	quit with buffer check
	qhold	exit ted while maintaining the safe environment
X	r	read
X	s	substitute
	t	type string to user_output
	u	translate to lowercase
X	v	(see "Global Requests" below)
X	w	write
	wm	write modified buffers (blank mode only)
X	x	status of all buffers
	xm	status, modified buffers only (blank mode only)
	y	tabin, process in HT characters
	zdump	dump octal/ASCII
	z.fi.ad	fill/adjust
	z.fi.na	fill/no-adjust

Alternate Requests

^b	delete buffer
^r	force pathname
!a	bulk append
!b	remember current buffer, then change
!c	bulk change
!e	execute a command line after printing it
!f	fileout to a buffer, no auto reversion
!i	bulk insert
!j	special sort
!k	kopy-append
!l	linefeed to error_output
!m	move-append
!p	print with linenumber
!q	force-quit
!r	read with abbrev expansion of pathname
!s	no-fail substitute
!t	type string to error_output
!u	translate to uppercase
!w	write with abbrev expansion of pathname
!x	status of named or current buffer

Global Requests

X	g=	global linenumber
X	gd	global delete
X	gp	global print
	g!p	global print with linenumber
	g.	print in the manner specified by the null = option
	g*	global-if
X	v=	exclusive global--inverse action to the g requests
X	vd	
X	vp	
	v!p	

Flow-of-control Requests

#	if line
^#	if-not line
%	call buffer with optional arguments
*	if expression found
^*	if expression not-found
:	label define, must be at beginning-of-line
>	transfer of control (goto)
^>	error exit
zif {...}	conditional test
~	return from buffer

APPENDIX B

TED REQUESTS

REQUEST DESCRIPTIONS

The request descriptions include several standard headings (not all of the headings are used for each request):

Request Name (e.g., a (append))

Immediately following the request name is a description of the request.

Syntax

Shows the proper format(s) to use when invoking the request.

Default

Tells what is assumed if no address is given. When no addresses are expected, this heading is not present.

Value of "."

Identifies the position of the current line AFTER the request operation is completed. This value is normally always set to the last line dealt with, therefore it is only included on individual requests that have unusual or unexpected end results. The value of "." is normally included as (".">) within examples.

Examples

Shows correct usage of the request, including buffer contents before and after the execution of the request, and shows lines displayed on the terminal as a result of the request.

Notes

As required.

The ted request descriptions are in the following order: special characters first (i.e., ", #, %, *, etc.), followed by the alphabetized list (a, !a, b, !b, 'b, ^b etc.). Also, a number of abbreviations, metasymbols, and special characters are utilized in the request descriptions (refer to Appendix C when in doubt, or if unfamiliar with specific terms, usage, etc.).

" (comment)

Is generally used to annotate ted_coms and online work. The remainder of the request line following (") is ignored by ted. If an address is included with the request, "." (or current line) is set to that address location.

Syntax

```
ADR" STR
```

Default

If no address is present, "." is not changed.

Example

Buffer contents:

```
"This ted_com enters special headers
"for subroutine descriptions
b (heads)
a
<This is the ted_com>...
```

(if line)

Tests for various conditions of buffer status dependent upon the address configuration (i.e., if the condition is satisfied then execute the rest of request line.)

Since "#" normally means "delete character" to Multics, it must be handled in a special way (i.e., in order to get past Multics it must be entered as "\#"). The examples show what ted expects, not what you type.

The value of "." is unchanged by this request.

Syntax

Format 1	Format 2	Format 3
# <REST>	ADR# <REST>	ADR1,ADR2# <REST>

Format 1 tests for data being in the buffer (i.e., buffer not empty). Format 2 tests for "." being the line addressed. Format 3 tests for "." being within the range addressed.

Example 1

Request:

```
$# Q
```

says, if the current line (".") is at the end-of-buffer, execute Q (force-quit).

Example 2

Request:

```
$*-9,$ # Q
```

says, if "." is within the last-ten-lines of the buffer, execute Q (force-quit).

^# (if-not line)

Tests for the absence of various conditions of buffer status dependent on the address configuration (i.e., if the condition is not satisfied, then execute the rest of the request line).

This request may also be entered as "...'# ...".

The value of "." is unchanged by this request.

Syntax

Format 1	Format 2	Format 3
^# <REST>	ADR^# <REST>	ADR1,ADR2^# <REST>

Format 1 tests for no data being in the buffer (i.e., buffer empty). Format 2 tests for "." not being the line addressed. Format 3 tests for "." being outside the range addressed.

Example 1

Request:

```
$^# Q
```

says, if the current line is anything other than the last-line of the buffer, execute force-quit (Q).

Example 2

Request:

```
$-9,$^# Q
```

says, if the current line is not within the last-ten-lines of the buffer, execute force-quit (Q).

% (call buffer)

Invokes execution of a buffer (similar to the input function \b(x)), optionally passing parameters. These parameters are available as p[N] within <eval>. See Example 3 (below) for more information. The current buffer remains the same buffer. The called buffer becomes the new executing buffer. The ~ request stops execution in the execution buffer and reverts control to the prior level of execution. "Falling out" the end of the buffer also reverts control to the prior level.

Syntax

```
%(X)|p1|p2...
```

where the first character after the buffer name is a delimiter that begins a parameter. The delimiter may be a blank.

Example 1

Request:

```
%(X)|a|b|
```

%

%

provides three parameters, the third being a null string.

Example 2

Buffer contents:

(where b(0) is the current buffer with data and b(a) contains a ted_com to be executed)

Current Buffer	**Buffer (a)**
first	ln
second	:a */e/~
third	?+1;.n >a
	t|didn't find it| 1

Request:

```
o trace    (turn on tracing so the execution
           of the ted_com in b(a) can be observed)
%a         (call the ted_com in b(a) into execution)
```

Terminal output:

```
1  **EDIT**  %a
2  **EDIT**  ln
3  **EDIT**  :a */e/~
4  **EDIT**  ?+1;.n >a
5  **EDIT**  :a */e/~
```

the line numbers above are not part of the terminal output, but are included for explanation purposes as follows:

line 1
Trace of the call (%a) request.

line 2-5
Trace of the execution of b(a).

line 2
Set the current line to the first line of b(a).

line 3
Test for the presence of an "e" on the current line (which is "first" at this time). It is not found, so skip the rest of the request line.

line 4
Test for the presence of a line following the current line of the buffer. If one exists, set the current line to the following line and then loop back to label a.

line 5
Again test for the presence of an "e" on the current line (which is now "second"). This time it succeeds and the return "~" is executed. At this point ted waits for user input.

Request:

```
P          (you want to see the line the test succeeded on,
            including the line number)
```

Terminal output:

```
*EDIT*  P       (trace of previous request)
    2   second  (the result)
```

Request:

s/e/E/ %a (remove the "e" which was matched and recall b(a) again)

Terminal output:

```
 1  **EDIT**  s/e/E/ %a
 2  **EDIT**  ln
 3  **EDIT**  :a */e/ ~
 4  **EDIT**  ?+1;.n >a
 5  **EDIT**  :a */e/ ~
 6  **EDIT**  ?+1;.n >a
 7  **EDIT**  :a */e/ ~
 8  **EDIT**  ?+1;.n >a
 9  **EDIT**  t|didn't find it| l
10  didn't find it
```

line 1
Trace of the substitute and call request.

line 2-7
Trace of the execution of b(a). Same sequence as first call, except this time no match is found. The current line now contains "third" (last line of b(0).

line 8
This time there is no next (following) line, so the rest of the line is ignored (i.e., the "goto label a" is ignored).

line 9
Trace of the failure message contained in b(a).

line 10
Message displayed for the user to indicate failure of the ted_com.

Example 3

Request:

```
%(exec) first of all we can use spaces
```

provides values for evaluation:

```
pn  =7
p[0]="first of all we can use spaces"
p[1]="first"
p[2]="of"
p[3]="all"
p[4]="we"
p[5]="can"
p[6]="use"
p[7]="spaces"
```

Request:

```
%(exec)!nothing would please me more!
```

values:

```
pn  =2
p[0]="!nothing would please me more!"
p[1]="nothing would please me more"
p[2]=""
```

Request:

```
%(exec)
```

values:

```
pn  =0
p[0]=""
```

Request:

```
%(exec)/first/second/third/fourth//sixth
```

values:

```
pn  =6
p[0]="/first/second/third/fourth//sixth"
p[1]="first"
p[2]="second"
p[3]="third"
p[4]="fourth"
p[5]=""
p[6]="sixth"
```

Note

Refer to zif request for additional examples.

* (if expression found)

Searches the addressed text for an <RE>, and if it succeeds, executes the rest of the requests on the same line.

The value of "." is unchanged by this request. The addresses only specify where to look and have nothing to do with the current location.

Syntax

```
ADR1,ADR2*/<RE>/ <REST>
```

Default

If no address is present, "." is assumed.

Note

Refer to the % request for examples.

Searches the addressed lines for an <RE>, and if it fails, executes the rest of the requests on the same line.

This request may also be entered as "...'*...".

The value of "." is unchanged by this request.

Syntax

```
ADR1,ADR2^*/<RE>/ <REST>
```

Default

If no address is present, "." is assumed.

.. (execute)

If any request line begins with "..", it is passed to the Multics command processor for execution without any input function processing.

Syntax

```
..  <command line>
```

: (define label)

Names a location in a ted_com to which control may be transferred. Labels must appear at the beginning-of-line only.

Syntax

Format 1	Format 2
:c	:(STR)

Format 1 defines a single-character label. Format 2 defines a multiple-character label. STR may be from 1 to 14 characters.

Notes

The labels "C", "(C)", and "(C<SP>)" are three different labels representing three distinct buffer locations.

A label search consists of looking through the buffer, starting at the beginning, looking for the string <NL>:(C) for example. There is no attempt to try to determine if a given location in a buffer would be in input mode or edit mode. The user must ensure that no data which is in input mode can mistakenly be found as a label.

> (transfer of control)

Unconditional transfer (or goto) to a label (for looping purposes) or to a relative line for the next request sequence.

Syntax

Format 1	Format 2	Format 3
>C	> (STR)	>+N

Format 4
>-N

Format 1 specifies goto label C (label to be found must have the form ":C" and consist of only one character). Format 2 specifies goto label STR (label to be found must have the form ":(STR)" with STR being from 1-14 characters). Format 3 specifies to go forward from the current line, up to lines (N must be a single digit). Format 4 specifies to go backward from the current line, up to 9 lines. If N is "0", that is, >+0 or >-0, then the current line is re-executed.

Note

Refer to the ":" request for a description of labels.

Example

Assume the current buffer contains a text segment and you are interested in searching the file for instances of "xxx" which you want to change to "something else" each time it is found. For example, the requests:

0n (set pointer to the line ahead of the first line of the buffer. This ensures that the search begins on line 1.)

```
/xxx/ s//\r/ >-0
```

finds the first occurrence of "xxx" and substitutes whatever is entered from user_input (\r request), and then continues the search looking for the next occurrence of xxx.

Another method of accomplishing the same results as above is:

```
:a /xxx/ s//\r/ >a
```

^> (error exit)

Set up to catch a potential error condition. It means do not goto the specified label, just remember it for the rest of the request line. If any error occurs, then goto this label instead of displaying the error message. Normally when an error message is displayed, all ted_com execution is aborted and the next input is from request level. If this request is in effect when an error occurs, the request aborts, but control remains in the current level of ted_com execution (assuming that the label is defined in the ted_com).

This request may also be entered as "'>...".

Syntax

Format 1	Format 2	Format 3
^>C	^> (STR)	^>+N

Format 4

```
^>-N
```

Format 1 specifies goto label C (label to be found must have the form ":C" and consist of only one character). Format 2 specifies goto label STR (label to be found must have the form ":(STR)" with STR being from 1-14 characters). Format 3 specifies to go forward from the current line, up to lines (N must be a single digit). Format 4 specifies to go backward from the current line, up to 9 lines. If N is "0", that is, >+0 or >-0, then the current line is re-executed.

Note

The error message text can be displayed later by entering "{emt();}". (Refer to "Data Elements" in Section 5 for a description of "emt()".)

Example

Request:

```
^>+1 r \b(arg1)
```

executes the following line even if the r (read) request fails. The error message associated with a read failure is also suppressed.

= (print linenumber)

Displays the linenumber of the addressed line.

Syntax

```
ADR=
```

Default

If no address is present, "." is assumed.

Example 1

Buffer contents:

```
a: procedure;
   x=y;
   p=q;
   end a;
```

Request:

```
/q;/=
```

Terminal output:

```
3
```

Example 2

Buffer contents:

Same as above.

Request:

```
$=
```

Terminal output:

```
4
```

\? (where am I?)

Query asking for current status, mode, etc. It is a means of reorienting the user to the current invocation of ted. This feature normally is used by persons working on more than one invocation of ted or if a person thinks he/she is in edit mode but doesn't seem to be getting a response. This request is executed as the only thing on a line.

Syntax

```
\?<NL>
```

Example

Request:

```
\?<NL>
```

Terminal output:

```
ted(2.6) EDIT MODE[01]
```

This output is in this form:

```
ted(v.r)xxx MODE[rc]safe
```

where:

`v.r`
: is the current version/revision of ted.

`xxx`
: can be any of the following: INPUT, EDIT, READ, BULK, or BREAK (refer to "Modes of Operation" in Section 1).

`rc`
: is the current recursion level of ted.

`safe`
: indicates that ted is operating in safe mode; not present if "not safe."

{} (evaluate)

Evaluate request. This does not expect a return value, it simply means to just "do something." If <eval> does return a value, a warning message displays the value.

In contrast, the input function \{<eval>} does expect a return value and uses it as a replacement for the \{<eval>} sequence in the input stream. (Refer to "Input Functions" in Section 1.)

Syntax

```
{<eval>}
```

Note

Refer to Section 4 for more details regarding <eval>.

Example

The multiline request sequence:

```
1  {k:="475446535"}
2  {a:=0; j:=1}
3  :a {a:=a+fs(k,j,2)}
4  zif {(j:=j+2)<fl(k)} >a
5  $a \{a}
6  \f
```

line 1
Sets up a string to work on.

line 2
Initializes a and j.

line 3
Picks up the first two digits of k ("47") and adds them to a. Each return to this line then picks up the next two successive digit pairs to the right.

line 4
Increments the value of j and then tests to see if any more pairs of digits remain (i.e., if j is not less than the length of k, then there is not a pair of digits left).

line 5
Appends the sum (which is 47+54+46+53=200) to the end of the current buffer.

line 6
Terminates the append request. The data is now available for the user.

|xxx (external request ll)

Calls an external request named ted_xxx_. The routine is looked for via the system search rules. The way that the addressed portion of the buffer and <REST> are utilized is determined by each particular routine. In this way, a user may add requests to ted.

Syntax

```
ADR1,ADR2|xxx <REST>
```

<REST> is made available to the external request. It may be utilized as arguments to it. It may be defined to be ignored. It may be defined as other requests which will be executed or not depending on the result of the external request. This action is entirely up to the writer of an external function.

Default

If no address is present, "." is assumed.

However, when the current buffer is empty, then "." is set as undefined. No error message results.

Note

Refer to Appendix E for details on how to code user_written requests.

~ (return)

Stops execution of the running ted_com and reverts to the prior level. This request may be used no matter which way the ted_com was called (i.e., % request or \b input function).

Syntax

```
~
```

Notes

Refer to the % request for additional description and example of its use.

If execution reaches the last character of a ted_com it also reverts to the prior level.

a (append)

Enters input lines from the terminal to create a new segment, or to append input lines after the line addressed by the request. Ted looks for and processes input functions. The request is terminated with a "\f".

Syntax

```
ADRa
TEXT
\f
```

Default

If no address is present, "." is assumed.

Notes

Refer to "Input Mode" (under Modes of Operation in Section 1) for a description of how Ted handles a SP or NL immediately following the "a" request (i.e., on the same line as the request).

The request 0a can be used to append text before line 1 of the buffer.

Example 1

Buffer contents:

```
<buffer empty>
```

Request:

```
a
This can be
a letter,
memo, report,
or user manual.
\f
```

Resulting buffer contents:

```
This can be
a letter,
```

```
      memo, report,
"."-> or user manual.
```

Example 2

Buffer contents:

```
This can be
a letter,
or user manual.
```

Request:

```
2a                                  /letter/a
memo, report,       -or-            memo, report,
\f                                  \f
```

Resulting buffer contents:

```
      This can be
      a letter,
"."-> memo, report,
      or user manual.
```

!a (bulk append)

Accepts lines of data from the terminal to create a new segment, or to append lines of data after the line addressed by the request. Ted does not process input functions. The request is terminated with a "." on a line by itself (beginning-of-line only).

Syntax

```
ADR!a <REST>
LINES
.
```

Default

If no address is present, "." is assumed.

Notes

The request 0!a can be used to append text before line 1 of the buffer.

<REST> which may follow the request does not represent input data. It represents requests which are executed when the input mode is terminated. For example:

```
2!a 1,$p
```

if included in Example 2 below, would display the buffer contents after the termination of the bulk append request.

Example 1

Buffer contents:

```
<buffer empty>
```

Request:

```
!a
This can be
a letter,
memo, report,
or user manual.
.
```

Resulting buffer contents:

```
       This can be
       a letter,
       memo, report,
"."-> or user manual.
```

Example 2

Buffer contents:

```
This can be
a letter,
or user manual.
```

Request:

```
2!a                -or-        /letter/!a
memo, report,                  memo, report,
.                              .
```

Resulting buffer contents:

```
       This can be
       a letter,
"."-> memo, report,
       or user manual.
```

b (change buffer)

Designates an auxiliary buffer as the current buffer. The previously designated current buffer becomes an auxiliary buffer.

Restored to the value of "." when this buffer was last used as the current buffer (i.e., the value of "." is maintained separately for each buffer and saved as part of the buffer status). If a new buffer is created, then "." is undefined. If a window is being selected and the existing value of "." is not within this window, then it becomes undefined.

Syntax

Format 1	Format 2	Format 3
b(X)	b(X,ADR1,ADR2)	b()

Format 1 changes to the named buffer. Format 2 changes to a window within the buffer. This window may be one or more lines. Until the next buffer change, all requests act as though the buffer contained only data within the window (i.e., 1p prints the first line of the window regardless of what line it might be in the whole buffer). Format 3 changes

the current buffer to the last remembered auxiliary buffer (see !b request, up to 10 may be remembered).

Example

Line numbers are shown on the example for the purpose of commentary immediately following example.

```
1   !   b0 1,$P
2           1    first line
3           2    second line
4           3    third line
5   !   b(0,1(7),2) 1,$P
6           2     line
7           3    second line
8   !   !b(0) 1,$P
9           1    first line
10          2    second line
11          3    third line
12  !   b() 1,$P
13          1    first line
14          2    second line
15          3    third line
```

line 1
Shows what is in buffer 0.

line 5
Changes to a window within b0 and shows what is accessible.

line 8
Remembers where we are and changes to b0, then shows that all is available again.

line 12
Changes to the remembered buffer. The window is normally remembered, but was lost due to the fact that the buffer was referenced in its entirety while remembered.

!b (change buffer remembering)

Causes the current buffer to be remembered in a last-in-first-out stack before going to the new one. This remembered buffer may be utilized by means of the b() request.

Restored to the value of "." when this buffer was last used as the current buffer (i.e., the value of "." is maintained separately for each buffer and saved as part of the buffer status). If a new buffer is created, then "." is undefined. Also, "." becomes undefined if a window is selected which does not include the existing value of ".".

Syntax

Format 1	Format 2
!b(X)	!b(X,ADR1,ADR2)

Format 2 addresses a window within the called buffer. This window may be one or more lines. Until the next buffer change, all requests act as though the buffer contained only data within the window. A single part address results in a window consisting of only the line addressed.

Note

The stack can only hold 10 remembered buffers. If this count is exceeded, ted displays an error message. This stack is cleared whenever an error occurs.

^b (delete buffer)

Deletes a buffer (removes from known buffer list).

This request may also be entered as "'b...".

Syntax

```
^b (X)
```

X must not be the current buffer or in the buffer stack.

c (change)

Enters input lines from the terminal and inserts the new text in place of the addressed line or lines. Ted looks for and processes input functions. The request is terminated with a "\f".

Syntax

```
ADR1,ADR2c
TEXT
\f
```

Default

If no address is present, "." is assumed.

Note

Refer to "Input Mode" (under Modes of Operation in Section 1) for a description of how Ted handles a SP or NL immediately following the "c" request (i.e., on the same line as the request).

Example

Buffer contents:

```
This can be                    This can be
a letter,          -or-        a letter,
memo,                          memo
report                         or user manual.
or user manual.
```

Request:

```
3,4c               -or-        3c
memo, report,                  memo, report,
\f                             \f
```

Resulting buffer contents:

```
        This can be
        a letter,
"."-> memo, report,
        or user manual.
```

!c (bulk change)

Accepts lines of data from the terminal and inserts them in place of the addressed line or lines. Input functions are not processed by ted. The request is terminated with a "." on a line by itself (beginning-of-line only).

Syntax

```
ADR1,ADR2!c <REST>
LINES
.
```

Default

If no address is present, "." is assumed.

Note

<REST> which may follow the request does not represent input data. It represents requests which are executed when the input mode is terminated.

Example

Buffer contents:

```
This can be          -or-        This can be
a letter,                        a letter,
memo,                            memo
report                           or user manual.
or user manual.
```

Request:

```
3,4!c                -or-        3!c
memo, report,                    memo, report,
.                                .
```

Resulting buffer contents:

```
        This can be
        a letter,
"."-> memo, report,
        or user manual.
```

d (delete)

Deletes the addressed line or set of lines from the buffer. After completion of this request, all linenumbers within the buffer are automatically reassigned.

Set to the line immediately following the last line deleted. When the last line of a buffer is deleted, "." is set to $+1. Thereafter, negative relative addressing works, but any other kind of reference to "." results in an "undefined" message.

Syntax

```
ADR1,ADR2d
```

Default

If no address is present, "." is assumed.

Example

Buffer contents:

```
a:  procedure;
    x=y;
    q=r;
    s=t;
    end a;
```

Request:

```
3,4d      -or-      /q=/,/s=/d
```

Resulting buffer contents:

```
        a:  procedure;
            x=y;
"."->       end a;
```

e (execute)

Invokes the Multics command processor without exiting from ted. When the execute request is used, the rest of the request line is passed and executed at Multics command level. The execute request can be followed by any valid Multics command.

A similar request is ".. <command line>".

Syntax

```
e <command line>
```

Example

Request:

```
e print report
```

can be used to display the segment in your working directory named report. After the segment is displayed on the terminal, you can continue work in ted as though you had not issued the execute request (i.e., you are still in ted).

Request:

```
e list; print_mail
```

lists the contents of the working directory and displays the contents (if any) of the users mailbox.

!e (print and execute)

Invokes the Multics command processor without exiting from ted. When the execute request is used, the rest of the request line is passed (after first being displayed on the terminal) and then is executed at Multics command level. The execute request can be followed by any valid Multics command.

This request may also be entered as "E <rest>".

Syntax

```
!e <command line>
```

Example

Request:

```
!e print report
```

displays the segment named report in the current working directory.

Request:

```
!e list; print report
```

lists the contents of the current working directory and displays the segment named "report".

f (fileout into buffer)

Causes user_output to be diverted into a buffer (i.e., replaces the contents of the target buffer). At the end of the request line user_output is automatically reverted. The current buffer may be the target of the fileout request as no change is made to the target until the action is complete.

The current address of the current buffer is not changed by this request unless the target buffer is the current buffer; this is only at completion time. The current address of the target buffer becomes undefined when the action is complete.

Syntax

```
f (X) <REST>
```

Example

At times it may be desirable to have output that would normally be sent to the terminal be placed instead in a buffer. One use of this facility would be to have the contents of a segment, in terminal output form, available for user manipulation. Line numbers are shown on the example for purposes of commentary immediately following the example.

```
1   r a.pl1
2   f (0) vd/:/ 1,$P<NL>
```

line 1
Read segment "a.pl1".

line 2
Start fileout (in current buffer (0)), produce a list of all the labels in "a.pl1" by deleting all lines not containing ":", then generate line numbers for all label lines. The <NL> character terminates the fileout request. The contents of b(0) now contains a numbered list of all the lines containing ":" for the user.

!f (fileout into buffer)

Causes user_output to be diverted into a buffer (i.e., replaces the contents of the target buffer). The current buffer may be the target of the fileout request as no change is made to the target until the action is complete.

This request may also be entered as "F...".

The current address of the current buffer is not changed by this request unless the target buffer is the current buffer; this is only at completion time. The current address of the target buffer becomes undefined when the action is complete.

Syntax

Format 1	Format 2
`!f(X)`	`!f<NL>`

Format 1 causes user_output to be diverted to a temporary segment in the process directory. Format 2 causes user_output to be switched back; then the contents of the temporary segment become the contents of b(X). The temporary segment is then deleted. In other words, Format 1 starts it and Format 2 stops it.

Example

To instruct ted to send this output to a buffer, enter:

```
!f(name)
```

Then any request used to enter direct output to the terminal, instead places that output in buffer (name). For example,

```
!f(sizes)           -start fileout
:a                  -label
r .1\b(arg.s)       -read file named on line 1 of b(args)
!x 1,$d             -show path and number of lines, then clear out data
b(args)d            -delete name used
?1 >a               -go back to label a if still more
!f                  -terminate fileout
b(sizes) w sizes    -save results
```

g (global)

The global and exclude (v) requests operate on all lines of a range which do or do not satisfy a given condition.

The value of "." is set to ADR2 of request if an address range is given, or to the last line in the buffer if no address is given.

Syntax

Format 1	Format 2
ADR1,ADR2gX/<RE>/	ADR1,ADR2vX/<RE>/

where X must be one of the following:

d	delete lines
p	print lines
!p P	print lines with the associated line number
.	print in the manner specified by the null= option
=	print the line numbers

The character immediately following the X is taken to be the regular expression delimiter and can be any character not appearing in <RE>. If any of the special characters (^$*.) are used as a delimiter then their corresponding special functions cannot be used in the expression.

Format 1 goes through the addressed range one line at a time and does X for each line which contains a match for <RE>. Format 2 goes through the addressed range and does X for each line which does not contain a match for <RE>.

Note: For all but the delete request, a NL is displayed to signal completion whether or not any lines matched the expression.

Default

If no address is present, "1,$" is assumed. Any byte address given is ignored (i.e., the line begin/line end is used for the range to be processed).

Example 1

Buffer contents:

```
eagle
whale
wolf
baby sea lion
star
```

Request:

```
gd/w/
```

Resulting buffer contents:

```
        eagle
        baby sea lion
"."-> star
```

Example 2

Buffer contents:

```
The text editor,
ted, is a
line or string oriented
text editor.
```

Request:

```
gp/x/
```

Terminal output:

```
The text editor,
text editor.
```

g* (global-if)

Bears a slight resemblance to the global request. It goes through the file a line at a time and decides whether something is to be done or not done to the line. However, in this case the test can be a complex expression and many things can be done when a match is found.

The value of "." is set to the last line processed, unless the g* request terminates by executing the goto request. In this case "." is set to the line which was last examined.

Syntax

Format 1	Format 2
ADR1,ADR2g* <REST>	ADR1,ADR2g*<exp> <REST>
Format 3	**Format 4**
ADR1,ADR2g*= =<NL>	ADR1,ADR2g*// <REST>

Format 1 processes every line. <REST> is one or more requests separated by spaces.

Format 2 <exp> is a (possibly) complex logical expression made up of /<RE>/s and <eval>s ("/" is the required delimiter on the /<RE>/). The following operators may be used in making up the expression:

^	negates the sense of whatever item follows it.
&	does the AND of the item on either side of it.
|	does the OR of the item on either side of it.
( )	groups in order to change precedence.

<SP>s (spaces) cannot surround these operators. The first <SP> encountered terminates the <exp>.

Format 3 uses the same g* as the last one (both selections and requests).

Format 4 uses the same selection criteria as the previous g* but executes a new list of requests.

Allowable requests are:

```
[  ($)] a<SP>string\f
[(1,$)] c<SP>string\f
[  (1)] i<SP>string\f
[(1,$)] d
[(1,$)] m(X)
[(1,$)] !m(X)
[(1,$)] k(X)
[(1,$)] !k(X)
```

```
[(1,$)] p
[(1,$)] P
        l
        !l
        {<eval>}
        >(C)
        >C
        >+n
        >-n
        t/.../
        !t/.../
        =       Prints a 6-digit linenumber, with no NL.
[(1,$)] u/<RE>/
[(1,$)] !u/<RE>/
[(1,$)] s/<RE>/<REPL>/
```

The above request line is like the !s or S request, in that a substitute-failed does not cause any message and does not abort processing of the g*.

Notes

If the g*NL option is set and <REST> contains any p, =, t, or l requests, a NL is displayed as a signal of completion.

The goto causes the g* to be prematurely aborted with "." set to the line which met the match criteria.

The total global-if expanded request line is limited to 512 characters.

The only addressing available is numeric byte address. "(1,3)" and "($-10,$)" are examples. If either address specified does not exist on any particular line, then the request is skipped on that line. (1,10)p is skipped if there are not 10 characters on the line.

The logical expression is processed in an optimal fashion. The /<RE>/ is TRUE if a match is found, otherwise it is FALSE. The returned value of {<eval>} is examined (i.e., "0" or "false" is false, anything else is true).

```
/a/&/b/&/c/   fails immediately if "a" is not found.
/a/|/b/|/c/   matches immediately if "a" is found.
```

This means that "/abc/&{a[10]:=a[10]+1} increments a[10] whenever a line containing an "abc" is found.

Example

Buffer contents:

```
first
second
third
forth->
```

Now use the deferred evaluation function to give a different value each time. Note the functionality of using the value of an assignment as a value for some other purpose.

Request:

```
g* s/^/\g{fak((i:=i+10),"0000")}/
1,$p
```

Terminal output:

```
0010first
0020second
0030third
0040forth->
```

Now use the input function to give the same value each time.

Request:

```
g* s/$/ \{(i:=i+10)}/
1,$p
```

Terminal output:

```
0010first 50
0020second 50
0030third 50
0040forth-> 50
```

Request:

```
g*/^--/ s/those/these/ s|which|that|
```

changes "those" to "these" and "which" to "that" on every line which begins with a "--".

Request:

```
1,200g*/if/|/then/ p
```

prints every line in the first 200 which contain either an "if" or a "then".

Request:

```
{i:=0} g*/:$/&{i:=3;1}|{i} P {i:=i-1}
```

prints, with line number, every line which ends with a ":" and the two lines which follow. The logical expression proceeds like this: if :$ is found, then sets i:= 3 and returns a 1 for successful, otherwise return i, which is successful if ^=0. On success, print/linenumber and decrement i by 1.

Request:

```
g*/SEE/&^/SEE:/ P
```

all occurrences of "SEE" in a file are to be followed by a ":". This request shows all which do not.

h (process out pseudo-tabs)

Modifies the text of the addressed lines by changing all occurrences of the pseudo-tab character, specified by the request, into the appropriate number of spaces. Additionally, text can be left- or right-justified, or centered. This tab-to-space conversion makes the physical length of the segment being edited somewhat longer since one pseudo-tab character is replaced by several consecutive spaces.

A file with horizontal tabs can be difficult to process because two lines which print as though they are the same length might radically differ in terms of actual character length. Performing byte-addressing where horizontal tabs are involved can lead to confusion. After using the h request, there are no horizontal tabs to cause trouble, but backspace characters can still upset things.

This request may also be entered as "...H...".

Syntax

Format 1	Format 2
ADR1,ADR2h/C,n,n.../	ADR1,ADR2h/C/

where C (pseudo-tab) is any character (even ",") which the user wishes to use as a tab character. Spaces replace the pseudo-tab character using tabstops defined by n,n.... Tabstop specifications are of the form:

```
n   set text left
nL  set text left
nC  set text centered
nR  set text right
```

where n represents the column position where the character following the pseudo-tab is to be placed. It must be in the range of 1 through 200.

If tabstops (Format 2) are not specified, ted uses the standard Multics tabstops (11, 21, 31, etc.).

Default

If no address is present, "." is assumed.

Left-justification is assumed, if not specified.

Note

Also refer to the y (tabin) request.

Example 1

Buffer contents (uses "|" character as the pseudo-tab):

```
 6        ....+....1....+....2....+....3....+....4....+....5
 7        |Column 1|Column 2|Column 3|
 8
 9        |a|aa|aaa|
10        |bb|bbb|b|
11        |ccc|c|cc|
12
```

Request (assumes the above data is contained in lines 21-27):

```
6,12h/|,5,20,35/
```

Resulting buffer contents:

```
 6    ....+....1....+....2....+....3....+....4....+....5
 7        Column 1        Column 2        Column 3
 8
 9        a               aa              aaa
```

```
    10          bb              bbb             b
    11          ccc             c               cc
-> 12
```

The first column specified cannot be column 1. If the first column is to begin in column 1, omit the first delimiter.

Example 2

Centered or right-justified tabbing may also be accomplished. In this case, the lines in the file must be terminated with the pseudo-tab character and the beginning character position of the righthand column +1 must be specified as well as a tab position.

Buffer contents:

```
 6          ....+....1....+....2....+....3....+....4....+....5
 7          |Column 1|Column 2|Column 3|
 8
 9          |a|aa|aaa|
10          |bb|bbb|b|
11          |ccc|c|cc|
12
```

Request:

```
6,$h/|,5,20c,35l,50r/
```

The first column (always) begins in position 1.
The second column begins in position 5.
The third column begins in position 20.
The fourth column begins in position 35.
The fifth column begins in position 50.

The "20c" means that in tabbing to column 20 the data being tabbed over is centered.

Resulting buffer contents:

```
    6     ....+....1....+....2....+....3....+....4....+....5
    7           Column 1    Column 2              Column 3
    8
    9              a        aa                         aaa
   10             bb        bbb                          b
   11             ccc       c                           cc
-> 12
```

help (online information)

Gives additional information about the last error which occurred or about some requested subject. If there is no last error, then information about the help request is given. The ted help facility is implemented via the same subroutine used by the Multics system help command. Thus, all question/answer sequences are the same as the user is accustomed to. (Refer to Appendix D for an example of using this facility.)

This request can be invoked at any time during the current invocation of ted (i.e., the user does not need to request the expanded error message immediately after an error condition; it can be asked for at any time, but only as regards the "last error encountered").

The info file (ted_help.info) contains many different subdivisions, known as "infos". This request interfaces with the help_ subroutine so that its interaction is like that of the Multics help command. Most of the infos are not available via the Multics help command. The file however, may be dprinted (see "Notes" below).

Syntax

```
help {args}
```

this request must appear on a line by itself. The args are optional (when no args are present, then further information is given on the last error which occurred) and may be chosen from the following:

`info`
is a request for specific information about "info". It begins by listing all the section titles in the info and then asks if you want to see the first one. (If there is only one section, then the title list is omitted and the section is displayed.) Most infos are made up of several sections. A list of some of the infos can be displayed via:

```
help help list
```

When an info has multiple sections, a table of these is shown first. Then ted asks if you wish to see the first one.

`info sec`
tries to find section "sec" in info "info". It begins by displaying the named section and then asks if you want to see more (if there is other data to be seen). Titles are not automatically given.

If "info" is given as "**", then all infos are displayed which have a section "sec". For example, if you say "help ** overview" it displays all Overview sections of all infos.

`-about topic`
asks to see all sections (of all infos) which contain a given character string. This is not a true word search. This form should be used cautiously since the use of

```
help about b
```

would probably match every info available.

`info -about topic`
finds a section of a given info which contains a given "word."

`-from date`
displays all infos which have been modified since the specified date. It displays the info name, the header line (which shows the date), and the size of the info.

`** change -from date`
displays all the change notices of infos which have been modified since the specified date. It selects the section "mm/dd/yy Change" from all infos which have been updated since "date". It begins displaying with the selected section. If a section by that name does not exist (i.e.,an info is new or has never been changed, then no display occurs.

`|func`
attempts to find info about the external function ted_func_. If it exists, it is in a segment named ted_func_.info. The search consists of looking for ted_func_ via the linker search rules. If it is found, then ted_func_.info is looked for in the

directory where ted_func_ exists. If ted_func_ is not found, then no info is looked for.

|func info
attempts to find info "info" in an external function info file.

Note

To get a dprint of the ted help segment (at Multics command level) type:

```
dprint [wsp info ted.info]
```

CONVENTIONS FOR TED INFO FILES

Many info sections describe one or more constructs. In general, the title of a section gives a summary of the constructs it contains. This summary is enclosed with brackets "[]". Thus a section title like:

Basic [a c i]

indicates that the "a", "c", and "i" requests are described therein.

When a request form is shown, the default address is shown enclosed in brackets "[ ]" as:

[.,.] p

which means that the p request can have two addresses and defaults to the current line if no address is given.

When a new info is added to ted.info it gets the current date. If any changes are made to the technical content of the info, a section is added which is named "mm/dd/yy Change" and is inserted ahead of any prior change indicators.

Many request descriptions begin with a section named "Basic". This gives the ordinary form of the request. Some infos have a section named "Overview" and contain auxiliary information which is not absolutely necessary to know, but which may be of benefit to have read at least once. All request names will lead to an info. Following is a list of other ted infos.

<REPL>	active_function	execute	not
<aexp>	address	expression	options
<assign>	alternate	fileout	prefix
<cat>	arg	global	print
<data>	assign	if	quit
<eval>	break	input	quote
<factor>	buffers	label	read
<lexp>	call	line/string	sort
<lpart>	copy	linenumber	status
<mask>	cut	modes	substitute
<part>	delete	move	trace
<term>	escape	nop	write
<ucat>			

The following is an example of the ted "help" request.

```
!   help b
    11/30/78  Buffers

    Basic [b] (3)              Overview (7)
    Windowing [b(X,a)] (5)     Names (11)
    Push/Pop [!b b()] (9)      Related topics (2)
    Delete ['b ^b] (3)

!   Basic [b] (3 lines).  More help?   yes

    Basic [b]:
    b(X)
       change current buffer to buffer X.

!   Windowing [b(X,a)] (5 lines).  More help?   yes

    Windowing [b(X,a)]:
    b(X,a)
       change current buffer to buffer X, but set a window as specified
       by the addresses.  Until the next buffer change, all requests
       will act as though the buffer contained only data within the
       window.

!   Push/Pop [!b b()] (9 lines).  More help?   skip

!   Delete ['b ^b] (3 lines).  More help?   skip

!   Overview (7 lines).  More help?   yes

    Overview:  Input and editing operations are not performed directly on the
    target segments but in a temporary workspace known as a buffer.  ted supports a
    virtually unlimited number of them.  One buffer at a time is designated as the
    "current buffer".  Most requests operate on the current buffer.

    When ted is entered, a single buffer named 0 is created and designated as
    current.

!   Names (11 lines).  More help?   no
```

At this point the help request is exited and ted awaits further user requests.

i (insert)

Enters input lines from the terminal and inserts the new text immediately before the addressed line. Ted looks for and processes input functions (\b and \r). The request is terminated with a "\f".

Syntax

```
ADRi
TEXT
\f
```

Default

If no address is present, "." is assumed.

Note

Refer to "Input Mode" (under Modes of Operation in Section 2) for a description of how Ted handles a SP or NL immediately following the "i" request (i.e., on the same line as the request).

Example 1

Buffer contents:

```
This can be
memo, report,
or user manual.
```

Request:

```
2i                                  /memo/i
a letter,              -or-         a letter,
\f                                  \f
```

Resulting buffer contents:

```
       This can be
"."-> a letter,
       memo, report,
       or user manual.
```

Example 2

Buffer contents:

```
The text editor,
ted, is a
text editor.
```

Request:

```
3i                                      $i
line or string oriented    -or-         line or string oriented
\f                                      \f
```

Resulting buffer contents:

```
       The text editor,
       ted, is a
"."-> line or string oriented
       text editor.
```

!i (bulk insert)

Accepts lines of data from the terminal and inserts them immediately before the addressed line. Input functions are not processed by ted. The request is terminated with a "." on a line by itself (beginning-of-line only).

Syntax

```
ADR!i <REST>
LINES
.
```

Default

If no address is present, "." is assumed.

Note

<REST> which may follow the request does not represent input data. It represents requests which are executed when the input mode is terminated.

Example

Buffer contents:

```
This can be
memo, report,
or user manual.
```

Request:

```
2!i                    -or-        /memo/!i
a letter,                          a letter,
.                                  .
```

Resulting buffer contents:

```
      This can be
"."-> a letter,
      memo, report,
      or user manual.
```

j (sort)

Sorts addressed text lines on specified keys.

Syntax

Format 1	Format 2
ADR1,ADR2j/key.../	j/s=/

Format 3	Format 4
j/s=X.../	j/?/

Format 1 sorts the addressed lines using the given key specifications.

Format 2 sets the special collating sequence to its default value (i.e., lowercase maps into uppercase). This format replaces the obsolete "-Jset" control argument.

Format 3 sets the special collating sequence using the supplied information. The special sequence is first reset to the default and then is adjusted as specified by X... (see description of X below). This format replaces the obsolete "-Jset X..." control argument.

Format 4 displays the special collating sequence. This format replaces the obsolete "-Jshow" control argument.

where key... is a list of comma-separated keys each in one of these forms:

```
{a|d|=}
{a|d|=}n:m
{a|d|=}n,l
```

and:

<LQ>xxx (where <LQ> is the left single quote (octal 140))
: can precede the first key to specify the record delimiter used (where <LQ>xxx is three octal digits). The DEFAULT value of <LQ>xxx is <LQ>012 (NL character).

m
: represents the character number where the field ends. The key need not end in all records. If the record ends before the key, it is compared as if it were padded with spaces. Beginning and ending keys may be either "X", "$", or "$-X", where "X" represents digits, "$" references the location of the record delimiter, "$-1" the last visible character in the record, etc.

n
: represents the character number where the field begins. The key need not exist in all records. Any missing key sorts as LOW. The key cannot begin before the record.

l
: represents the length of the key, with "-" meaning the rest of record.

a
: represents ascending-order sort (Default).

d
: represents descending-order sort. When multiple keys are present, they can intermix ascending and descending orders. The first key present is the major key. The rest are minor keys of decreasing importance.

=
: eliminates duplicates.

Duplicates (if they exist and are not being eliminated) remain in the original file order.

Duplicates (if they exist and are being eliminated) are eliminated first to last (i.e., the last of the duplicates in the original file is the one retained, all others being discarded).

The "=" form states that duplicates are to be removed. The "=n,l" or "=n:m" forms specify that the duplication count be placed in the record, beginning at character "n" with length "l". This is a zero-filled number of length l. It will contain the value 1 when there is no duplication. If this field is not large enough to hold the count, the count is truncated on the left without comment.

Note: If a record is not long enough to hold the specified count field, the record is extended so it fits.

The following examples show several sample keys:

`j/d/`
sorts the whole record in descending order.

`j/$-5:$-1,1,30/`
given lines which begin with a person's last name, first name, address, ZIP code, sorts in order by ZIP and then the first 30 characters of name.

`j/=11:13/`
sorts with "a1:$" eliminating duplicates, placing the number of occurrences of each record in positions 11, 12, and 13.

Each X in Format 3 represents a mapping pair in one of the forms:

`CC`	map first character to second
`X>YC`	map range of characters to second
`X>YX>Y`	map range of characters to range of characters. Ranges must be of equal size.

where:

`C` is any of three forms:

`'ooo`	`ooo`	is three octal characters
`'C`	`C`	is any character
`C`	`C`	is any character not conflicting with the first two forms

A character mapped to `'777` is meant to be ignored.

Note: The ' and '-> are necessary (i.e., a literal ' is entered as ''; if the string -> is literally required, it is entered as '->).

X->Y means the contiguous characters X through Y, where X must be less than Y in the 9-bit collating sequence. X and Y are any of the C forms.

The default collating sequence (uppercase=lowercase) is as if the following string had been supplied:

```
'000->'177'000->'177'200->'777'777a->zA->Z
```

where:

`'000->'177'000->'177`	maps characters 000 through 177 to themselves
`'200->'777'777`	maps all other to "ignore."
`a->zA->Z`	maps lowercase to be the same as uppercase.

Default

If no address is present, "1:$" is assumed. If no order is specified, ascending is assumed. If no field is specified, the default is "1:$".

In addition, if no keys are given, then "a1:$" is assumed.

Example 1

Buffer contents:

```
S = Select a device
CO= Controller type
FM= Designates the meaning of bits 9-23
SP= Magnetic tape speed
```

```
OP= Type of operation
TD= Magnetic tape density
```

Request:

```
1,$j/1,2/
```

Resulting buffer contents:

```
CO= Controller Type
FM= Designates the meaning of bits 9-23
OP= Type of operation
S = Select a device
SP= Magnetic tape speed
TD= Magnetic tape density
```

The sort could have been in descending sequence by preceding the first number within slashes by the d character (1,$j/d1,2/).

Example 2

Sort all of the lines in a buffer which begin (in character position 1) with dates (in the form mm/dd/yy) into calendar sequence of yy/mm/dd.

Request:

```
j/7,2,1,5/
```

To accomplish the same sort as above, except delete duplicates:

```
j/=,7,2,1,5/
```

To accomplish the same sort, delete duplicates, and count:

```
1,$s/^/00000 / j/17,2,11,5,=1,5/
```

where the first request makes room for the count. Then since ten characters were added to the front of each line, the offsets into the record were adjusted by ten.

Example 3

Buffer contents:

```
1    ll
2    !
3    jules
4    !
5    bakti
6    !
7    amble
8    !
9    zz
```

The buffer is really just a string of data. The first line below represents this, with "@" representing <NL>. The second line represents the way the string is broken into nine records by these <NL>s.

```
ll@!@jules@!@bakti@!@amble@!@zz@
--1-2-----3-4-----5-6-----7-8--9
```

However, when expecting the sort request below, only a portion of the buffer is being used and "!" is the record delimiter. This results in the five records indicated below. Note that the first record is empty, i.e., there is only a delimiter. Note also that the fifth record consists of a single character and has no delimiter.

```
ll@!@jules@!@bakti@!@amble@!@zz@
   1-------2-------3-------4
```

Request:

```
2,8j/'041,1,2/
```

This sorts lines 2 thru 8 in columns 1 and 2 with "!" delimiting the records.

Resulting buffer contents:

```
1     ll
2
3
4     amble
5     !
6     bakti
7     !
8     jules
9     !!zz
```

From our normal perspective this looks very strange, but is exactly what was asked for. Look at the resulting buffer string delimited by the "!"s and using the same record numbers as shown above:

```
ll@@@amble@!@bakti@!@jules@!!zz@
   5-------4-------2-------31
```

But instead, use the following request on the original data:

```
2($),8(1)j/'041,1,2/
```

This will look at the buffer like this:

```
ll@!@jules@!@bakti@!@amble@!@zz@
    -------1-------2-------3
```

This will print in a manner we think is "right".

```
1     ll
2     !
3     amble
4     !
5     bakti
```

```
6    !
7    jules
8    !
9    zz
```

!j (special sort)

Sorts addressed text lines on specified keys with a user-specified collating sequence.

This request may also be entered as "...J...".

Syntax

```
ADR1,ADR2!j/key.../
```

Note

See j (sort) request above for "Syntax" description and "Default" conditions.

Example

The following example utilizes both the j and !j requests to contrast them.

Buffer contents:

```
First
second
first
Secod
last
First
```

Request:

```
j//      (default is "j/a1,-/")
```

Resulting buffer contents:

```
First
First
Secod
first
last
second
```

The file is sorted (ascending order in ASCII sequence), but assume this is not the intended order. The !j sort collating sequence can be modified by calling "ted -Jset" (default collating sequence is like ASCII except uppercase and lowercase are equivalent). Refer to the ted command, -Jset control argument, for additional information.

Request:

```
!j//      (default is "!j/a1,-/")
```

Resulting buffer contents:

```
First
First
```

```
first
last
Secod
second
```

At this point, assume you want to eliminate duplicates.

Request:

```
j/=,1,-/
```

Resulting buffer contents:

```
First
Secod
first
last
second
```

Now, to further eliminate duplicates (i.e., First, first) and retain a count of how many there were of each, use the "=n,l" parameter which says to eliminate duplicates, placing the count in the field beginning at n and which is l characters long. Additionally, sort in ascending (default) order with special collating sequence. At this time it becomes necessary to open up a location in the file where the "duplication" count will be stored. This could be done as:

```
1,$s/^/<SP><SP><SP><SP><SP>/
```

Request:

```
!j/=1,4,6,-/
```

Resulting buffer contents:

```
0002 first
0001 last
0001 Secod
0001 second
```

The duplication count (0002 first) does not reflect the actual count as regards the initial file duplicates (i.e., duplicates were eliminated twice in this example--the first eliminate request did not count eliminated duplicates).

k (kopy)

Copies (duplicates) the addressed test from the current buffer into buffer (X). The copied data overwrites the previous contents of b(X); that is, the previous contents of b(X) are destroyed. If the target buffer contains a modified or not-pasted file (see the m (move) request), ted asks if you wish to overwrite it, unless running in -abort mode or running ted$qedx (refer to Appendix A).

The current buffer cannot be the target of the "kopy".

In the target buffer, the value of "." is undefined.

In the source buffer, the value of "." is last line copied.

Syntax

```
ADR1,ADR2k(X)
```

Default

If no address is present, "." is assumed.

Example

Buffer contents (assuming b1 is the current buffer):

Current Buffer	Buffer 2
a	m
b	n
c	o
d	
e	

Request:

```
2,4k(2)
```

Resulting buffer contents:

Current Buffer	Buffer 2
a	b
b	c
c	d
d	
e	

!k (kopy-append)

Copies (duplicates) the addressed text in buffer(X). The copied data is appended somewhere in buffer(X). Data that existed in buffer(X) before the kopy-append execution is not affected, neither is the data in the buffer copied from. The current buffer may be the target of the kopy-append (i.e., no restrictions exist where the data can be copied).

This request may also be entered as "...K...".

In the target buffer, the value of "." is undefined.

In the source buffer, the value of "." is last line copied.

Syntax

Format 1	Format 2
`ADR1,ADR2!k(X)`	`ADR1,ADR2!k(X,ADR)`

Format 1 causes the data to be placed at the end of buffer (X). Format 2 specifies the line after which the data is to be appended in b(X).

Default

If no address is present, "." is assumed.

Example

Buffer contents (assuming b1 is the current buffer):

Current Buffer	Buffer 2
a	m
b	n
c	o
d	
e	

Request:

```
2,4!k(2)
```

Resulting buffer contents:

Current Buffer	Buffer 2
a	m
b	n
c	o
d	b
e	c
	d

If the above request had been "2,4!k(2,1)" then the resulting buffer contents would be:

Current Buffer	Buffer 2
a	m
b	b
c	c
d	d
e	n
	o

l (linefeed to user_output)

Sends a NL to the user_output switch.

Syntax

```
l
```

Note

Refer to the description of the "%" request (above) for examples that include the "l" request.

!l (linefeed to error_output)

Sends a NL to the error_output switch.

This request may also be entered as "L".

Syntax

```
!l
```

m (move)

Moves the addressed text from the current buffer into buffer (X). The moved data is deleted from the current buffer upon execution. The moved data overwrites the previous contents of b(X); that is, the previous contents of b(X) are destroyed. If the target buffer contains a modified or not-pasted file, ted asks if you wish to overwrite it, unless running in -abort mode or running ted$qedx (refer to Appendix A).

The current buffer cannot be the target of the "move".

The value of "." is set to the line after the last line moved in the source buffer.

The value of "." is set to undefined in the target buffer.

Syntax

```
ADR1,ADR2m(X)
```

where X is the name of the auxiliary buffer to which the lines are to be moved.

Default

If no address is present, "." is assumed.

Note

When data is moved, it no longer exists where it was. The target buffer contains the only copy of the data. To protect this data from being lost, the target is flagged as "not-pasted". This flag is reset when a \bX or r(X) is done. This flag is then tested by the quit request as a reminder that nothing has been done with the data.

Example

Buffer contents:

Current Buffer	**Buffer B**
eagle	a letter
whale	a memo
baby sea lion	
wolf	
star	

Request:

```
3,4m(B)      -or-      /bab/,/wo/m(B)
```

Resulting buffer contents:

Current Buffer	Buffer B
eagle	baby sea lion
whale	wolf
"."-> star	

!m (move-append)

Also called "cut and paste." Moves the addressed data from the current buffer and appends them in buffer (X). Data that existed in buffer (X) before the move-append is not affected. The moved data is deleted from the current buffer upon execution. The current buffer may be the target of a move-append, but the place specified must not be within the range being moved. When data is moved to another buffer, it is marked "not-pasted" until it is utilized via the r or \bx requests, or is deleted. The q request complains if there is unused data left in a buffer.

This request may also be entered as "...M...".

The value of "." is set to the line after the last line moved in the source buffer.

The value of "." is set to undefined in the target buffer.

Syntax

Format 1	Format 2
`ADR1,ADR2!m(X)`	`ADR1,ADR2!m(X,ADR)`

Format 1 appends the data at the end of buffer (X). Format 2 specifies the line after which the data is to be appended in b(X).

Default

If no address is present, "." is assumed.

Note

The user must take into account the fact that each of the move requests change the line numbers of the buffer by deleting same after execution. The request line shown in Example 2 below (three distinct move requests) changes the current buffer line numbers three different times.

Example 1

Assume you are in b(0) and want to move lines 3 through 5 after line 8.

Buffer contents:

```
1    Now is
2    the time
3    to the
4    aid of
5    their country.
6    for all
7    good men
8    to come
```

Request:

```
3,5!m(0,8)
```

Resulting buffer contents:

```
1    Now is
2    the time
3    for all
4    good men
5    to come
6    to the
7    aid of
8    their country.
```

Example 2

Again assume b(0), but this time you want to move more than a single group of lines.

Buffer contents:

Current Buffer	Buffer 1
1 ten	1 moose
2 one	2 duck
3 two	3 deer
4 three	4 robin
5 six	
6 eight	
7 four	
8 five	
9 seven	
10 nine	

Request:

```
2,4m1 4,5!m1 1M1
```

Resulting buffer contents:

Current Buffer	Buffer 1
1 six	1 one
2 eight	2 two
3 seven	3 three
4 nine	4 four
	5 five
	6 ten

n (nothing)

Sets the value of "." to a particular line if an address is present. No other action is taken.

Syntax

```
ADRn
```

Example

Buffer contents:

```
        read
"."-> write
        substitute
        change
        delete
```

Request:

```
/ch/n
```

Resulting buffer contents:

```
        read
        write
        substitute
"."-> change
        delete
```

o (option)

Sets/resets various ted options.

Syntax

Format 1	Format 2
o<NL>	o <STR><NL>

Format 1 displays the current option settings. Format 2 sets/resets options, <STR> is a space- or comma-separated list of option settings.

The display format is:

```
NAME (V.R) [NN] global-options local-options
   {comment=non-empty-comment-string}
```

where:

NAME
: is the name under which ted is being used (could be "ted" or "qedx").

V.R
: is the current version/revision of ted (e.g., "2.5").

NN
: is the recursion number (e.g., "01" for one activation of ted, "02" indicating the second (recursive) activation, etc.).

global-options
: are options which are shared between all invocations of ted (good for the current process only).

local-options
: are options which are effective only within this invocation of ted.

The options may be set/reset by choosing from the following list of keys. When a key begins with ^ it means to turn the option off (opposite meaning to the following description), otherwise it means to turn the option on. (Refer to Appendix A for an alternate method to setting options -- see "-option" control argument.)

Global-options:

`blank, ^blank`
blanks between requests on the same line are required/optional. If blank is specified, a blank character is required between all multiple requests issued in a single input line. The option this sets is global (i.e., it affects any current and future invocation of ted in this process). In general, any number of requests can be issued in a single input line. However, the !e, E, e, !q, Q, q, !r, R, r, .., !w, W, and w requests use up the remainder of the entry line because an NL character is required for termination. Each of these requests must therefore appear on a line by itself or at the end of a line containing multiple requests. The a, c, and i requests are terminated by \f. If the \f appears on the same line as the request, then these requests can be embedded in a multiple request line. For the a, c, !e, E, e, i, !q, Q, q, !r, R, r, !w, W, and w requests, a blank is required immediately after the request character(s), that is, between the request name and the remaining request data (e.g., w<SP>path<NL>, not wpath<SP><NL>). This should not be confused with requests such as substitute which has the valid entry of "s/full/null/<SP>" and not "s<SP>/full/null/". (The suggested mode of operation is "blank".)

`caps, ^caps`
capital letters are permitted/not permitted as requests. (Default is caps.)

`resetread, ^resetread`
executes a resetread on user_input if an error occurs. (Default is resetread.)

Local-options:

`break, ^break`
process \037 characters (as a break) in edit mode/ignore completely (refer to Section 3).

`comment="STR"`
stores STR for display by the "o" request and when invoking ted with the -restart or the -status control arguments. If several environments are being saved, then this request provides the user with a way of adding some ID so that at a later restart time, the user knows which is which at a glance. For example:

```
!     o comment=*`weekly report*'

!     .. ted4 -st
       #     Started, by whom, as what
       1  07/26/82  1420.2 mst  Able.Multics (ted[1]
                comment=weekly report
```

`edit, ^edit`
edit mode lines traced/not traced.

`g*NL, ^g*NL`
when g*NL is enabled, a NL is displayed for the g* request as a signal of completion. (See g* request above for additional information.)

`input, ^input`
input mode lines traced/not traced.

`label, ^label`
labels encountered are traced/not traced.

`null=X`
sets the action executed by the null request. X may be p, !p, or P. (Default is "null=p".)

When no request is given, that is, an address followed by a NL, it is called the null request. It causes the addressed data to be displayed. The null= option selects the specific print format desired (also applies to "g.").

For example:

```
ted -opt null=p
r foo
1,$<NL>
```
results in the complete file being displayed, the same as if the user had entered "1,$ p".
```
o null=!p
1,$<NL>
```
results in the complete file now being displayed with linenumbers, as if the user had entered "1,$!p".

`old-style ^old-style`
when old-style is enabled (Default), buffers containing old-style escape sequences work. When disabled, only the \f, \r, \c, and \b forms are recognized.

`read ^read`
if read is turned OFF (^read) and a read is done in an empty buffer, the file is actually referenced in-place and not copied into the buffer until an attempt is made to modify the buffer. This in no way inhibits the editing which may be done on the segment. When a change is made, the segment is then copied into the buffer. Thus if the segment is only being examined, use of ^read eliminates the overhead of copying the segment. (Default is read.)

`string ^string`
implies string mode (^string is line mode), set/reset by invoking \l and \s requests only; it is displayed here for information purposes only.

`tablit ^tablit`
HT characters in an h request are literal (i.e., whenever HTs are enclosed in a quoted string, do not remove, just account for the space used up).

`trace, ^trace`
equivalent to "edit input"/"^edit ^input". (Refer to Section 3.)

p (print)

Displays the addressed line or set of lines on the terminal. (Also refer to "Note" below.)

Syntax

```
ADR1,ADR2p
```

Default

If no address is present, "." is assumed.

Note

There is a special case of the print request that sets the value of "." to a specific line and prints the line. This usage needs no call-letter or call-character to tell ted what operation to perform; you merely type a valid address (generally a context address although any type is permitted) followed by an NL character (ADR<NL>). In context addressing (/.../), a search is done through the file starting with the first line after the current line to the last line in the file, and then from line number 1 to the current line. For example,

Buffer contents:

```
        aardvark
"."-> emu;
        gnu
        kiwi
        rhea
3276
```

Request:

```
/^k/        -or-     +2     -or-     4
```

Terminal output:

```
kiwi
```

Request:

```
1,2
```

Terminal output:

```
aardvark
emu
```

Example 1

Buffer contents:

```
a:  procedure;
    x=y;
    q=r;
    s=t;
    end a;
```

Request:

```
2,4p    -or-    /x=/,/s=/p
```

Terminal output:

```
x=y;
q=r;
s=t;
```

Example 2

Request:

```
1,$p
```

Terminal output:

```
a: procedure;
   x=y;
   q=r;
   s=t;
   end a;
```

!p (print with linenumber)

```
:procedure/ P

/a:procedure/ P
=
```

Displays the addressed text with linenumbers.

This request may also be entered as "...P".

Syntax

```
ADR1,ADR2!p
```

Default

If no address is present, "." is assumed.

Example

Buffer contents:

```
a:procedure;
        x=y;
        q=r;
        s=t;
        end a;
```

Request:

```
2,4!p -or- /x=/,/s=/P
```

Resulting buffer contents:

```
a:procedure;
        x=y;
        q=r;
```

```
".!-> s=t;
      end a;
```

Terminal output:

```
2     x=y;
3     q=r;
4     s=t;
```

q (quit with buffer check)

Is used to exit from ted after checking for modified buffers. If any buffers are flagged as having been modified (or "not-pasted"), they are listed, and the user is asked if he/she still wishes to quit. The force-quit request (!q or Q) is available which quits ted without the buffer check.

Syntax

```
q
```

Note

The quit request cannot have an address and must be the last request on a line.

!q (force-quit)

Bypasses the buffer check.

This request may also be entered as "Q".

Syntax

```
!q
```

Note

The force-quit request cannot have an address and must be the last request on a line.

qhold (quit-hold)

Exits from ted without destroying the -safe environent. The environment may then be restarted at a later time. This request is not valid if not running in a safe environment.

Syntax

```
qhold
```

r (read)

Puts the contents of an already existing segment into a buffer. This request appends the contents of a specified segment to a buffer after the addressed line.

This request may also be entered as "...!r ...".

Syntax

Format 1	Format 2	Format 3
ADRr path	ADRr	ADRr (X)

Format 4
ADRr (X,ADR1,ADR2)

where path is the pathname of the segment to be read into the buffer. The pathname may be preceded by any number of spaces and must be followed immediately by an NL character. Format 2, with the path omitted, uses the default pathname for the buffer. If the buffer has no default pathname, the message "No pathname given" is displayed and no action is taken. If the default pathname is not trusted, ted asks if you want to use it anyway. Formats 3 and 4 read data from a ted buffer. Format 3 reads the whole buffer while Format 4 reads a selected portion of it.

The path can specify various kinds of files. The forms available to the read request are:

`A::b` which references component b of the archive A (the form A!b may be used for compatibility).

`A` which references segment A.

for example:

```
r help.list     -or-     r bound_help_::help.pl1
```

Any path which begins with the string " [pd] " has these four characters replaced with the name of the process directory.

Default

If no address is present, "$" is assumed.

Notes

This request always uses the rest of the line.

The request "Or path" inserts the contents of a segment before line 1 of the buffer.

The read request sets the default pathname for the buffer to the pathname given in the request, if the buffer was empty before the request. If the buffer is not empty, then the new path is not remembered. Also, if the buffer already has an associated pathname, it is flagged as not trusted (^trust). This ^trust flag is also set when a write is done of less than the whole buffer.

In addition, when a read is being done, a check is made to see if the sum of the present data and the new data exceeds the maximum segment size. If this is true, then the resulting number of pages is displayed and the request is not performed. (Also refer to the w (write) request.) For example,

Buffer contents:

```
<buffer empty>
```

Request:

```
r file1
```

reads the contents of file1 into the buffer; if you edit file1 and then issue a write request:

```
w
```

the edited version is written to the default pathname, file1.

However, a read request issued in a nonempty buffer flags the default pathname for the buffer as not trusted and will not remember the subsequent pathnames. Thus, you can read an unlimited number of segments into the buffer, but when you attempt to issue a write, you must specify a pathname (i.e., those segments are "protected" from being destroyed). For example,

Buffer contents:

```
<buffer empty>
```

Request:

```
r file1
r file2
w
```

Terminal output:

```
Do you want to w with the untrusted pathname >...>file1?
```
(where the user response must be "yes" or "no")

Example 1

Buffer contents:

```
You can input:
text,
programs
```

Request:

```
2r extra    -or-    /text/r extra
```

where extra is a segment containing the following text:

```
(such as
letters, memos,
lengthy reports)
```

Resulting buffer contents:

```
      You can input:
      text,
      (such as
      letters, memos,
"."-> lengthy reports)
      programs
```

Example 2

Buffer contents:

```
<buffer empty>
```

Request:

```
r b.pl1
```

where b.pl1 is the following:

```
b: procedure;
   c=d;
   end b;
```

Resulting buffer contents:

```
b:    procedure;
      c=d;
"."-> end b;
```

Example 3

Several examples of the r(X) request are:

```
r(1)
```

says to read the contents of buffer 1 after the last ($) line (Default).

```
6r(a,2,4)
```

says to read lines 2 through 4 of buffer a after the addressed line (line 6 in this case).

```
r(a,2(1,3))
```

says to read characters 1 through 3 of line 2 of buffer a after the last ($) line (Default).

```
12r(a,2)
```

says to read the entire line 2 of buffer a after the addressed line (line 12 in this case).

!r (abbrev-expand-read)

The same as the r (read) request except that the pathname is abbrev expanded before use.

This request may also be entered as "R".

^r (force pathname)

Does not read a specified file; it merely forces the pathname of the buffer to be "path". This path can then never be removed. Thereafter, r path and w path are not allowed on the buffer.

This request may also be entered as "...'r ...".

Syntax

```
^r path
```

s (substitute)

Modifies the contents of the addressed lines, by replacing all strings that match a given regular expression <RE> with a specified character string <REPL>.

Syntax

```
ADR1,ADR2s/<RE>/<REPL>/
```

The first character after the "s" is taken to be the request delimiter and can be any character other than <SP> and <NL>. It must be the same in all three instances. The delimiter should be one which is not in either <RE> or <REPL> as this would necessitate concealing the internal occurrences. (This greatly increases the probability of making a typing error.)

Some special features are available in <REPL> by using the special sequences "&", "X\=", and "\g{}" in the <REPL> string. The "&" is replaced by the matched string. The "X\=" is replaced by a string of "X"s which is as long as the matched string. The "\g{<eval>}" calls for an evaluation whose returned value is substituted in. The string matching <RE> is available as "Ks" during evaluation. It differs from \{} in that \{} is expanded while the request line is being built, whereas this form is expanded each time the <REPL> is used. (See Example 4 below.)

Notes

Each character string in the addressed line or lines that matches the <RE> is replaced with the character string REPL>. If REPL> contains the special character &, each & is replaced by the string matching the <RE>. The special meaning of & can be suppressed by preceding the & with the \c escape sequence. However, when used in an <RE> the & character has no special meaning. If <RE> ends with a $ character, it means that the NL (i.e., the end-of-line indicator) is used in the search, but is not included in the match. The NL is skipped over before the next search is begun since some expressions could again match the NL. Also, the $ character at the end cannot succeed on the last line of a buffer if there is no NL there. (Refer to "String Mode" in Section 1 for variations.)

The number of matches found during a substitute request is available within <eval>. One way this could be displayed after a substitute is with the request:

```
{"match="||mct();}
```

If the expression has no match when typed from the console, a "Substitute failed" message results. If the expression has no match, and a buffer is being executed, then execution reverts to the previous level with no error message displayed. This condition can be caught however (see the ^> request).

Example 1

Buffer contents:

```
The quick brown sox
```

Request:

```
s/sox/fox/
```

Resulting buffer contents:

```
The quick brown fox
```

Example 2

Buffer contents:

```
        a=b
        c=d
"."-> x=y
```

Request:

```
1,$s/$/;/
```

Resulting buffer contents:

```
        a=b;
        c=d;
"."-> x=y;
```

Example 3

Buffer contents:

```
The text editor,
ted, is a
line or string oriented
text editor.
```

Request:

```
1,$ s/^/?/
```

Resulting buffer contents:

```
?The text editor,
?ted, is a
?line or string oriented
?text editor.
```

Example 4

Buffer contents:

```
1  .
2  .
3  .
4  PROJECT NAME
```

Request:

```
4s/PROJECT/(&)/ P
```

Terminal output:

```
4     (PROJECT) NAME
```

Request:

```
4s/PROJECT/<\=&>\=/ P
```

Terminal output:

```
4     (<<<<<<<PROJECT>>>>>>>) NAME
```

Resulting buffer contents:

```
1  .
2  .
3  .
4  (<<<<<<<PROJECT>>>>>>>) NAME
```

!s (no-fail substitute)

Acts the same as the s request except that a fail condition never occurs. It is used when you do not care if the string appears in the buffer or not. The intent is to do something to any of the strings which might exist.

This request may also be given as "S".

t (type string to user_output)

Sends a specified string of data to the user_output switch. No NL is included.

Syntax

```
t|STR|
```

Note

The character immediately following "t" is used as the delimiter. Any character not in STR can be used. Following this request with the 1 request (t|STR| 1) results in a linefeed also being sent.

See the "%" request "Examples" for correct usage.

!t (type string to error_output)

Sends a specified string of data to the error_output switch. No NL is included.

This request may also be given as "T".

Syntax

```
!t|STR|
```

Note

By following this request with the 1 request (t|STR|+ 1), a linefeed is also sent.

u (translate to lowercase)

Provides the user with the ability to translate each string that matches a given <RE> in the user-specified address range to lowercase. Uppercase and lowercase translation leaves numbers and punctuation unchanged.

Syntax

```
ADR1,ADR2u/<RE>/
```

Default

If no address is present, "." is assumed.

Example

Assume a number of compose controls have been entered into a file in uppercase, but should have been lowercase. Further, assume they are dispersed throughout the file.

Buffer contents:

```
.AL
.BB
 .
 .
 .
.TA
.WRT ...
```

Request:

```
1,$u/^\c..*$/
```

Resulting buffer contents:

```
.al
.bb
 .
 .
 .
.ta
.wrt ...
```

!u (translate to uppercase)

Provides the user with the ability to translate each string that matches a given <RE> in the user-specified address range to uppercase characters.

This request may also be entered as "U".

Syntax

```
ADR1,ADR2!u/<RE>/
```

Default

If no address is present, "." is assumed.

Example

Assume a number of items (each preceded with a number, a period, and a space) exist in a file, some of which utilize multiple lines of data, and the intent is to initial capitalize only the lines preceded by a number.

Buffer contents:

```
1. now is the
   time for all....
2. the quick
```

```
   brown fox....
3. i pledge....
```

Request:

```
1,$!u/^.\c. ./
```

Resulting buffer contents:

```
1. Now is the
   time for all....
2. The quick
   brown fox....
3. I pledge....
```

v (exclude)

A variation of the g (global) request which performs an action when a condition is NOT true. Refer to the g (global) request for additional details.

The value of "." is set to ADR2 of request if an address range is given, or to the last line in the buffer if no address is given.

Syntax

```
ADR1,ADR2vX/<RE>/
```

Default

If no address is present, "1,$" is assumed.

Example 1

Buffer contents:

```
eagle
whale
baby sea lion
wolf
star
```

Request:

```
v=/w/
```

Terminal output:

```
1
3
5
```

Example 2

Buffer contents:

```
The text editor,
ted, is a
line or string oriented
text editor.
```

Request:

```
vp/x/
```

Terminal output:

```
ted, is a
line or string oriented
```

w (write)

Writes the addressed line or set of lines from the buffer into a specified segment. The buffer and current line are unchanged.

Syntax

Format 1	Format 2	Format 3
ADR1,ADR2w path	ADR1,ADR2w	ADR1,ADR2w (X)

Format 4
wm

where path is the pathname of the segment whose contents are to be replaced by the addressed lines in the buffer. If the segment does not already exist, a new segment is created with the specified name. If the segment does already exist, the old contents are replaced by the addressed lines. The old segment contents are destroyed.

The pathname may be preceded by any number of spaces, may not contain any imbedded spaces (e.g., "foo bar"), and must be followed immediately by an NL character. Format 2, with the path omitted, uses the default pathname for the buffer. If the buffer has no default pathname, the message "No pathname given" is displayed and no action is taken. If the default pathname is not trusted, ted asks if you want to use it anyway.

Format 3 writes the data to buffer (X). In this context, X must be enclosed in ()s even if it is a single character. This form is basically the "k" request with a different default address. It is present for symmetry with the "r (X)" form.

Format 4 (not available in qedx mode) writes all modified buffers to their respective pathnames. (This format is only available when in blank mode. That is, if not in blank mode, "wm" would write the current buffer to segment "m".)

Any path which begins with the string " [pd] " has these four characters replaced with the name of the process directory.

Note

The write request must be the last request on a line. If pathname is not given, then the buffer in question must have a trusted pathname. See the discussions on "trust" of pathnames under the "r" request.

Example 1

Buffer contents:

```
A regular expression
searches for a
```

```
certain character
string in the buffer
```

Request:

```
w reg_exp
```

Resulting buffer contents:

The buffer contents are unchanged.

Example 2

Buffer contents:

```
The text editor,
ted, is a
line or string oriented
text editor.
```

Request (assumes that you are in b(0) and the buffer contents are as shown in Example 1):

```
1,2w (X)
```

Resulting buffer contents (b(X)):

```
A regular expression
searches for a
```

!w (abbrev-expand-write)

The same as the w (write) request except that the pathname is abbrev expanded before use.

This request may also be entered as "W".

x (buffer status)

Displays a summary of the status of all known buffers. The name and length (in lines) of each buffer is listed; the current buffer is marked with a right arrow "->" to the left of the buffer name. Finally, each buffer's default pathname, if any, is listed.

Syntax

Format 1	Format 2
x	xm

Format 1 displays information about all the buffers. Format 2 displays a list of all modified or unpasted buffers. (This format is only available when in blank mode).

Example

Four buffers are presently in existence.

Request:

```
x
```

Terminal output:

```
  3 -> mod (0)
555    mod (a) [^trust] >udd>m>jaf>install>ted_info.ec
 15        (farble)    m from b(a)
1816       (info) >udd>m>jaf>install>ted_info.compin
```

Column 1 indicates how many lines are in the buffer. Column 2 has an -> symbol to mark the current buffer. Column 3 indicates mod when the buffer has its modify status on. Note that b(0) has this status even though there is no associated pathname. Column 4 indicates the name of the buffer and the last column contains the associated pathname, if any (it has a prefix of [^trust] when the name is not trusted). After the pathname a "*" character is included if the file is not-read, that is, it is being referenced directly instead of having a copy in a buffer segment.

Note

Refer to the "-ag args" control argument in Appendix A for additional information about this request.

!x (buffer status)

Displays the buffer status of the current buffer or a named buffer.

This request may also be entered as "X".

Syntax

Format 1	Format 2
!x(X)	!x

Format 1 displays the status of buffer (X). Format 2 displays the status of the current buffer. Refer to the "x" request for information about the format, etc. of the display.

y (tabin)

Processes the addressed text replacing all possible blanks with horizontal tab characters. It also removes any trailing whitespace to compact lines. Whitespace can end up on the end of lines as a result of a combination of operations. This whitespace can be troublesome as it does not show up when data is displayed. Canonical form says that there is no trailing whitespace. This then gets rid of any which may exist.

After use of the h request, the file may be 2-3 times as long as it was. The y request goes through the file (the addressed part) and puts in horizontal tabs wherever possible. This minimizes the quota requirements when you write the file out.

This request does not accept specified tabstops, therefore ted uses the standard Multics tabstops (11,21,31 etc.).

Syntax

```
ADR1,ADR2y
```

Default

If no address is present, "." is assumed.

Note

Also refer to the h (process out pseudo-tabs) request.

z.fi.ad (fill/adjust)

Fills the addressed lines, moving text from line to line and adding spaces as necessary in order to right-justify the text.

Syntax

```
ADR1,ADR2z.fi.ad<SP>/STR/N1,N2{,N3}/
```

where:

STR
: All lines beginning with this literal string are untouched. Empty lines are also untouched. If STR is null then all lines are processed.

N1
: Lines are indented to this position. 1 is the first position of the line.

N2
: Lines end up in this position. This is not line length unless N1=1.

N3
: "Initial" lines are indented to this location (optional). "Initial" lines are those which are at the beginning of the range to be processed or any non-empty line which follows an empty line. If N3 is not supplied then N1 is used.

Default

If no address is present, "." is assumed.

Example

Request:

```
1,$z.fi.ad //6,40,1/
```

would result in a block text format of:

```
XXXXXX...XXXXX
     X...XXXXX
     X...XXXXX
     X...XXXXX

XXXXXX...XXXXX
     X...XXXXX
     X...XXXXX

etc.
```

Request:

```
1,$z.fi.ad //1,40,6/
```

would result in a block text format of:

```
     X...XXXXX
XXXXXX...XXXXX
XXXXXX...XXXXX
XXXXXX...XXXXX

etc.
```

Note

Blank lines are not affected (i.e., they remain as blank lines).

z.fi.na (fill/no-adjust)

Fills the addressed lines by moving text from line to line, but does not pad with spaces to right-justify the text (i.e., the right margin is ragged).

Syntax

```
ADR1,ADR2z.fi.na<SP>/STR/N1,N2{,N3}/
```

This has the same format and considerations as z.fi.ad except that no justification is done of the line.

Default

If no address is present, "." is assumed.

zdump (dump octal/ASCII)

A line-oriented dump that includes the octal and ASCII representation of each line referenced in the request invocation. Each NL encountered causes the current line of output to be terminated and the next line begins anew at the left margin. When non-integral-line addresses are given, the first byte within the line number is the number given by the byte address.

Syntax

```
ADR1,ADR2zdump
```

Default

If no address is present, "." is assumed.

Example

Output format:

```
+-----------line number
|       +---byte within line
|       |
V       V  <--------------------  octal  --------------------->  <---- ASCII --->

1       1  146151162163 164011163145 143157156144 040040164150  first.second  th
          17  151162144040 040040040040 146157162164 150012        ird     forth

2       1  011040040040 040040040040 040155157162 145012                 more
```

Some of the "." characters in the ASCII portion of the dump represent unprintable or carriage motion characters (e.g., the NL character (012), horizontal tab character (011), pad character (177), etc.). All of the "."s must be checked against the associated octal representation to verify if the character is a literal "." or a flag for an unprintable or carriage motion character.

zif (conditional test)

Tests the result of the given evaluation. If it is either "0" or "false" then the remainder of the request line is ignored (i.e., any result other than these two values allows the remainder of the request line to be executed).

Syntax

```
ADR1,ADR2zif<SP>{<eval>} <REST>
```

Default

There is no default address. Certain built-ins within <eval> can reference buffer data. These built-ins are not defined if no address is given.

Example

The following example shows differences between \b (input function) and the % request and shows how these differences interact with conditional execution. These differences apply to any of the conditional requests. Assume b(0) is the current buffer and that the buffer contents are:

b(0)	b(abc)
.	t!first! 1
.	t!second! 1
.	{"pn=": pn;}

Note

Refer to Section 5 for more details regarding <eval>.

Then execute these requests:

```
!   o trace

!   %(abc)

    *EDIT*   (abc)
    **EDIT**  t!first! 1
    first
    **EDIT**  t!second! 1
    second
    **EDIT**  {"pn=": pn;}
     pn=0
!   %(abc) a b c d
    *EDIT*   (abc) a b c d
    **EDIT**  t!first! 1
    first
    **EDIT**  t!second! 1
    second
    **EDIT**  {"pn=": pn;}
    pn=4
!   \b(abc)

    **EDIT**  t!first! 1
    first
    **EDIT**  t!second! 1
    second
    **EDIT**  {"pn=": pn;}
    pn=0
    **EDIT**
!   zif {1=2} \b(abc)

    **EDIT**  zif {1=2} t!first! 1




    **EDIT**  t!second! 1
    second
    **EDIT**  {"pn=": pn: }
    pn=0
    **EDIT**
!   zif {1=2}  (abc)

    *EDIT*  zif {1=2}  (abc)
```

-Turn on tracing so the execution of b(abc) can be observed.

-Call b(abc) into execution without any arguments.

-The trace of the execution.

-Call b(abc) into execution with arguments.

-Input function, one line at a time from b(abc).

-This is the NL after \b(abc).

-Attempt to conditionally execute b(abc) by means of the \b form.

-zif fails, <REST> is ignored. Note that <REST> is just the rest of the first line, not the rest of the buffer.

-Another attempt to conditionally execute, this time with the % request.

-zif fails, % is ignored. This means that the buffer is not called at all. This correctly does conditional buffer execution.

APPENDIX C

ABBREVIATIONS, METASYMBOLS, AND SPECIAL CHARACTERS

Abbreviations and metasymbols within syntax indicate the user is required to provide a substitute "string" in their place (i.e., they are not literal occurrences of the characters). Special characters on the other hand are literal occurrences and must be entered exactly.

ABBREVIATIONS

LINES
: Lines of input characters (with no input functions processed) terminated by a line consisting only of a "." followed by an NL character (related to bulk input requests). (Also see "TEXT").

N
: A single digit or character.

NL
: Newline character (i.e., the terminal key NL pressed by the user to move the display mechanism to the beginning of a new line).

PATH
: Pathname of a storage system entry; unless otherwise indicated, it can be either a relative or an absolute pathname.

STR
: More than one digit or character and implies "string".

TEXT
: Single or multiple lines of text or data (with input functions processed) terminated by "\f" (related to basic input requests). (Also see "LINES").

X
: All buffer names in this document are referred to as "X" (i.e., X implies either a 1-character or up to 16-character name).

METASYMBOLS

...
: A continuation of the line (i.e., it may consist of any combination of data, text, requests, or nothing).

.
.
.
: A continuation of lines (i.e., it may consist of strings or lines of text, data, requests, or nothing).

<GA>
: The user must provide the grave accent (octal 140) character in the location flagged.

<NL>
The user must provide an NL (newline) character in the location flagged (i.e., the user must press the NL key on the terminal to provide a terminating sequence).

<RE>
Any regular expression.

<REPL>
A character string used to replace an <RE>.

<request>
Single or multiple sequence of requests (also see "<REST>").

<REST>
Represents everything out to the end of the line. The meaning of <REST> varies depending on circumstances.

<SP>
The user must provide a space in the location flagged.

ADR
A single address.

ADR1
First half of expected address pair.

ADR2
Second half of expected address pair.

SPECIAL CHARACTERS OR LITERALS

"
Delimits quotes within an <eval> (quotes within a quoted string must be doubled, for example, "a""bcd""e").

#
Error correction character (used to delete previous entered character or characters). This is a Multics default erase character.

$
In a line address, the last-line of the buffer.

In a byte address, the NL, or where the NL would be if there is none.

In an <RE>, match the end-of-line.

&
Special character in the <REPL> string of a substitute request, meaning "give me the string which was matched ".

Logical AND operator in the g* request.

'
Special character (apostrophe) meaning "not".

()
Used to bracket byte addresses, multicharacter buffer names, multicharacter labels, arithmetic grouping within <eval>, and logical grouping in g*.

*

Operator in an <RE> (matches 0-or-more occurrences of the character which precedes it).

Multiply operator within an <eval>.

+

As a prefix on a number in line or byte addresses, represents forward relative addressing.

Addition operator within an <eval>.

,

As an address separator (comma), the following (or right half of the) address is to be "searched for" from the same starting point as the previous (or left half of the) address (also see ";").

-

As a prefix on a number (hyphen) in line or byte addresses, represents backward relative addressing.

Subtraction operator within an <eval>.

.

The current-line in a line address or the current-byte in a byte address.

Terminates bulk input requests when entered on a line by itself.

Matches any character in an <RE>.

/

Normally used within a line or byte address to delimit a search expression.

Division operator within an <eval>.

:

Separator within an <eval>.

:=

Assignment operator within an <eval> (not to be confused with the = (equality) operator).

;

As an address separator, the following (or right half of the) address is to be "searched for" from the location found by the previous (or left half of the) address (also see ",").

Used as a separator within an <eval>.

<

Less-than operator within an <eval>.

=

Equality operator within an <eval>.

>

Greater-than operator within an <eval>.

@

Error correction character (used to delete the entire contents of the current line being built). This is a Multics default erase character.

Absolute buffer referencing character. Must be entered as \@.

[]

Brackets subscripts within <eval>.

Surrounds limit address for address expression searching.

^

Special character (circumflex) meaning "not".

{...}

A literal delimiter for evaluations {<eval>}.

|

Logical OR operator within an <eval> or the g* request.

||

Concatenate operator within an <eval>.

APPENDIX D

ERROR MESSAGES

This section contains an alphabetically sorted list of ted error messages that may be displayed on the terminal during a ted session. The actual ted message is shown on the first line, the second line further describes the error condition and is available to the user if "help" is invoked after an error message display. This expanded message can be invoked at any time during the invocation of ted (and once called, the message extension is related to the last error condition which occurred). The extended or second message (invoked by "help") is available as shown below with the exception that line length of messages on the terminal probably will not be folded as shown here. A sample ted session describing an error situation immediately follows the error messages listing.

Most messages are followed by information which specifies the string in error. Usually, the string begins at the last place known to be valid and ends with the character in error.

`.*\[...] not handled.`
This form of an <RE> search is not supported.

`// undefined.`
The search expression // means to use the last one specified. There is none.

`? can only appear first.`
The "?" character signals an address prefix. An address prefix can only be present before the first address.

`Abbrev result >512.`
The result of trying to abbrev expand the pathname exceeded 512 characters.

`Addr- after buffer.`
The given address attempts to reference beyond the end of the buffer.

`Addr- after line.`
The given address attempts to reference beyond the end of the line.

`Addr- before buffer.`
The given address attempts to reference before the beginning of the buffer.

`Addr- before line.`
The given address attempts to reference before the beginning of the line.

`Addr- wrap-around.`
The location of the second address is before that of the first.

`Backup search failed.`
The backward search reached the beginning of the buffer without finding a match for the given <RE>.

`b(xxx) not found.`
The named buffer is being used in a circumstance which requires that it exist, but it does not.

`Bad ? form.`
An address prefix cannot have the form given.

`Bad char in byte addr.`
A character was encountered which is not valid within a byte address.

`Bad decimal digit.`
Only "0123456789" are allowed.

`Bad hex digit.`
Only "0123456789abcdefABCDEF" are allowed.

`Bad move spec.`
A string cannot be moved in the current buffer to a place within that string.

`Bad octal digit.`
Only "01234567" are allowed.

`Buffer empty.`
No address may be given when the buffer is empty.
-or-
When a buffer is empty, the only requests allowed which use addresses are the "r" and "a" requests.

`Can't delete current buffer.`
The current buffer may not be deleted.

`Can't m/k to current buffer.`
The m/k requests delete the contents of the target buffer. This does not work if the source is the target, because the data to be used will be lost.

`Can't process multisegment file.`
MSFs must be handled one component at a time.

`Can't trust saved pathname.`
Due to the sequence of events which have occurred, ted cannot be sure that the pathname saved truly reflects the status of the buffer. The user must be explicit in intent.

`Can't write to an archive.`
This operation is not supported by ted.

`Char search failed.`
The <RE> specified in a byte address could not be found.

`Entry not found.`
The *|function requested was not found using the current system search rules.

`External function error.`
The last error was in an external function. The implementer of this function has made no additional information available about the error.

`External symbol not found.`
The *|function requested was found, but did not contain the necessary entry point. This condition can result from adding a name to an object segment which is not defined within the object.

`Incorrect buffer name.`
A buffer cannot be named "," or ")".
-or-
You are running under a limited system which does not allow the buffer request to be used. Therefore b(0) is the only buffer available. You must use the form M(0,address) or K(0,address).

`Invalid addr char.`

`Invalid option.`
Only the options shown when o<NL> is typed may be specified.

`Invalid request.`
A request was expected at this point, but what was present is not a valid request.

`Label > 16 chars.`
A multicharacter label is limited to 16 characters including the parentheses. The closing ")" must be found within 16 characters of the opening "(". A buffer name specification which begins with a "(" must have the matching ")" present.

`Label Y not defined in b(X).`
The specified label is not present in the buffer mentioned. Labels must always begin lines (otherwise they are not labels).

`Level > 500.`
Buffer nesting cannot go beyond a depth of 500.

`Line search failed.`
The <RE> specified in a line address could not be found.

`Linkage section not found.`
The *|function requested was found, but it is not an executable segment. This condition can result from inadvertent over-writing of the object segment.

`Misplaced.`
A character (which appeared in the message) was in an improper place among g* subrequests.

`Missing quote.`
A quoted string was begun but never terminated.

`Missing right parenthesis.`
There is no closing ")" to match an existing open "(".

`Name > 16 char.`
Buffer names are restricted to 16 characters.

`No blank after Z`
Global-if subrequests are always in blank mode. One or more blanks must separate subrequests on the line.

-or-

Ted is running in blank or part-blank mode. One or more blanks must separate requests when more than one request occurs on a line. However, since "r", "w", "e", and "o" requests use up the rest of the line, then the required blank is between the request and the rest of the line. The requests which are under the scope of part-blank are: a, c, i, d, r, e, w.

`No ).`
A ">(" was given in a global-if but the closing ")" was left out. Only absolute addresses are available for global-if subrequests (i.e., n or $n).

`No 1st addr.`
Some address part must precede the "," or ";" character.

`No 1st delimiter.`
Neither blank nor NL may be a delimiter.

`No 2nd delimiter.`
An <RE> was started, but no terminal delimiter was present. The character which began the expression serves as a delimiter and thus must appear a second time.

`No 3rd delimiter.`
A substitute request must have a third delimiter which specifies the end of the replace string.

`No NL.`
This condition comes about when the buffer request line is exceeded.

`No \f.`
A g* a/c/i request does not have the EOI mark (\f).

`No buffer remembered.`
The current-buffer stack is empty.

`No char for \=.`
The "equals convention" of the substitute request must have a character before it which is the character to be replicated.

`No pathname given.`
An r or w request can be given without a pathname only when there is a pathname remembered for the current buffer.

`No }.`
Either a "{" or "\g{" was specified but the matching "}" was not present.

`No routine name supplied.`
The call external support routine request needs to have a name immediately after the "|". The name must be terminated with a "<SP>" if any arguments are to follow or if other requests follow on the same line. Otherwise it must be followed by an NL.

`Not allowed on this buffer.`
The status of this buffer is such that the operation attempted is not allowed.

`Not-defined-`
Buffer related built-ins are not defined in \{...} context, or when no address is given for {...} or zif {...}.

`Null buffer name.`
Buffer names may not be zero-length.

`One address allowed.`
The given global-if subrequest can only have a single address.

`Only 2 addr allowed.`
Only two addresses may be present. However, there can be any number of address prefixes.

`Out_of_bounds occurred. Request aborted.`
The operation being performed attempted to form a file which is larger than the maximum segment size of the system.

`Pathname ignored, buffer was forced to`
The "^r" request has been issued in this buffer. The name given at that time is the only name which is allowed for this buffer. Any given by the user are ignored.

`Remembered >10 buffers.`
Only 10 buffers can be remembered in the current-buffer stack.

`Search addr not allowed-`
Only absolute addressing is available in global-if subrequests.

`Subfile name too long.`
The length cannot exceed 32 characters.

`Substitute failed.`
The specified <RE> was not found in the area addressed.

`System error code.`
This error code is not normally expected. No further information is available.

`Unknown X* request x.`
The apparent subrequest encountered is not one of the valid ones for use with global-if.

`Zero-length segment.`
The segment read contains no data. This is a warning that the pathname has been associated with the buffer.

`\r read \f.`
While trying to fill a "\r" function from request level, a "\f" was received.

`curline undefined.`
The value of "." (current line) for a buffer becomes undefined under the following circumstances: 1) data is moved or copied into it, 2) a fileout is executed into it, 3) the last lines of a buffer are deleted, or 4) a window is defined which does not contain the current location.

`q rejected.`
Whenever a buffer with an associated filename is modified, a flag is set. It is reset when the whole buffer is written. Whenever data is moved to a buffer, a flag is set on the target buffer. It is reset when a read is executed from that buffer, or the whole buffer is invoked (\b(X)). The q request checks these flags. If any are set, warnings are printed and the user is asked if the quit is really intended. If the reply to this query is "no", then the q is rejected. Also, if the q request has any address, or is not followed by an NL, it is also considered an error.

`substr from outside string`
An fs(...) request tried to begin either at location 0 or at a location greater than the length of the string extracted from.

EXAMPLE

The following example describes but two of many possibilities resulting in ted error conditions. The standard error message is shown first (prefaced with a message identifier, in this case"Abe") followed by a more detailed message resulting from the user invoked "help" request.

```
!   ted
!   d
    Buffer empty
!   help
    Abe) Buffer empty

    When a buffer is empty, the only requests allowed which use addresses are
    "r" and "a".
!   /abc/
    Buffer empty.
!   help
    Abe) Buffer empty.

    No address may be given when the buffer is empty.
```

APPENDIX E

USING AND WRITING EXTERNAL REQUESTS

Ted can be used to invoke external requests. These requests can be standard system routines or user-supplied. This facility allows a user to add specialized editing requests to ted.

Following is a summary of standard support functions:

`[.] |ax`
append LINES after addressed line (after Speedtype expansion).

`[.,.] |cx <REST>`
change addressed lines, replacing with LINES (after Speedtype expansion).

`[.] |ix <REST>`
insert LINES before addressed line (after Speedtype expansion).

Note: LINES are read from user_input, with Speedtype expansion being done as each line is received. The end-of-input signal is "\f". <REST>, which may follow the request, does not represent input data, but requests which are executed when the input mode is terminated.

`[.,.] |comment`
add comments to addressed lines. Each nonblank line is displayed without a NL. Then, whatever is typed is added to the end of the line. However, the last two characters can be one of the following control sequences:

\d	delete the displayed line
\F \f	end of input
\i	insert next line typed, then examine for "\a"
\a	append next line typed, then examine for another "\a"

`[.,.] |tabin`
convert spaces to HTs where possible and remove trailing white space from all lines.

`[.,.] |tabout /C,n,n.../`
convert pseudo-tab C to spaces using tabstops defined by n,n,... Tabstop specifications are in the form:

n	set text left
nL	set text left
nC	set text centered
nR	set text right

where n represents the column where the character following the tab character is to be placed. It must be in the range 1 through 200.

When the left or center options are selected, they apply to the text leading up to the tabstop. The location of each tabstop in turn is remembered. Then when a left/center is called for, the data since the last tabstop is involved. The number of spaces needed is calculated. If centering, half of this number is placed before the data and the rest after; otherwise, all the spaces are placed before the data.

`[.,.] |gtabout /RE/C,n,n.../`
(global | tabout) convert pseudo-tab C to spaces on all lines which match the expression RE.

`[.,.] |vtabout /RE/C,n,n.../`
(exclusive | tabout) convert pseudo-tab C to spaces on all lines which do not match the expression RE.

`[.,.] |fiad /STR/L,R{,I}/`
fill the addressed data and adjust to make an even right margin.

`[.,.] |fina /STR/L,R{I}/`
fill the addressed data without adjusting.

In respect to the two fill functions above:

STR is a literal string for line exclusion.
L is the left (beginning) character position.
R is the right (ending) character position.
I is the indention character position. The indention position is assumed to be the same as the left if not given. Indentation applies to each "first" line. The first line of data addressed is a "first" as well as any one which follows an empty line. All empty lines within the addressed range remain as such. If STR is non-null, any line which begins with this string is left intact and filling begins at the left again on the next line. The line following is not a "first" one.

`[.,.] |dumpl`
dump (long) addressed string in octal and ASCII 20 across (needs 110 print positions).

`[.,.] |dumps`
dump (short) addressed string in octal and ASCII 10 across (needs 65 print positions).

`[.,.] |dumpvs`
dump (very-short) addressed string in octal and ASCII 5 across (needs 39 print positions).

`[.,.] |dump`
dump addressed string in octal and ASCII using long, short, or very-short, depending on current linelength.

USER-WRITTEN REQUESTS

A ted external request is a PL/I program conforming to certain design criteria. The procedure accepts three arguments: a pointer to the ted_support data structure, a char (168) varying string which may be used to return information about an error condition, and a return code.

A RESOLVE ted external request must have a segname and entryname of ted_RESOLVE_$ted_RESOLVE_. It is invoked by entering:

`[.,.] |RESOLVE <REST>`

HOW AN EXTERNAL REQUEST AND TED WORK TOGETHER

It is assumed that most external requests work in the same general fashion. The ted command, therefore, tries to remove as much burden as possible from the writer by doing all the work that is common to this model. Processing is done from an input string to an output segment.

The whole buffer content is available for reference and the input string is the addressed portion of that data. A request may do whatever it is called upon to do to the output segment, but it can not modify the input data in any way, and may not count on the string being in the same location if called again in the same buffer. The actions performed are:

1. Copy all of the input string which precedes the addressed range into the output segment.
2. Call the routine. <REST> (any remaining characters on the request line) is passed along for whatever the routine wishes to use it for. The call is:

   ```
   dcl  ted_xyz_ entry (ptr, char (168)var, fixed bin (35));
   call ted_xyz_ (addr (ted_support), msg, code);
   ```

3. Replace the input data up to the end-of-address location with the contents of the output segment. The end-of-address location may be updated by the request to end-of-string if it has modified all of the input data.
4. Set the current location, if specified.

The return code may specify that Step 3 is to be omitted, that Steps 3 and 4 are to be omitted (no change), or that an error occurred. <REST> is passed to the request with the assumption that everything which follows is information for the request. The request may then:

1. Do nothing. A new line is read for execution.
2. Set the next location after any data utilized by the request. In this case, execution continues on the rest of the line.
3. Supply a new value and length for the data in the request line and set the next location back to 1. In this case, execution begins with the line supplied by the request.

GLOBAL EXTERNAL REQUESTS

The syntax of a global request is:

```
|function /expression/additional-info/ <REST>
```

or

```
|function /expression/ <REST>
```

A global processing mechanism is available whereby the writer of a request may accomplish global types of action (g or v) without having to know all the necessary details. The ted_support structure contains a pair of entry variables for this purpose.

The first procedure below is invoked when a global expression needs to be processed. A globally executing function could be (but is usually not) written to always use the remembered expression; in that case this procedure would not be called.

```
call proc_expr (ted_support_p, msg, code);
```

The arguments to proc_expr are exactly the ones which the request received. proc_expr takes the first nonblank character as the delimiter, scans and compiles the expression, and leaves the second delimiter as the current character. If ted responds with a zero return code, everything is ready to process any additional information which the function may require. If there are no arguments for the request and it allows other requests to follow in the request line, the current location must be advanced over this second delimiter.

The second procedure accomplishes all the global overhead.

```
call do_global (worker, mode, ted_support_p, msg, code);
```

where:

`worker`
is an internal procedure, having no arguments, that actually performs the function. It is called once for each line in the addressed range which matches the criteria. When called, "inp.sb" points to the first character of the line to be processed and "inp.se" points to the NL of this line. These two values may not be modified.

`mode`
is either a "g" or "v" to indicate which kind of operation is needed.

The remaining three arguments are the same ones with which the function itself was called.

REGULAR EXPRESSION USE

Information is also available in the ted_support include file which allows a request writer to make use of ted's regular expression searching facility. Using this facility is a 3 step process.

Step 1. Initialization of an Expression Area

This step is usually not needed. It exists so that a function can make use of regular expressions without impacting the remembered expression in ted.

```
call tedsrch_$init_exp (addr (someplace), size (someplace));
```

The first argument identifies the hold area location, and the second argument describes its length in words.

Step 2. Compilation of an Expression

This step may also be optional. If a null expression (i.e., "//") is given to a request, it means "use the remembered expression." In this case the step is skipped.

Here a regular expression is compiled into its internal form. Usually, an expression is compiled once and then used for searching many times.

```
call tedsrch_$compile (ex_p, ex_l, re_p, linemode, REmode,
   msg, code);
```

where:

`ex_p`
points to first character of the expression to be compiled.

`ex_l`
is the number of characters in the expression.

`re_p`
points to the area where the compiled expression is held (the ted default area may be utilized by referencing "ted_support.reg_exp_p").

`linemode`
is "1"b for line mode or ""b for string mode.

REmode
is "1"b for regular expression mode or ""b for literal mode.

The remaining two arguments are the ones which were passed to the procedure.

Step 3. Searching For a Match

This step is not likely to be optional. It is the one which searches an area of a buffer looking for a string which matches the expression.

Searching may be done in the input string, but not the output segment since it is not a buffer.

```
call tedsrch_$search (re_p, cb_p, string_b, string_e, match_b,
   match_e, match_e2, msg, code);
```

where:

re_p
points to the area containing the compiled expression.

cb_p
points to the control block associated with the input data.

string_b
is the offset in the buffer string where the search is to begin.

string_e
is the offset where the search is to end.

match_b
is the offset where a match begins.

match_e
is the offset where a match ends.

match_e2
is the last character used to find the match (sometimes higher than "match_e").

The remaining two arguments are the function parameters.

REQUEST INFO

An external request may also have info available for the ted help request.

The info is in standard help file format. The help file must be named ted_RESOLVE_.info. The ted help request looks for ted_RESOLVE_.info in the directory where ted_RESOLVE_ is found (via system search rules). The info is requested in ted with one of these forms (see help):

```
help |RESOLVE
help |RESOLVE section
help |RESOLVE -about topic
```

The last two are usually not of any use because the info is usually not complicated enough to have more than one section.

Following is an example of a request to convert vowels to uppercase, both regular and globally.

```
ted_uppercase_: proc (ted_support_p, msg, code);
/* This routine converts all vowels to uppercase in the range addressed.     */
/* It also can do this action globally (g | v).                              */
/* It does not allow any request to follow in the same line.                 */
/* Usage: {.,.} |uppercase {ignored}                                         */
    mode = " ";
    goto common;

ted_guppercase_: entry (ted_support_p, msg, code);
/* Usage: {.,.} |guppercase /regexp/ {ignored}                               */
    mode = "g";
    goto common;

ted_vuppercase_: entry (ted_support_p, msg, code);
/* Usage: {.,.} |vuppercase /regexp/ {ignored}                               */
    mode = "v";

common:
    if (ted_support_version_2 ^= ted_support.version)
    then do;                                /* check for proper version      */
       code = error_table_$unimplemented_version;
       return;                              /* can't handle this one.        */
    end;
    if (inp.de = 0)
    then do;                                /* must be some data to work on  */
       msg = "Buffer Empty.";               /* supply the message text       */
       code = tederror_table_$Error_Msg;/* say that a message is present     */
    end;
    else do;
       if (mode = " ")
       then call worker;
       else do;
          call proc_expr (ted_support_p, msg, code);
          if (code ^= 0)
          then return;
          call do_global (worker, mode, ted_support_p, msg, code);
       end;
       current = out.de;                    /* and say that "." is there     */
       code = tederror_table_$Copy_Set; /* tell ted to finish up             */
    end;

worker: proc;
       i = inp.se - inp.sb + 1;             /* calc how much to process      */
       substr (ostr, out.de+1, i)
          = translate (substr (istr, inp.sb, i),
          "AEIOU", "aeiou");                /* translate that much           */
       out.de = out.de + i;                 /* update the output length      */
    end worker;
```

```
dcl (msg              char (168)var,
     code             fixed bin (35)) parm;
dcl mode              char (1);
dcl  i                fixed bin (24);
%include ted_support;
 end ted_uppercase_;
```

The next example is a request to renumber a range of a buffer.

```
ted_renumber_: proc (ted_support_p, msg, code);

/* This routine renumbers the addressed portion of the buffer. It takes 1 or */
/*  2 arguments within a delimited string. The 1st argument is the beginning */
/*  number; the 2nd argument specifies the increment to use (assumes 10).    */
/*  Whatever follows the argument string will be left for further ted        */
/*  execution.                                                               */

/* --- This is not robust code. It does not include exhaustive error        */
/* --- checking. It is intended only to show how to use the interface       */
/* --- stucture's various data.                                             */

/* Usage: {1,$} |renumber /from,incr/                                        */

dcl  msg char (168)var,                  /* error message text        (OUT) */
     code fixed bin (35);                /* error code                (OUT) */

    if (ted_support_version_2 ^= ted_support.version)
    then do;                             /* make sure its correct version    */
       code = error_table_$unimplemented_version;
       return;                           /* can't handle this                */
    end;
    if (inp.de = 0)
    then do;                             /* must be some data to process     */
       msg = "Buffer Empty.";
       code = tederror_table_$Error_Msg;
       return;
    end;
/**** 1) Parse the arg list                                                  */
    req.nc = req.cc;                     /* move back to current location    */
    delim = rchr (req.nc);               /* save the delimiter char          */
    if (delim = " ") | (delim = NL)
    then do;
       code = tederror_table_$No_Delim1;
       return;
    end;
    req.nc = req.nc + 1;                 /* skip over it                     */
    i = verify (substr (rstr, req.nc), "0123456789") -1;
    ln = fixed (substr (rstr, req.nc, i)); /* get starting line number       */
    req.nc = req.nc + i;                 /* skip over part used up           */
    if (rchr (req.nc) = ",")
    then do;                             /* 2nd arg is present               */
       req.nc = req.nc + 1;              /* skip over the comma              */
       i = verify (substr (rstr, req.nc), "0123456789") -1;
```

```
      incr = fixed (substr (rstr, req.nc, i));  /* get increment          */
      req.nc = req.nc + i;                      /* skip over part used up */
   end;
   else incr = 10;                              /* supply the default     */
   if (rchr (req.nc) ^= delim)
   then do;
      msg = "Only 2 args allowed";
      code = tederror_table_$Error_Msg;
      return;
   end;
   req.nc = req.nc + 1;                         /* leave "next" for continued */
                                                /*  execution of request line data */
/**** 2) see if default address needed                                        */
   if addr_ct = 0
   then do;
      inp.sb = 1;                               /* the default is 1,$     */
      inp.se = inp.de;
      out.de = 0;                               /* forget anything already done here */
   end;
/**** 3) do all lines in address range                                       */
   do while (inp.sb <= inp.se);
      i = verify (substr (istr, inp.sb), "0123456789") -1;
      inp.sb = inp.sb + i;                      /* strip any existing line number */
      pic5 = ln;                                /* get display form of line number */
      substr (ostr, out.de+1, 5) = pic5;
      out.de = out.de + 5;                      /* place new number in output */
      i = index (substr (istr, inp.sb), NL); /* find line length          */
      substr (ostr, out.de+1, i) = substr (istr, inp.sb, i);
      out.de = out.de + i;                      /* add in the line of data */
      inp.sb = inp.sb + i;                      /* account for used-up input data */
      ln = ln + incr;                           /* move on to next number value */
   end;
/**** 4) tell ted its AOK                                                    */
   code = tederror_table_$Copy_Set;

dcl  i fixed bin (24);
dcl  delim char (1);                            /* arg list delimiter     */
dcl  ln fixed bin;                              /* line number value      */
dcl  incr fixed bin;                            /* line number increment  */
dcl  pic5 pic "99999";                          /* display form of line number */
dcl  NL char (1) int static options (constant) init ("
");
%include ted_support;

end ted_renumber_;
```

The ted_support.incl.pl1 segment contains the interface information needed to write an external request as well as an example of a simple request.

```
/* BEGIN INCLUDE FILE ..... ted_support.incl.pl1 ..... 03/16/81           */

/* more information may be found in ted_support.gi.info                   */
```

```
dcl  ted_support_p ptr;
dcl ted_support_version_2 fixed bin int static init(2);
dcl 1 ted_support based(ted_support_p),
    2 version fixed bin,       /* 1                                          */
    2 addr_ct fixed bin,       /* number of addresses given: 0,1,2      (IN) */
    2 checkpoint entry (       /* routine to update "safe" status       (IN) */
         fixed bin(21),        /* amount of input used up                    */
         fixed bin(21)),       /* amount of output used up                   */

    2 inp,                     /***** input string parameters               */
                               /* The input data may NOT be modified.       */
      3 pt ptr,                /* pointer to base of data string        (IN) */
      3 sb fixed bin(21),      /* index of addressed string begin       (IN) */
      3 lno fixed bin(21),     /* linenumber in data string of sb       (IN) */
      3 se fixed bin(21),      /* index of addressed string end      (IN/OUT) */
      3 de fixed bin(21),      /* index of data end                     (IN) */

    2 out,                     /***** output string parameters              */
      3 pt ptr,                /* pointer to base of output string      (IN) */
      3 de fixed bin(21),      /* index of data end (already copied) (IN/OUT) */
      3 ml fixed bin(21),      /* max length of output string           (IN) */

    2 req,                     /***** request string parameters             */
      3 pt ptr,                /* pointer to base of request string     (IN) */
      3 cc fixed bin(21),      /* index of current character            (IN) */
      3 nc fixed bin(21),      /* index of next character            (IN/OUT) */
      3 de fixed bin(21),      /* index of data end                  (IN/OUT) */
      3 ml fixed bin(21),      /* max length of requsest buffer         (IN) */

/* req.nc is initialized to req.de, i.e.  request line used-up.  A routine  */
/* can set req.nc to 1, put some data into req and set req.de                */
/* appropriately.  The data will be the next ted requests executed after    */
/* the routine returns.                                                      */

/* Or if req.nc is set equal to req.cc then the rest of the request line    */
/* will be executed after return.                                            */

    2 string_mode bit(1),      /* 0- line mode, 1- string mode          (IN) */
    2 current fixed bin(21),   /* current location                   (IN/OUT) */
                               /* current is initialized to "undefined"     */
    2 get_req entry (),        /* fill the request string with the next line */
                               /*  from ted's input stream. req.de will be  */
                               /*  updated to reflect the new length.       */
                               /* req.cc and req.nc are not changed.        */
    2 proc_expr entry          /* process the expression for global execution */
         (ptr,                 /* -> ted_support structure              [IN] */
         char (168) var,       /* message text                         [OUT] */
         fixed bin (35)),      /* completion code                      [OUT] */
    2 do_global entry          /* globally execute some action               */
         (entry (),            /* worker procedure                      [IN] */
         char (1),             /* which action, "g" or "v"              [IN] */
         ptr,                  /* -> ted_support structure              [IN] */
         char (168) var,       /* message text                         [OUT] */
```

```
        fixed bin (35)),     /* completion code                          [OUT] */
    2 reg_exp_p ptr,         /* -> the remembered regular expression area      */
    2 bcb_p ptr;             /* -> buffer control block                        */
/* ____________________________________________________________________________ */
/*                              ENTRY CONDITIONS                                */
/* ____________________________________________________________________________ */
/* Upon entering, three substructures describe the environment in which the     */
/* request is to operate. (Refer to the INPUT diagram) Note that the            */
/* "normal" operational steps are:                                              */
/* 1) ted copies the string from 1:inp.sb-1 to the output string                */
/* 2) ted_xyz_ takes care of the data from inp.sb:inp.se                        */
/* 3) ted copies the string from inp.se+1:inp.de to the output string           */
/* 4) ted sets "." as (possibly) specified by xyz                               */

/* The following 3 diagrams represent conditions upon entering ted_xyz_:        */
/* ____________________________________________________________________________ */
/*               req.pt                  (\ represents NL)                      */
/*                  |                                                           */
/* [REQUEST]        x 2,3|req /farfle/ 1,$P\.....................               */
/*                       |              |                       |              */
/*                    req.cc          req.de                  req.ml          */
/*                                    req.nc                                    */
/* ____________________________________________________________________________ */
/*               inp.pt                  (\ represents NL)                      */
/*                  |                                                           */
/* [INPUT]          now is\the time\for all\good men\to come.\.....             */
/*                       |              |                |                     */
/*                    inp.sb          inp.se           inp.de                   */
/* The request may make no modifications to the input string. It may make no    */
/*  assumptions about its location, i.e. that it occupies a segment all by     */
/*  itself.                                                                     */
/* ____________________________________________________________________________ */
/*               out.pt                  (\ represents NL)                      */
/*                  |                                                           */
/* [OUTPUT]  ?      now is\.....................................               */
/*           |      |                                          |               */
/*        current   out.de                                   out.ml           */
/* ____________________________________________________________________________ */

/* ____________________________________________________________________________ */
/*                              EXIT CONDITIONS                                 */
/* ____________________________________________________________________________ */
/* Assume a request replaces each addressed line with the string following      */
/* it, (in this case "farfle") and leaves "." at the beginning of the range.    */
/*               out.pt                  (\ represents NL)                      */
/*                  |                                                           */
/* [OUTPUT]         now is\farfle\farfle\.....................                  */
/*                       |              |                       |              */
/*                    current         out.de                  out.ml          */

/* ____________________________________________________________________________ */
/* 1) If the data after the string are to be treated as more ted requests,      */
/*  the request data would be left like this.                                   */
```

```
/*              req.pt                    (\ represents NL)                  */
/*                 |                                                         */
/* [REQUEST]       x 2,3|req /farfle/ 1,$P\.....................            */
/*                                      |   |                      |         */
/*                                  req.nc  req.de               req.ml     */
/* ------------------------------------------------------------------------- */
/* 2) If the request is going to return a string to be executed, the request */
/*  data (and buffer) would be left like this:                               */
/*              req.pt                    (\ represents NL)                  */
/*                 |                                                         */
/* [REQUEST]       -1,.lp w\ /farfle/ 1,$P\.....................            */
/*                 |       |                                  |              */
/*              req.nc  req.de                              req.ml           */

/* These are special return codes relating to ted:                           */
dcl (tederror_table_$Copy_Set,/* copy rest of input to output, and set "."    */
                              /*  from current. "rest of input" is the       */
                              /*  string which begins at char inp.se+1 and   */
                              /*  extends to inp.de. If the input has all    */
                              /*  been processed, then inp.se should be set  */
                              /*  to inp.de+1.                               */
     tederror_table_$NoChange,/* dont copy, dont set current                 */
     tederror_table_$Set,     /* dont copy, set current (in input buffer)    */
     tederror_table_$Error_Msg,/* msg is being returned.                     */
                              /*  no copy or set is done                     */
     tederror_table_$No_Delim1,/* missing 1st delimiter                      */
     tederror_table_$No_Delim2,/* missing 2nd delimiter                      */
     tederror_table_$No_Delim3)/* missing 3nd delimiter                      */
                   fixed bin(35)ext static;
/* Any other codes returned must be standard system codes.                   */
dcl  error_table_$unimplemented_version fixed bin(35) ext static;

dcl  istr char(inp.de) based(inp.pt);   /* the input string                  */
dcl  ichr(inp.de) char(1) based(inp.pt);
dcl  ostr char(out.ml) based(out.pt);   /* the output string                 */
dcl  ochr(out.ml) char(1) based(out.pt);
dcl  rstr char(req.ml) based(req.pt);   /* the request string                */
dcl  rchr(req.ml) char(1) based(req.pt);

/* These declarations are used if the expression search is needed by the     */
/* request. There are 2 parts to getting an expression search done:          */
/*        1) compiling        2) searching                                   */
/* If a function uses the remembered expression, it does this:               */
/*        call tedsrch_$search (ted_support.reg_exp_p,                       */
/*          ted_support.bcbp, string_begin, string_end, match_begin,         */
/*          match_end, search_end, msg, code);                               */

/* If a function utilizes an expression the user supplies, it must first be  */
/* compiled:                                                                 */
/*        if (expression_length > 0)                                         */
/*        then call tedsrch_$compile (addr (ichr (expression_begin)),        */
/*          expression_length, ted_support.reg_exp_p,                        */
/*          ted_support.string_mode, ""b, msg, code);                        */
```

```
/* This results in the remembered expression being changed to the one just   */
/* compiled.                                                                  */

/* If a function wishes to utilize a function without it being remembered    */
/* by ted, it may declare an area of its own and compile into it. It first   */
/* must be initialized:                                                       */
/*        dcl expr_area (200) bit (36);                                       */
/*        call tedsrch_$init_exp (addr (expr_area), size (expr_area));        */
%include tedsrch_;

/* END INCLUDE FILE ..... ted_support.incl.pl1 .....                         */
```

The tedsrch.incl.pl1 segment invoked in the previous example "%include tedsrch_;" contains:

```
/* BEGIN INCLUDE FILE ..... tedsrch_.incl.pl1 ..... 10/21/82 J Falksen */

dcl tedsrch_$init_exp entry ( /* initialize an expression area              */
          ptr,                /* -> compiled expression area          [IN] */
          fixed bin (21));    /*   length of area in words            [IN] */

dcl tedsrch_$compile entry (  /* compile a regular expression              */
          ptr,                /* -> regular expression to compile     [IN] */
          fixed bin (21),     /*   length thereof                     [IN] */
          ptr,                /* -> compiled expression area          [IN] */
          bit (1)aligned,     /* 0- line mode      1- string mode     [IN] */
          bit (1)aligned,     /* 0- reg expr       1- literal expr    [IN] */
          char (168) var,     /* error message                       [OUT] */
          fixed bin (35)      /* error status code                   [OUT] */
                    );

dcl tedsrch_$search entry (   /* search for expression                     */
          ptr,                /* -> compiled expression area          [IN] */
          ptr,                /* -> buffer ctl block for file         [IN] */
          fixed bin (21),     /* beginning of string to search in file [IN] */
          fixed bin (21),     /* end of string to search              [IN] */
          fixed bin (21),     /* beginning of match                  [OUT] */
          fixed bin (21),     /* end of match                        [OUT] */
          fixed bin (21),     /* end of string used for match        [OUT] */
          char (168)var,      /* error message return                [OUT] */
          fixed bin (35)      /* error status code                   [OUT] */
                    );

/* END INCLUDE FILE ..... tedsrch_.incl.pl1 ..... */
```

MULTICS TEXT EDITOR(TED) REFERENCE MANUAL ADDENDUM B

SUBJECT

Changes to the Manual

SPECIAL INSTRUCTIONS

This is the second addendum to CP50-00, dated December 1981. Refer to the Preface for "Significant Changes."

This manual is being reissued complete due to a production style change. The reader need only be concerned with those pages containing change bars or asterisks.

Insert the attached pages into the manual according to the collating instructions on the back of this cover. Throughout the manual, change bars in the margin indicate technical additions and asterisks denote deletions.

Note: Insert this cover after the manual cover to indicate the updating of the document with Addendum B.

SOFTWARE SUPPORTED

Multics Software Release 11.0

ORDER NUMBER

CP50-00B

October 1985

46524
1086
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
With the exception of the Front Cover, remove the manual in its entirety.	Insert the Addendum Self Cover behind the existing Front Cover, then insert the entire manual.

File No.: 1L13

CP50-00B

10/85

INDEX

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: MULTICS TEXT EDITOR (TED) REFERENCE MANUAL

ORDER NO. CP50-00B

DATED OCTOBER 1985

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

PLEASE FILL IN COMPLETE ADDRESS BELOW.

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

CUT ALONG LINE

PLEASE FOLD AND TAPE–
NOTE: U.S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA 02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

CUT ALONG LINE

FOLD ALONG LINE

FOLD ALONG LINE

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 155 Gordon Baker Road, Willowdale, Ontario M2H 3N7
In the U.K.: Great West Road, Brentford, Middlesex TW8 9DH
In Australia: 124 Walker Street, North Sydney, N.S.W. 2060
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

34577, 7.5C582, Printed in U.S.A.

CP50-00