M T S

The Michigan Terminal System

The MTS File Editor

Volume 18

February 1988

University of Michigan Information Technology Division
Research Systems

DISCLAIMER

The MTS Manual is intended to represent the current state of the Michigan Terminal System (MTS), but because the system is constantly being developed, extended, and refined, sections of this volume will become obsolete. The user should refer to the *U-M Computing News*, Computing Center Memos, and future Updates to this volume for the latest information about changes to MTS.

CONTENTS

PREFACE

The software developed by the Information Technology Division staff for the operation of the high-speed IBM 370-compatible computers can be described as a multiprogramming supervisor that handles a number of resident, reentrant programs. Among them is a large subsystem, called MTS (Michigan Terminal System), for command interpretation, execution control, file management, and accounting maintenance. Most users interact with the computer's resources through MTS.

The MTS Manual is a series of volumes that describe in detail the facilities provided by the Michigan Terminal System. Administrative policies of the Information Technology Division and the physical facilities provided are described in other publications.

The MTS volumes now in print are listed below. The date indicates the most recent edition of each volume; however, since volumes are periodically updated, users should check the file *CCPUBLICATIONS, or watch for announcements in the *U-M Computing News*, to ensure that their MTS volumes are fully up to date.

Volume 1 *The Michigan Terminal System*, January 1984
Volume 2 *Public File Descriptions*, January 1987
Volume 3 *System Subroutine Descriptions*, April 1981
Volume 4 *Terminals and Networks in MTS*, March 1984
Volume 5 *System Services*, May 1983
Volume 6 *FORTRAN in MTS*, October 1983
Volume 7 *PL/I in MTS*, September 1982
Volume 8 *LISP and SLIP in MTS*, June 1976
Volume 9 *SNOBOL4 in MTS*, September 1975
Volume 10 *BASIC in MTS*, December 1980
Volume 11 *Plot Description System*, August 1978
Volume 12 *PIL/2 in MTS*, December 1974
Volume 13 *The Symbolic Debugging System*, September 1985
Volume 14 *360/370 Assemblers in MTS*, May 1983
Volume 15 *FORMAT and TEXT360*, April 1977
Volume 16 *ALGOL W in MTS*, September 1980
Volume 17 *Integrated Graphics System*, December 1980
Volume 18 *The MTS File Editor*, February 1988
Volume 19 *Tapes and Floppy Disks*, November 1986
Volume 20 *Pascal in MTS*, December 1985
Volume 21 *MTS Command Extensions and Macros*, April 1986
Volume 22 *Utilisp in MTS*, February 1988
Volume 23 *Messaging and Conferencing in MTS*, March 1987

Other volumes are in preparation. The numerical order of the volumes does not necessarily reflect the chronological order of their appearance; however, in general, the higher the number, the more specialized the volume. Volume 1, for example, introduces the user to MTS and describes in general the MTS operating system, while Volume 10 deals exclusively with BASIC.

The attempt to make each volume complete in itself and reasonably independent of others in the series naturally results in a certain amount of repetition. Public file descriptions, for example, may appear in more than one volume. However, this arrangement permits the user to buy only those volumes that serve his or her immediate needs.

Richard A. Salisbury
General Editor

PREFACE TO VOLUME 18

The following changes have been made to the MTS File Editor since the previous edition of MTS Volume 18 was issued:

(1) The $EDIT command now accepts file-name patterns. Each file in the pattern sequence will be edited.

(2) The SAVE and REMEMBER commands have been added to save and remember states of the edit session.

(3) The VMARK command has been added to define an editor region in visual mode.

(4) The PRINT VMSG command has been added to print a message below the vruler/status line in visual mode.

(5) The VPF ATTN and VPF RETURN commands have been added for assigning the ATTN and RETURN keys in visual mode.

(6) The following new predefined editor variables have been added:

ED_COUNT
ED_SAVENAME
ED_TIME
ED_VMARK_FIRST_COL
ED_VMARK_FIRST_LINE
ED_VMARK_LAST_COL
ED_VMARK_LAST_LINE

EDIT MODE

BASIC CONCEPTS

The MTS file editor is used for editing MTS files. The editor is invoked by entering the MTS command

$EDIT filename

where "filename" is the name of the MTS file to be edited (the edit file). If "filename" is omitted from the $EDIT command, the previous edit file remains the edit file. "filename" may be an MTS file-name pattern, in which case each file satisfying the pattern will be sequentially edited.

A "one-shot" command may be executed by appending an edit command to the $EDIT command in the form

$EDIT filename :edit-command

Editor commands are read from the pseudodevice *SOURCE* and editor output messages, diagnostics, and verification comments are written on *SINK*. When the edit file is ready for editing, the editor prints a colon ":". This prefix character indicates that the editor is ready to process edit commands.

A simple edit session (using an unpatterned file name) is terminated by entering the STOP edit command or an end-of-file condition. The session may be suspended temporarily by entering the MTS or the RETURN edit command. A one-shot command will also leave the edit session loaded (suspended). All of the above responses cause the system to return to the caller (normally MTS command mode). While in MTS command mode, any MTS command may be executed. If a subsequent $EDIT command is entered after a previous edit session was suspended, the same edit session is reactivated; if the previous edit session was terminated, a new edit session is initiated. A simple edit session may be suspended any number of times. When an edit session is suspended, all checkpoint information and all settings of global editor SET options (described later in this section) are preserved. When an edit session is terminated, all of this information is released. The edit file may be changed in the same edit session by entering the edit command

EDIT filename

Selecting a new file for editing does not affect the setting of the global editor SET options; however, all checkpoint information for the previous file is released.

If a patterned file name is given with the $EDIT command, an MTS, RETURN, STOP command, or an end-of-file condition will cause the next file in the pattern sequence to be edited. An attention interrupt will abort the editing sequence.

There may be only one edit file at any time. Line-number range specifications given with "filename" are ignored; the entire file is the edit file. Any explicit or implicit concatenation to other files or devices is also ignored. The file is read with the MTS I/O modifiers for implicit concatenation and trimming disabled (¬IC and ¬TRIM). The file is written with trimming disabled.

The editor may be used for both line files and sequential files. However, for sequential files, lines may not be deleted or changed in length; they only may be inserted at the end of the file.

COMMAND STRUCTURE

Edit commands perform the editing functions on the edit file. These functions fall into three general categories:

(1) changing the contents of the file,

(2) searching the file for a character string or set of character strings, and

(3) setting global editor options and modifiers.

Each command may be abbreviated, usually by one or two characters. A few commands begin with nonalphabetic characters. The editor scans the entire command name to ensure that it is correctly spelled.

An attention interrupt immediately terminates the current edit command; the editor returns to edit command mode. A second attention interrupt without an intervening edit command causes the editor to return to MTS. The editor may be reentered by issuing the $EDIT command.

Line-number parameters, string parameters, and modifiers are used in edit commands. These are described below.

Line Numbers

Each line in a file is referenced by a line number. A line number is specified as a number ranging from –2147483.648 to +2147483.647 with at most three decimal places. There are seven special symbols that may also be used for line numbers:

*F	is the first line in the file
*L	is the last line in the file
*	is the current line in the file
*N	is the next line in the file
*P	is the previous line in the file
*VT	is the top line of the visual-mode screen
*VB	is the bottom line of the visual-mode screen

If the file is empty, the above seven special symbols initially have the value of 1.

Any of the specifications below may be used in place of a line-number parameter “lpar” in the command prototypes given in this volume.

(1) linenumber

This specifies a single line. This may be given in the form of an explicit line number or one of the above special symbols, e.g., 1, 50, –100, *F, or *L.

(2) linenumber linenumber

This specifies a range of lines beginning with the first line number and continuing to the second. The line numbers may be given in ascending or descending order, e.g., 10 20, 200 100.

(3) linenumber COUNT=n

This specifies "n" line numbers beginning with "linenumber". "n" is a directional count; if "n" is negative, the lines are taken in descending line-number order starting with the line specified.

(4) static predefined regions

See the section "Regions" below.

(5) dynamic predefined regions

See the section "Regions" below.

(6) /name

This is the line-number region specified by "/name". See the section "Regions" below for details of defining region names.

(7) COUNT=n

This is the same as (3), but starts with the current line (*).

(8) (linenumber,linenumber)

This is an alternate form for (2). The parentheses provide clarity when specifying line-number lists (10).

(9) (linenumber,COUNT=n)

This is an alternate form for (3). The parentheses provide clarity when specifying line-number lists (10).

(10) line-number lists

This is a list of one or more elements of the form (1) through (9), each element separated by a comma, e.g., (*F, 1 10, *L).

(11) COLUMNS={(l,r) | l}

A column range may be specified after any of the above line-number parameters. The column range restricts the action of the edit command to the width of the file between the left and right columns specified. See the section "Column Ranges" for a further description of the use of column ranges with edit commands.

(12) Hexadecimal line numbers are given as X'xxxxxxxx', where "xxxxxxxx" is from one to eight hexadecimal digits. When hexadecimal format is used, the internal format of the line number is required, which is the external format times 1000. For example, line

number 1.000 is represented internally as 1000 and line number 10.000 as 10000. Using hexadecimal format, these would be specified as X'3E8' and X'2710', respectively. There are two hexadecimal line numbers that have special meaning: X'80000000' represents *F, and X'7FFFFFFF' represents *L.

(13) Character Representation

Character line numbers are given as C'cccc', where "cccc" is four characters. When character format is used, the hexadecimal representation of the character string is used as the line number. As with hexadecimal format, the resulting hexadecimal number is the internal format of the line number. For example, the line number C'aaaa' is equivalent to X'81818181' or –2122219.135.

The editor maintains a current-line pointer which points to the current line (*) in the file. The current line is initially set to the first line in the file. All edit commands that read or write lines in the file move the current-line pointer unless the @NMCL modifier is given on the command or the MCL (MOVECURRENTLINE) option is OFF. The LINE command may be used to explicitly set the current line. In visual mode, an asterisk will mark the current line if the VMARKER option is ON.

If no line-number specification is given for an edit command requiring a line-number parameter, the current line (*) is assumed. Exception: The SCAN and MATCH commands act on the current line to the last line of the file (* *L), if no line-number parameter is specified.

Strings

A string is a sequence of characters delimited at each end by a current delimiter character. The set of acceptable delimiter characters is as follows:

```
' " ¢ < > ? : ; % & ¬ #
```

The first occurrence of a character from this set is taken as the current delimiter character for that string. If the command requires a pair of strings, they must be presented in the following form: the delimiter, the first character string, the same delimiter, the second character string, and the same delimiter. If hexadecimal conversion is ON, the character string is converted to internal character form only if it contains legal hexadecimal characters. The following examples illustrate the format of strings:

The character string:	'STRING'
The null string:	''
The hexadecimal string:	#E2E3D9C9D5C7#
A pair of strings:	'ABC'DEF'

If the final delimiter (e.g., ') is omitted when entering a string or pair of strings, the editor prints the comment

```
:Terminating " missing (Y,N)?
```

and waits for a response. If "Y", "YE", or "YES" (either upper- or lowercase) is entered, the missing delimiter is appended to the end of the command; otherwise, the command is canceled. If a negative response is given, this prompt may be repeated if the PATTERN option is ON. This is because the pattern processing algorithm attempts to parse the command after normal command processing has failed.

Modifiers

Command modifiers may be applied to edit commands to override the default action of an edit command. For example, the @NVERIFY modifier may be specified to suppress verification of edited lines. The modifiers are effective only for a single edit command.

Command modifiers are specified by appending the modifier directly to the command name or as the last parameter of the edit command line. The "@" modifier character must precede the modifier. A modifier is negated by prefixing the modifier name with the letter "N", the not sign "¬", or the minus sign "–". As many modifiers as required may be specified by concatenation. No blanks are allowed between the command name and its modifiers, or within a sequence of modifiers. For example,

```
ALTER@ALL@NVERIFY 15 COUNT=5 'gamma-rays'X-rays'

ALTER@ALL@¬VERIFY 15 C=5 'gamma-rays'X-rays'

ALTER 15 C=5 'gamma-rays'X-rays' @ALL@NVERIFY
```

The above three edit command sequences all perform the same function; they alter all occurrences of the string "gamma-rays" to "X-rays" in 5 consecutive lines of the file, starting at line number 15, without verifying (printing) the results.

The default action of the edit commands may be permanently changed by editor SET options. Each modifier has a corresponding SET option, which may be set to reverse the default action of the modifier. In the above example, another equivalent command sequence would be

```
SET VERIFY=OFF ALL=ON
ALTER 15 C=5 'gamma-rays'X-rays'
```

The modifiers are described below. Each description gives the shortest form of the modifier followed by a list of acceptable longer names in parentheses.

@A (@ALL) Default: @NA

If @A is specified, the ALTER, CHANGE, MATCH, and SCAN commands will process *all* occurrences of the pattern throughout the specified range of the file. If @NA is specified, these commands will process only the first occurrence of the pattern.

@AC (@ANYCASE) Default: @NAC

If @AC is specified, the ALTER, CHANGE, MATCH, and SCAN commands will match any occurrence of alphabetic characters in the pattern regardless of case. If @NAC is specified, these commands will match only the exact alphabetic characters as specified in the pattern.

@CH (@CHECKPOINT) Default: @CH

If @CH is specified and if the checkpoint buffer has been opened by the CHECKPOINT command, the results of edit commands are recorded in the buffer so that they may be undone if the need arises. See the section "Checkpoint/Restore Facility" below.

@COL (@COLUMN) Default: @COL

If @COL is specified, the action of the edit command applies only to the column range set by the COLUMN command (see the description of the COLUMN command). This modifier does not apply to the PRINT command; the @WINDOW modifier should be used for printing.

@HEX (@X, @HEXADECIMAL) Default: @NHEX

If @HEX is specified, all character strings given in the command are treated as hexadecimal strings. For example, the string

```
"C8C5D3D3D640E3C8C5D9C5"
```

is equivalent to

```
"HELLO THERE"
```

when this modifier is specified. Verification of the line text is printed in hexadecimal. Using the @HEX modifier with the VISUAL command will cause the visual screen data to be represented in hexadecimal format. To change the visual screen back to character format, another VISUAL command (without the @HEX modifier) must be executed.

@LEN (@LENGTH, @LINELENGTH, @LL) Default: @NLEN

If @LEN is specified, verification of a line includes its length.

@LNR (@LINENUMBER) Default: @LNR

If @LNR is specified, verification of a line includes its line number.

@MACRO Default: @NMACRO

The INSERT and PROCEDURE commands accept the @MACRO modifier which causes lines read by these commands to be processed with the MTS FDname modifiers @MACRO@MFR applied (any line beginning with the MTS command macro flag ">" will be processed by the MTS macro processor). By default, lines are read without these MTS modifiers.

@MCL (@MOVECURRENTLINE) Default: @MCL

If @MCL is specified, the current-line pointer (*) is moved by the action of edit commands. If @NMCL is specified, the current-line pointer remains unchanged, even after a successful SCAN or MATCH command.

@NOT Default: @NNOT

If @NOT is specified, the pattern search in the SCAN and MATCH commands is successful when the pattern is *not* found. That is, the search succeeds at any position in the line where the pattern string fails to match. For example, this function in conjunction with the COLUMN command may be useful in finding a line which does not contain a blank in a given column range.

@OPL (@ONCEPERLINE) Default: @NOPL

If @OPL is specified, only one operation such as an alteration or successful scan is allowed per line. The specification of @OPL often implies the specification of @A. @OPL is necessary when modifying a null string to a nonnull string if @A is also applied.

@PA (@PRINTALL) Default: @NPA

If @PA is specified, verification of every alteration of a line is given when both @A and @V are specified. If @NPA is specified, only one verification is printed per line regardless of the number of alterations made. The @PA modifier only applies to the ALTER and CHANGE commands.

@PC (@PRINTCOLUMN) Default: @NPC

If @PC is specified, the column number of any successful pattern search is printed.

@PLN (@PREFIXLINENUMBER) Default: @PLN

If @PLN is specified and an increment has been specified on the INSERT or VINSERT command, the user will be prompted for insertion lines with the line numbers at which the lines are to be inserted.

@RS (@RESCAN) Default: @NRS

If @RS is specified in combination with the @A modifier, the ALTER and CHANGE commands resume pattern scanning at the replaced string (rather than after it) after a successful alteration. If @RS and @A are specified, accidental infinite loops may be created. For example,

```
ALTER@A@RS 123 "the"then"
```

would cause the following line:

```
Farmer in the dell
```

to become:

```
Farmer in thennnnnnnnnnnnnn ...
```

To limit accidental loops, the rescan count at the same column position is limited to a number equal to the maximum possible length of the line.

One useful purpose for the @ALL@RESCAN combination is to alter strings of blanks to one blank:

```
ALTER@A@RS /FILE "  " "
```

where all pairs of blanks are changed to a single blank.

@RTL (@RIGHTTOLEFT) Default: @NRTL

If @RTL is specified, pattern scanning is performed from right to left within a line, i.e., the last occurrence of the pattern in the line is matched first. Note that the pattern does not need to be specified differently. For example,

```
SCAN@RTL 'the'
```

but not

```
SCAN@RTL 'eht'
```

may be used to match the underlined part of the following sentence:

```
I am the only one in the theatre.
```

@TR (@TRIM) Default: @NTR

If @TR is specified, all lines written or changed in the file are trimmed to remove all but one trailing blank.

@TX (@TEXT,@LINETEXT) Default: @TX

If @TX is specified, verification of a line includes its contents.

@V (@VERIFY) Default: @V

If @V is specified, the resulting lines are printed after they are processed by an edit command. The following modifiers govern the output produced by verification: @HEX, @LEN, @LNR, @PA, @TX, and @W.

@VMV (@VISUALMODEVERIFY) Default: @NVMV

If @VMV is specified, verification of commands executed from the work area in visual mode will appear in the conversation buffer, i.e., when visual mode is exited, the verification will appear on the screen.

@W (@WINDOW) Default: @W

If @W is specified, the text of lines printed during verification includes only those column ranges specified by the WINDOW command.

Column Ranges

Several edit commands apply only to a specified column range in the file. Two column pointers are associated with the column range, which is defined as extending from the character pointed to by the first column pointer through (and including) that pointed to by the second column pointer. Any edit command that specifies a line range “lpar” or a static predefined region may use a COLUMNS specification.

The column pointers may be specified within an edit command by using the COLUMNS keyword in the form

COLUMNS=(f,l)

where “f” is the first column pointer and “l” is the last column pointer. COLUMNS may be abbreviated as COL. For example, the command

```
SCAN 1 10 COL=(2,4) 'ABC'
```

will scan lines 1 through 10 in columns 2 through 4 for the string 'ABC'. If only “f” is specified, the parentheses may be omitted, e.g.,

```
COL=f
```

In this case, the last column pointer defaults to the value set by the previous COLUMN command. In order to set a column range to be one character wide, a specification such as COL=(3,3) must be used.

The column pointers must be in the range of 1 to 32767, the default pointer values. The first column pointer must be less than or equal to the value of the second pointer. The values assigned to the column range pointers do not place any additional restrictions on the length of the individual lines within the files.

The COLUMN command may be used to set globally the column pointers for all edit commands that recognize column ranges. This command is given in the form

COLUMN f l

where “f” and “l” are as defined above. If the edit command specifies the COLUMNS keyword, it will override the setting of the COLUMN command.

REGIONS

A range of lines may be specified by a user-defined name. This name is called a region and takes the form

/name

where “name” is from 1 to 7 alphanumeric characters. Several regions may be defined within a file.

The REGION command defines and establishes the scope of a region. The command is given in the form

REGION /name lpar

where “/name” is the name of the region being defined and “lpar” is the scope of the region. The scope value may be any valid line-number specification “lpar” (see the section “Line Numbers” above). A region name may be used in place of a line-number parameter in an edit command.

When region names appear as line-number parameters in the definition of a new region via the REGION command, the value of those regions (i.e., the line-number parameters they consist of) are copied into the values of the region being defined. Thus, if the current line is 9, the command

```
REGION /REG1 1,2,/HEAD
```

will result in a definition for REG1 of lines 1 and 2 and the portion of the file from the first line of the file through line 9. If the current line is later moved to line 400, the definition of REG1 remains the same.

Sometimes it is desirable to make the definition of REG1 change as the value of /HEAD changes. This can be done by inserting two slashes (//) at the front of the region name in the line-number parameters of the REGION command. For example, if line 9 is the current line, the command

```
REGION /REG1 1,2,//HEAD
```

would result in the same definition for REG1 as long as line 9 remains the current line. But if the current line is moved to line 400, the definition of REG1 changes to lines 1 and 2 and the portion of the file from the first line through line 400. This type of deferred evaluation may be done for any region; thus, the command

```
REGION /X //Y
```

will cause the current value of region /Y to be used each time a reference is made to /X in a command.

There are several predefined regions.

(1) dynamic predefined regions

/FILE or /F	specifies the entire file: first line to last line (*F *L).
/TAIL or /T	specifies the remainder of the file after the current line (*N *L).
/HEAD or /H	specifies the beginning of the file through the current line (*F *).
/BACK or /B	specifies, in reverse order, the portion of the file starting with the line just before the current line and ending with the first line of the file (*P *F).
/REVERSE or /R	specifies the entire file in reverse order, last line to first line (*L *F).
/VISUAL or /V	specifies the current visual-mode screen (*VT *VB).
/VMARK	specifies the current user-defined visual-mode region (see the VMARK visual-mode command).

(2) dynamic predefined regions

/SCAN or /S	specifies all lines successfully matched by the last SCAN command.
/MATCH or /M	specifies all lines successfully matched by the last MATCH command.

EDITOR INITIALIZATION FILE

An editor initialization file may be specified by the MTS command

$SET INITFILE(EDIT)=filename

When the editor is being initialized, a check will be made to determine if an initialization file has been specified. If so, the editor will read initializing commands from the specified file before reading commands from *SOURCE* or before processing the command specified on the $EDIT command.

The editor initialization file is the preferred method of presetting editor options and edit procedures. Instead of initializing the editor in the user's sigfile, which is effective only once and only for the invocation of the editor from the MTS command language, using the initialization file is effective for all initializations of the editor including initializations of the editor via the EDIT subroutine. The EDIT subroutine is used, for example, to invoke the editor from the MTS message system ($MESSAGE).

For example, a user might include the command

```
$SET INITFILE(EDIT)=EDITSETUP
```

in his or her sigfile, where the file EDITSETUP could contain the editor commands

```
SET PATTERNS=ON
CHECKPOINT FULL
```

The initialization file may contain any edit command except for the MTS, RETURN, or STOP commands. It also may contain visual-mode commands and edit-procedure definitions. These are described in the sections "Visual-Mode Commands" and "Edit Procedures."

The initialization file facility may be disabled by the MTS command

$SET INITFILE(EDIT)=OFF

CHECKPOINT/RESTORE FACILITY

The checkpoint/restore facility may be used to preserve the state of the edit file so that recovery may be made from accidental destruction of all or part of its contents. This facility is controlled by the CHECKPOINT command and is given in the form

CHECKPOINT {ON | OFF | NEVER}

When the checkpoint/restore facility is enabled, a checkpoint buffer is opened. This buffer is used to record the action of edit commands and can be used to recover the previous contents of the file, if necessary.

If the CHECKPOINT ON command is given, a checkpoint buffer is opened, and the current contents of the file is established as the checkpoint base. All commands that modify the file are recorded in the buffer. Either the RESTORE or UNDO commands may be used to recover the contents of the file as of the checkpoint base. The RESTORE command is used to restore the entire file or a portion thereof and is given in the form

RESTORE [lpar]

For example, the command

```
RESTORE 1 20
```

restores the contents of lines 1 through 20 of the file. The RESTORE command always restores the specified portion of the file to the state of the checkpoint base. The UNDO command is used to restore the file to its state before the last edit command that modified the file and is given in the form

UNDO

For example, the command sequence

```
DELETE 1 20
UNDO
```

deletes lines 1 through 20 and then restores them to the file.

If the CHECKPOINT OFF command is given, a checkpoint buffer is still opened, but only the action of the last command that modified the file is recorded in the buffer. At this level, the UNDO command may be used to reverse the action of the last file-modifying command. The RESTORE command may only be used to restore a subset of lines changed by the *last* command. This is particularly useful when a line has been unintentionally altered through the use of the @A modifier.

If the CHECKPOINT NEVER command is given, no checkpoint information is saved. The previous checkpoint buffer is discarded. The RESTORE and UNDO commands have no effect.

If the CHECKPOINT command has not been given, the level of the checkpoint/restore facility defaults to OFF.

The CHECKPOINT global option and the @CH modifier may be used in conjunction with the CHECKPOINT command to further control the checkpoint/restore facility. The option may be set to either ON or OFF by the SET command; the default is ON.

When both the CHECKPOINT option is ON and the CHECKPOINT ON command is given, the action of all commands is recorded in the checkpoint buffer. However, the checkpointing feature may be suppressed for a specific command by appending the @NCH modifier to the command. In this case, the action of that command is not recorded in the checkpoint buffer. Alternatively, if the CHECKPOINT option is set to OFF and checkpointing has been initiated with the CHECKPOINT ON command, then only those commands appended with a @CH modifier are recorded in the checkpoint buffer. These two methods allow the user to select the commands that are to be checkpointed.

In cases where *many* modifications are being made to a file, the virtual memory available to checkpoint changes may be exhausted if CHECKPOINT ON is specified. The error messsage

```
GETSPACE unable to allocate space. Checkpoint information lost.
```

will appear if this occurs. This error is less likely to occur if CHECKPOINT is OFF.

EDIT PROCEDURES

An edit procedure is a sequence of edit commands stored as a program within the editor. These commands are executed by the editor when the edit procedure is executed. This allows the user to repeatedly execute commands without having to explicitly reenter them.

The PROCEDURE command is used to create an edit procedure and to enter the commands that comprise the edit procedure. The PROCEDURE command takes the form:

PROCEDURE !name

where “name” is a 1- to 254-character, alphanumeric, edit-procedure name (the ! prefix must always be present). After the PROCEDURE command is given, the user will be prompted to enter the procedure

commands comprising the body of the list. For example, the following simple procedure !JUST65 left-justifies a file to fit within the column range of 1 to 65:

```
:PROCEDURE !JUST65
?JUSTIFY /FILE LEFT 1 65
?END
:
```

The definition of the edit procedure is terminated by entering the END command, a null line, or an end-of-file. If the user already had an edit procedure named !JUST65, its contents would be erased before the new command list is entered.

The GOTO command may be used within a procedure to control looping. For example, the following procedure !LIST may be defined to print all lines that begin with the character string "***":

```
:PROCEDURE !LIST
?LINE@NV *F
?LOOP: SCAN@NV * *L POS(0) '***'
?      GOTO END ON FAILURE
?      PRINT *
?      LINE@NV *N
?      GOTO END ON ENDOFFILE
?      GOTO LOOP
?END
:
```

Three edit procedure switches are defined. Two of these switches, SUCCESS and FAILURE, are set as the result of the success or failure of an edit command that performs a pattern-searching operation (ALTER, MATCH, SCAN, etc.). Such a command succeeds if it fulfills its function at least once within its specified line range; in this case, the SUCCESS switch is enabled and the FAILURE switch is disabled. Such a command fails if it exceeds its line range or encounters an end-of-file before fulfilling its function; in this case, the FAILURE switch is enabled and the SUCCESS switch is disabled. The third edit procedure switch is the ENDOFFILE switch. This switch is enabled if an end-of-file is encountered during the processing of an edit command or if a line-number parameter refers to an empty set of lines; it is disabled if the command is completed without an end-of-file occurring (at least one line must be specified by the line-number range parameter).

These switches may be tested by the GOTO command to allow conditional branching in the procedure. In the above example, the command sequence

```
SCAN@NV * *L POS(0) '***'
GOTO END ON FAILURE
```

branches to the end of the procedure (exits) if the FAILURE switch is enabled by the failure of the completion of the SCAN command, i.e., if the pattern POS(0) '***' is not found within the * *L line range.

An optional label may be prefixed to each command in the procedure command list. The label may be from 1 to 8 characters long and must be immediately followed by a colon. This label is used for branching by the GOTO command. In the above example, the command sequence

```
LOOP: SCAN@NV * *L POS(0) '***'
      ...
      GOTO LOOP
```

illustrates the use of labels and GOTO looping.

An edit procedure is executed via the EXECUTE command or by issuing the edit procedure name "!name" as a command:

EXECUTE !name

or

!name

The following example creates an edit procedure !ALPHA, which alters all occurrences in the file of the string 'alpha' to the string 'beta', while saving a copy of each altered line in the temporary file –ALPHA.

```
PROCEDURE !ALPHA
LINE@NV *F
LOOP: SCAN@NV * *L 'alpha'
      GOTO END ON FAILURE
      COPY@NV * TO F=-ALPHA(*L+1)
      ALTER * 'alpha'beta'
      GOTO LOOP
END
```

This edit procedure may be executed by the command

```
EXECUTE !ALPHA
```

The execution processing for this procedure is as follows. The first command sets the current-line pointer to the first line of the edit file. A SCAN command is then executed to scan for the first occurrence of the string 'alpha'. If such a string is found, the current-line pointer is moved to the line containing the string. A COPY command is executed to save the line in the temporary file and then an ALTER command is executed to alter the 'alpha' to 'beta'. This continues until the SCAN command fails.

The @NV modifier is appended to the SCAN and COPY commands in the above example to suppress the printing of command verification. Command verification could be suppressed for all commands in the procedure by issuing the edit command SET VERIFY=OFF at the beginning of the procedure and then issuing the command SET VERIFY=ON at the end of the procedure:

```
PROCEDURE !ALPHA
SET VERIFY=OFF
LINE *F
LOOP: SCAN * *L 'alpha'
      GOTO EXIT ON FAILURE
      COPY * TO F=-ALPHA(*L+1)
      ALTER * 'alpha'beta'
      GOTO LOOP
EXIT: SET VERIFY=ON
END
```

A procedure may be executed more than once by specifying a count on the EXECUTE command in the form

EXECUTE !name count

or

!name count

where "count" specifies the number of times the procedure is to be executed. For example, the following procedure !LISTN could be defined to list the next "n" lines of the file:

```
:PROCEDURE !LISTN
?LINE@NV *
?PRINT *
?LINE@NV *N
?END
:
```

In order to list the next 10 lines of the line starting from the current line, the procedure would be executed as

```
EXECUTE !LISTN 10
```

The special edit counter COUNT also may be used with the GOTO command to control the number of times a procedure or part of a procedure is executed. Initially, COUNT is set to the value of "count" given on the EXECUTE command. If "count" is omitted, COUNT is set to one. Each time the command GOTO COUNT is executed within the procedure, the value of count is decremented by one. When COUNT reaches zero, execution of the procedure is terminated. For example, the procedure !LIST given above could be rewritten to print out the next "n" lines in the file that begin with "***":

```
:PROCEDURE !LIST
?LINE@NV *F
?LOOP: SCAN@NV * *L POS(0) '***'
?      GOTO END ON FAILURE
?      PRINT *
?      GOTO COUNT
?      LINE@NV *N
?      GOTO END ON ENDOFFILE
?      GOTO LOOP
?END
```

The edit procedure is checked for correct syntax when it is executed. If errors are detected, execution is suppressed. The procedure may be edited to correct the errors in the same manner as a file is edited, i.e.,

EDIT PROCEDURE !name
edit-command
.
.

The contents of the procedure may be printed by giving the command

PRINT !name

The names of all currently defined procedures may be printed by giving the command

PRINT PROCEDURES

Execution of an edit procedure is not the same as the execution of an edit command. There are some subtle, but important, differences:

(1) The action of an entire edit procedure may not be reversed by an UNDO command since an UNDO only reverses the action of the last edit command. However, if CHECKPOINT ON is given at the beginning of the procedure, a RESTORE command will reverse the action of the entire procedure.

(2) The time limit that governs individual edit commands does not govern an entire edit procedure (see the SET TIMEINTERVAL command). Therefore, it is possible to code infinite loops into a procedure. Care should be taken to ensure that the terminating conditions of a loop (ON SUCCESS, ON FAILURE, ON ENDOFFILE) will actually be met at some point.

(3) The value of the current-line pointer may change many times as an edit procedures executes. Users should take care to initialize and increment lines correctly. Notice that this was done with the commands LINE@NV *F and LINE@NV *N in the procedure example above.

The definition of an edit procedure is normally discarded when the edit session is terminated. However, the definition may be saved and restored in one of three ways:

(1) The definition may be inserted into a sigfile file so that it is redefined each time the user signs on. For example, the procedure !JUST65 could be stored in part of the users sigfile in the following manner:

```
$EDIT
*
* Procedure to justify a file
*
PROCEDURE !JUST65
JUSTIFY /FILE LEFT 1 65
END
MTS
```

(2) The definition may be inserted into an editor initialization file so that it is redefined each time the editor is entered. This method is similar to the above method except that the $EDIT and MTS commands are omitted:

```
*
* Procedure to justify a file
*
PROCEDURE !JUST65
JUSTIFY /FILE LEFT 1 65
END
```

(3) The definition may be saved in a permanent file and then retrieved when the user requires the procedure. The procedure (assuming that it is currently defined by the editor) is saved by issuing the following commands:

```
$CREATE PROC.JUST65
$EDIT
EDIT PROC !JUST65
COPY /FILE TO F=PROC.JUST65
```

The procedure may be retrieved by issuing the following commands:

```
$EDIT
EDIT PROC !JUST65
COPY F=PROC.JUST65 TO *
```

VISUAL MODE

For the display-screen terminal user, the visual-mode feature provides a very powerful method for entering data into a newly created file or for editing an existing file. The visual-mode feature is supported on a wide variety of display-screen terminals and is described in the section "Visual Mode" at the end of this volume.

PATTERN MATCHING

The editor provides a pattern-matching feature that allows the user to perform more sophisticated operations on a file than mere string searching or replacement. This feature is described in the section "Pattern Matching" at the end of this volume.

RECORDING THE STATUS OF THE EDITOR

The CSSET subroutine is called after each editor command is executed. This enables the MTS command-language extensions facility (the MTS macro processor) to determine the result of an editor command. The following codes are used:

Origin value: 648

Status	*Summary*	*Code*
Execution success	0	0
Warning issued	1	0
End-of-file	1	1
"Search failed"	1	2
Attention occurred	2	1
Illegal modifier	2	2
Illegal command	2	3
Prescan failed	2	4
Severe error	2	5
Error message issued	2	6
Execution failed	2	7
Parse failed	2	8
Unknown failure	2	–1

See the description of the CSSET subroutine in MTS Volume 3, *System Subroutine Descriptions*, and in MTS Volume 21, *MTS Command Extensions and Macros*, for further information about using the editor status information.

MTS EDITOR COMMANDS

The following notation conventions are used in the prototypes of the commands:

lowercase	represents a generic type which is to be replaced by an item supplied by the user.
uppercase	indicates material to be repeated verbatim in the command.
brackets [ ]	indicate that material within the brackets is optional.
braces { | }	indicate that the material within the braces represents choices, from which exactly one must be selected. The choices are separated by vertical bars.
ellipsis ...	indicates that the preceding syntactic unit may be repeated.
underlining	indicates the minimum unambiguous form of the command or parameter. Longer abbreviations are accepted.

In the command descriptions that follow, a string expression "strexp" is a combination of one or more strings and/or variables whose values are strings.

The following pages give a complete summary of the commands in the editor command language.

Summary of Editor Command Prototypes

Commands	Parameters
ALTER	[lpar] ['string1'string2']
ALTER	[lpar] pattern
APPEND	[lpar] [strexp]
BLANK	[lpar] [strexp]
CHANGE	[lpar] ['string1'string2']
CHANGE	[lpar] pattern
CHECKPOINT	[{ON | OFF | NEVER}]
CLOSE	
COLUMN	[first [last]]
COMBINE	[lpar] [LEN=n]
{COMMENT | *}	[message]
CONCATENATE	[lpar] [LEN=n]
COPY	[{lpar | F=FDname [COUNT=n]}] [[TO] {linenumber | F=FDname}] [COPIES=n]
CORRECT	[lpar] [strexp]
COUNT	[lpar]
DELETE	[lpar] [OK]
EDIT	{filename | PROCEDURE !name | VPF pfname | MACRO macro-name} [[:]edit-command]
EXECUTE	{!name [n] | 'edit-command' | ED_WORK}
EXPLAIN	[command]
GOTO	label [[ON] condition]
INSERT	[linenumber] [strexp]
INSERT	[linenumber] [INCREMENT=increment]
JUSTIFY	[lrange] [{LEFT | RIGHT | BOTH} lmar rmar]
LET	variable=value
LINE	[linenumber] [{increment | COUNT=n}]
MATCH	[lpar] ['string']

MATCH	[lpar] pattern
MATCH	VARIABLE=var-name pattern
MCMD	MTS-command
MOVE	[lpar] [[TO] {linenumber | F=FDname}]
MTS	[MTS-command]
OVERLAY	[lpar] [strexp]
PRINT	[lpar] [CHECKPOINT]
PRINT	{COLUMNS | FILENAME | PROCEDURES | REGIONS | SAVES | VARS | VPFS | WINDOWS | XECS}
PRINT	{VAR name | !name}
PRINT	{MESSAGE | MSG | VMSG} strexp
PROCEDURE	!name
REGION	/name lpar
REMEMBER	[name]
RENUMBER	[first last] [beg [inc]]
REPLACE	[lpar] [strexp]
RESTORE	[lpar]
RETURN	
SAVE	[name]
SCAN	[lpar] ['string']
SCAN	[lpar] pattern
SCAN	VARIABLE=var-name pattern
SET	keyword=value **...**
SHIFT	[lpar] {LEFT | RIGHT} count
SPREAD	[lpar]
STOP	
TRANSLATE	[lpar] ['string1'string2']
TRANSLATE	[lpar] [[FROM] fromset] [TO] toset
TRANSLATE	[lpar] strexp1=strexp2
UNDO	
UNLK	
UNLOCK	
VISUAL	[linenumber]
WINDOW	[left [right]]
WINDOW	{LEFT | RIGHT} [count]
XEC	!name
/name	
!name	[n]
±n	
±n.n	

An attention interrupt immediately terminates the current edit command; the editor returns to edit command mode. A second attention interrupt without an intervening edit command causes the editor to return to MTS. The editor may be reentered by issuing the $EDIT command.

ALTER

Edit Command Description

Purpose: To replace parts of lines.

Prototype:
(a) ALTER [lpar] 'string1'string2'
(b) ALTER [lpar]
(c) ALTER [lpar] pattern
(d) ALTER VARIABLE=vname pattern

Action: With prototype (a), the first occurrence of "string1" in the given line-number range "lpar" is replaced by "string2".

With prototype (b), the strings given in the most recent ALTER command that specified strings are used for the replacement.

Either "string1" or "string2" may be the null string. If "string1" is the null string, it will match before the first character, in between each character, and after the last character in the column range. Unless @RS is specified, the scan will resume immediately after the inserted "string2", if @A is specified.

With prototype (c), a pattern structure is used to match the line-number range "lpar" for the replacement operations.

With prototype (d), a pattern structure is used to match the value of the variable "vname" for the replacement operations. See the section "Pattern Matching" for details on using patterns with the editor.

The action of this command is identical to that of the CHANGE command.

Modifiers: @A, @AC, @CH, @COL, @LEN, @LNR, @MCL, @OPL, @PA, @PC, @RS, @RTL, @TR, @TX, @V, @VMV, @W, @X

Conditions: SUCCESS if a replacement is made. EOF if a read beyond the end of the file is attempted.

Examples:

```
ALTER 123 'IT'THAT'
```

The above command alters the first occurrence of IT in line 123 to THAT.

```
ALTER@ALL /FILE '1984'1985'
```

The above command alters all occurrences in the file of 1984 to 1985.

APPEND

Edit Command Description

Purpose: To append additional data on the end of lines.

Prototype:
(a) APPEND [lpar] strexp
(b) APPEND [lpar]

Action: With prototype (a), the string specified by the string expression "strexp" is appended to the end of all of the lines specified by the line-number range "lpar". The PATTERNS option must be ON if "strexp" is more complex than a simple string.

With prototype (b), each line of the given line-number range is printed on the terminal and the user is prompted for a string to be appended to that line. If the user enters an end-of-file, the command is terminated and the last line printed is not modified.

If the column range starts beyond the end of the line, pad characters are inserted from the end of the line up to the beginning of the column range.

Modifiers: @CH, @COL, @LEN, @LNR, @MCL, @TR, @TX, @V, @VMV, @W, @X

Conditions: SUCCESS if at least one line is appended. EOF if a read beyond the end of the file is attempted.

Examples:

```
APPEND /FILE " "
```

The above command appends a blank to the end of every line in the file.

```
APPEND@NV@TRIM /FILE " "
APPEND@NV /FILE COL=35
```

The above two commands may be used to document an assembler source program. The first APPEND command ensures that all operand fields are separated from the comment field by one blank. The second APPEND command prints the statement and then prompts for a comment. The command will not create a continued statement if the comment extends the length of the line beyond column 71.

```
APPEND 3 10 DUPL(".",6)
```

The above example appends 6 periods to the end of lines 3 through 10.

BLANK

Edit Command Description

Purpose: To blank-out portions of lines.

Prototype:
(a) BLANK [lpar] strexp
(b) BLANK [lpar]

Action: With prototype (a), each line in the given line-number range "lpar" is blanked according to the pattern of the string specified by the string expression "strexp", i.e., positions in the line corresponding to the current fill character (blank by default) in the string are replaced with fill characters; positions in the line corresponding to nonfill characters are not changed. The PATTERNS option must be ON if "strexp" is more complex than a simple string.

With prototype (b), each line of the given line-number range is printed on the terminal and the user is then prompted for a string to be used as the blanking pattern for that line. If the user enters an end-of-file, the command is terminated and the last line printed is not modified.

Modifiers: @CH, @COL, @LEN, @LNR, @MCL, @TR, @TX, @V, @VMV, @W, @X

Conditions: SUCCESS if at least one line is blanked. EOF if a read beyond the end of the file is attempted.

Example:

```
BLANK 123 "AAA    "
```

The above command replaces the fourth through the sixth characters of line 123 with blanks.

CHANGE

Edit Command Description

Purpose: To replace parts of lines.

Prototype:
(a) CHANGE [lpar] 'string1'string2'
(b) CHANGE [lpar]
(c) CHANGE [lpar] pattern
(d) CHANGE VARIABLE=vname pattern

Action: With prototype (a), the first occurrence of "string1" in the given line-number range "lpar" is replaced by "string2".

With prototype (b), the strings given in the preceding CHANGE command that specified strings are used for the replacement.

Either "string1" or "string2" may be the null string. If "string1" is the null string, it will match before the first character, in between each character, and after the last character in the column range. Unless @RS is specified, the scan will resume immediately after the inserted "string2", if @A is specified.

With prototype (c), a pattern structure is used to match the line-number range "lpar" for the replacement operations.

With prototype (d), a pattern structure is used to match the value of the variable "vname" for the replacement operations. See the section "Pattern Matching" for details on using patterns with the editor.

The action of this command is identical to that of the ALTER command.

Modifiers: @A, @AC, @CH, @COL, @LEN, @LNR, @MCL, @OPL, @PA, @PC, @RS, @RTL, @TR, @TX, @V, @VMV, @W, @X

Conditions: SUCCESS if a replacement is made. EOF if a read beyond the end of the file is attempted.

Example:

```
CHANGE 123 'IT'THAT'
```

The above command alters the first occurrence of IT in line 123 to THAT.

CHECKPOINT

Edit Command Description

Purpose: To save the current status of the file so that future editing may be undone back to a certain point.

Prototype:
(a) CHECKPOINT [ON]
(b) CHECKPOINT OFF
(c) CHECKPOINT NEVER

Action: The checkpoint/restore/undo facility provides the capability of establishing a base, or checkpoint, from which all or any part of the file may be restored (see the section "Checkpoint/Restore Facility").

CHECKPOINT ON opens the checkpoint buffer and establishes the current status of the file as the checkpoint base. Any previous information in the buffer is discarded. Either the RESTORE or UNDO commands may be used later to restore the file, or a portion thereof, to this checkpoint base. Issuing the CHECKPOINT command without a parameter is synonymous with issuing CHECKPOINT ON. Issuing CHECKPOINT ON when CHECKPOINT is already enabled provides a *new* base for RESTORE and UNDO; that is, all previous checkpoint information is released.

CHECKPOINT OFF still opens the checkpoint buffer. How ever, only the effects of the *last* command that changed the file are recorded in the buffer. The UNDO command may be used to reverse the action of that command. The RESTORE command may be used to restore a subset of the lines that were changed by the last command. OFF is the default level of the checkpoint/restore facility.

CHECKPOINT NEVER specifies that no checkpoint information is saved; neither the RESTORE nor the UNDO commands have any effect.

The checkpoint buffer contains the line numbers and original versions of modified or deleted lines and line numbers of new lines. This information is kept in virtual memory and thus increases the virtual memory storage integral and the cost. The user must be aware that a copy of the entire file may reside in memory for modifications ranging over the file. It is possible to collect so much checkpoint information that no more virtual memory is available, thus prohibiting further editing. In such a situation, if the user wishes to restore a portion of the file, the NOCHECKPOINT modifier must be used to prevent further checkpointing attempts of the RESTORE or UNDO commands. To clear the checkpoint information, another CHECKPOINT command must be given.

Modifiers: None.

Conditions: SUCCESS if no error messages are generated.

CLOSE

Edit Command Description

Purpose: To force MTS to write the present form of the file to the disk so that in the event of a system failure, the chances of losing the modifications made to the file in the current editor session are reduced.

Prototype: CLOSE

Action: If the file being edited is a disk file, all of the buffers in memory are written out to the disk. The action is the same as that of the MTS WRITEBUF subroutine (see MTS Volume 3, *System Subroutine Descriptions*).

CLOSE has no effect on checkpoint information; this information remains available to the RESTORE and UNDO commands.

The editor will CLOSE the file whenever a UNLOCK, VUPDATE, or VEXIT edit command is executed. Furthermore, whenever an edit command is entered or a PF-key is executed, the editor will CLOSE the file if it has not been closed within the last 5 minutes.

Modifiers: None.

Conditions: SUCCESS if no error messages are generated.

COLUMN

Edit Command Description

Purpose: To set column limits within which editor commands will perform their actions.

Prototype:
(a) COLUMN first last
(b) COLUMN first
(c) COLUMN

Action: With prototype (a), the column pointers are set to columns "first" and "last". With prototype (b), the column pointers are set to "first" and 32767. With prototype (c), the column pointers are set to their initial values, 1 and 32767.

The column pointers are used by the ALTER, BLANK, CHANGE, MATCH, OVERLAY, SCAN, and SHIFT commands as limits for their operations within any line of the file. The column range is defined as extending from the character pointed to by the first column pointer through that pointed to by the second pointer.

Modifiers: None.

Conditions: None.

Example:

```
COLUMN 2 4
```

The column pointers are set to columns 2 and 4, respectively.

COMMENT

Edit Command Description

Purpose: To print a comment while executing an edit procedure.

Prototype:
(a) COMMENT
(b) COMMENT message
(c) * message

Action: With prototype (a), no action is taken.

With prototype (b), the message after the command name is printed if the command is executed while processing an edit procedure.

With prototype (c), the lines are treated as comments. If they are inserted into an edit procedure, they will not be printed while processing the procedure unless ECHO is ON. This provides a method of internally documenting an edit procedure. Note however that all comments are printed if the editor is executing in batch mode.

The PRINT MESSAGE command may also be used to print comments.

Modifiers: None.

Conditions: SUCCESS always.

Example:

```
COMMENT I have gone this far in the edit procedure
```

The message “I have gone this far in the edit procedure” is printed if this command is executed within an edit procedure.

CONCATENATE

Edit Command Description

Purpose: To concatenate several lines into a single line or multiple lines of a specified length.

Prototype: CONCATENATE [lpar] [LEN=n]
COMBINE [lpar] [LEN=n]

Action: All of the lines in the specified region "lpar" are concatenated (combined) into one string. The concatenated lines are deleted from the file and the resulting string is written into their place. If the length of the string is greater than "n", the string is split into sections of length "n" (or less for the last section), and these sections are written as individual lines into the file. The default value for "n" is the maximum line length for the file (usually 32767).

Note: Trimming of blanks is never done by this command regardless of the setting of the TRIM option.

COMBINE is a synonym for CONCATENATE.

Modifiers: @CH, @LEN, @LNR, @MCL, @TX, @V, @VMV, @W

Conditions: SUCCESS always.

Example:

```
CONCATENATE 2 3 LEN=80
```

The above command takes all the lines from line 2 to line 3, concatenates them into a single string, splits the string into 80-character sections, and writes the sections into the region of the file between line 2 and line 3.

COPY

Edit Command Description

Purpose: To copy a set of lines to another part of the file.

Prototype:
(a) COPY [lpar] [TO] linenumber [COPIES=n]
(b) COPY [lpar] [COPIES=n]
(c) COPY [F=FDname [COUNT=n]] [TO] linenumber [COPIES=n]
(d) COPY [lpar] [TO] F=FDname [COPIES=n]

Action: With prototype (a), a copy of all of the lines in the line-number range "lpar" is inserted at "linenumber". The first line is placed at "linenumber" unless a line already exists with that number, in which case the line is inserted immediately after it. All copied lines are inserted before the next line following "linenumber". The word TO is optional, but caution must be used since prototype (a) may be confused with prototype (b).

With prototype (b), the copied lines are inserted immediately after the current line.

With prototype (c), the lines to be copied are obtained from the file or device specified by "FDname". If the COUNT parameter is given, only "n" lines are read from "FDname". The file is read with implicit concatenation enabled (@IC); the user may override this by specifying FDname@–IC, if implicit concatenation is not desired.

With prototype (d), the lines in "lpar" are copied to the file or device specified by "FDname". The beginning line number and increment for the file are as specified with "FDname"; hence, it is possible to overwrite existing lines in the file.

The COPY command reads all of the lines to be copied before it writes them. Thus it is not possible to duplicate lines by specifying overlapping regions. The changes performed by writing to another file or device may not be reversed by the RESTORE and UNDO commands.

If the COPIES keyword is specified, "n" copies of the lines specified will be inserted, one copy immediately after the other.

Modifiers: @CH, @LEN, @LNR, @MCL, @TR, @TX, @V, @VMV, @W

Conditions: SUCCESS if the region is copied. EOF if a read beyond the end of the file is attempted.

Examples:

```
COPY 123 TO 345
```

In the above example, a copy of line 123 is inserted at line 345.

```
COPY 123 345
```

In the above example, a copy of lines 123 through 345 is inserted between the current line and the next line. Note the difference between this example and the previous example.

```
COPY@NV 1 10 TO 44
```

In the above example, a copy of lines 1 through 10 is inserted at line 44.

```
COPY F=FILE1(1,100) TO *L @NV
```

In the above example, a copy of lines 1 through 100 of the file FILE1 is inserted after the last line in the file. Verification of the copy operation is suppressed.

```
COPY /FILE TO F=*PRINT*
```

In the above example, the entire file is copied to *PRINT* for printing on a line printer.

```
COPY 1 10 TO 15 COPIES=3
```

In the above example, three copies of lines 1 through 10 will be inserted after line 15.

```
COPY 1 10 TO F=FILE2(*L+1)
```

In the above example, lines 1 through 10 are copied to the end of the file FILE2. Note that if the FDname had been specified as F=FILE2(*L), the last line of the file FILE2 would have been overwritten by line 1 of the edit file.

CORRECT

Edit Command Description

Purpose: To correct parts of lines.

Prototype:
(a) CORRECT [lpar] strexp
(b) CORRECT [lpar]

Action: With prototype (a), each line in the given line-number range "lpar" is corrected according to the correction string specified by the string expression "strexp". The PATTERNS option must be ON if "strexp" is more complex than a simple string.

With prototype (b), each line in the given line-number range "lpar" is printed on the terminal, and the user is prompted for the correction string. The correction string is used to correct the line. The corrected line is then printed on the terminal for further corrections. When a null string (i.e., an input line of length zero) is entered, the next line in the line-number range is printed and the process repeated for that line. Entering an end-of-file is the same as entering a null string.

The correction string allows deletion, insertion, replacement, splitting, blanking, replacement with editor pad characters, replacement with editor fill characters, and truncation. The CORRECT single-character commands D, I, R, S, B, P, F, and T are used in the correction string to specify these operations. Each of these commands is described below. The fill character (defaults to blank) is used in the correction string to indicate that nothing is to be done to the corresponding character in the line being corrected. The remainder of the correction string following an I, R, or T CORRECT single-character command is used as text by the command, and is not scanned for further commands.

Setting the editor fill character to B, D, F, I, L, P, R, S, T, or U (or a lowercase equivalent) is not recommended. In a correction string for prototype (b) read in batch mode, the last character that is not a fill character is taken to be the end of the correction string (see the description of I, R, and T below). A line containing only fill characters is therefore treated as a null line in batch.

The correction string is examined one character at a time. The action specified by this CORRECT single-character command is applied to the current character in the line being corrected. The next character in the line being corrected now becomes the current character, and the process is repeated until no more characters remain in the correction string.

The CORRECT command only works within the current column range. If a CORRECT command is entered outside the current column range, an error message is printed. Expansion of a line within the current column range may cause truncation on the right within the current column range. Contraction within the current column range causes one editor pad character to be appended on the right within the current column range for each character removed.

When an error is detected, the line remains uncorrected by the current correction string. With prototype (a), execution of the CORRECT command ceases immediately. With prototype (b), the current line is printed again and the user is prompted for a new correction string.

The actions corresponding to each of the CORRECT single-character commands are as follows:

fill No change is made to the current character in the line.

B The current character in the line is changed to a blank.

D The current character in the line is deleted, and any text to the right (within the current column range) is shifted left one position. An editor pad character is placed in the vacated position at the right-hand side of the current column range. Note: This pad character is only added if there is at least one character to the right of the current column range in the original line.

F The current character in the line is changed to the editor fill character.

I All text in the correction string following the I is inserted prior to the current character in the line. The current character and all characters to its right in the line are shifted right as required. Characters shifted out of the current column range are lost. Note: When using prototype (b) in batch mode, the last character in the correction string that is not an editor fill character is taken to be the end of the string.

L The current character in the line is translated to lowercase.

P The current character in the line is changed to the editor pad character.

R All text in the correction string following the R is used to replace the current and following characters in the line, on a one-for-one basis. This is done until the end of text in the string. If the replacement text extends beyond the current column range, a message is printed and the line is unchanged. If the replacement starts beyond the end of text in the line, the line is extended as required with pad characters. Note: When using prototype (b) in batch mode, the last character in the correction string that is not an editor fill character is taken to be the end of the string.

S All text in the line beginning with the character corresponding to the S is deleted from the current line and is placed on a new line immediately following the current line (the line is split).

T All text in the line beginning with the character corresponding to the T is deleted (the line is truncated). Then all text in the correction string following the T is inserted where the truncation began. Note:

When using prototype (b) in batch mode, the last character in the correction string that is not an editor fill character is taken to be the end of the string.

U The current character in the line is translated to uppercase.

Modifiers: @CH, @COL, @LEN, @LNR, @MCL, @OPL, @TR, @TX, @V, @W, @X

Conditions: SUCCESS if the region is corrected. EOF if a read beyond end-of-file is attempted.

Examples: In the following examples, it is assumed that the editor fill and pad characters are both blank, and the current column range is at its default value of 1 through 32767.

```
CORRECT 10 'DDDF'
```

This deletes the first three characters in the current column range of line 10, and changes the fourth character to the fill character (i.e., a blank for this example).

```
CORRECT 10 20 'PPPP'
```

This changes the first four characters in the current column range to pad characters (i.e., a blank for this example) in lines 10 through 20, inclusive.

```
CORRECT *F 300 'B B B DBD D'
```

This changes the first, third, fifth, and eighth characters in the current column range to blanks, and deletes the seventh, ninth, and eleventh characters, in all lines between the first line and line 300, inclusive. All other characters remain unchanged.

```
CORRECT@X /FILE 'C4C440D9B0B0B0B0'
```

This, in hexadecimal mode, deletes the first two characters in the current column range for all lines in the file, leaves the next character unchanged, and replaces the next four characters with the character represented by hexadecimal 'B0'.

```
CORRECT 100#
:THIS LINE CONTAINSSERORS
?                   B IR#
:THIS LINE CONTAINS ERRORS
?                   I NO#
:THIS LINE CONTAINS NO ERRORS
?#
: 100 THIS LINE CONTAINS NO ERRORS
:
```

In the above example, the position at which the user enters the RETURN (end-of-line) character is significant. The “#” indicates where the user enters RETURN. The user does not type the “#” character.

Notes:

The initial editor fill and pad characters are blank. They may be changed with the FILLCHAR and PADCHAR options of the SET edit command.

The column range may be set with the COLUMN edit command.

In @X mode, the CORRECT single-character commands and the fill character must also be in hexadecimal.

CORRECTing long lines on ASCII-type display terminals using the prototype (b) CORRECT command may be impossible (depending where in the line the correction is to be made).

COUNT

Edit Command Description

Purpose: To count the number of lines between given line numbers.

Prototype: COUNT [lpar]

Action: The number of lines contained in the given line-number range "lpar" is printed.

Modifiers: None.

Conditions: SUCCESS if no error messages are generated. EOF if a read beyond the end of the file is attempted.

Example:

```
COUNT /FILE
```

The above command counts the number of lines in the entire file.

DELETE

Edit Command Description

Purpose: To remove lines from the file.

Prototype: DELETE [lpar] [OK]

Action: The lines in the given line-number range "lpar" are deleted from the file. The line numbers must be specified in ascending order. If the edit file is a sequential file, the lines cannot be deleted; however, the lines will be filled with blanks so that the data is effectively deleted. The message "Line padded" will be printed for each such line.

The line numbers delimiting the range are checked to ensure that they do not point to lines outside the actual bounds of the file. If such is the case, confirmation is requested. A reply of OK allows the command to proceed. Alternatively, the OK may be appended to the command. If the DV (DELETEVERIFY) option is not set to OFF, confirmation is required if more than the number of records specified by the DV value are to be deleted. The DV value is initially 5.

If the VERIFY option is ON (the default), a message giving the number of lines deleted is printed.

Modifiers: @CH, @MCL, @V

Conditions: SUCCESS if at least one line is deleted. EOF if a read beyond the end of the file is attempted.

Examples:

```
DELETE 234.3
```

The above command deletes line 234.3 from the file.

```
DELETE 300 *L OK
```

The above command deletes all lines from line 300 to the end of the file. Confirmation is given with the command.

EDIT

Edit Command Description

Purpose: To select another file to be edited.

Prototype:
(a) EDIT filename [[:]edit-command]
(b) EDIT {PROCEDURE | XEC} !name [[:]edit-command]
(c) EDIT VPF pfname [[:]edit-command]
(d) EDIT MACRO macro-name [[:]edit-command]

Action: With prototype (a), "filename" becomes the new file to be edited.

With prototype (b), the edit procedure "!name" becomes the "file" to be edited. The full facilities of the editor are available when editing an edit procedure.

With prototype (c), the visual-mode program-function cluster becomes the file to be edited. See the description of the VPF visual-mode command for details of using program-function clusters.

With prototype (d), the MTS command macro "macro-name" becomes the "file" to be edited. Note: This is valid only if the command $SET MACROS=ON has been given and the macro has been defined.

An optional colon and edit command may be appended to EDIT command. This command will be executed immediately after the edit file has been changed.

The current line (*) is set to the first line of the new edit file.

Any checkpoint information that has been saved for the previous file is discarded, and new checkpoint information is accumulated for the new file being edited. See the CHECKPOINT, RESTORE, and UNDO commands for further description of the checkpoint facility.

Modifiers: None.

Conditions: SUCCESS if no error messages are generated.

Examples:

```
EDIT MYFILE
```

In the above example, the file MYFILE becomes the edit file.

```
EDIT PROCEDURE !PROCB
```

In the above example, the edit procedure !PROCB becomes the edit file.

```
EDIT MACRO SEND :PRINT /FILE
```

The definition of the MTS macro named SEND is printed.

EXECUTE

Edit Command Description

Purpose: To execute an edit procedure, an edit command, or the visual-mode work area.

Prototype:
(a) EXECUTE !name n
(b) EXECUTE !name
(c) EXECUTE 'edit-command'
(d) EXECUTE ED_WORK

Action: With prototype (a), the edit procedure "!name" is executed "n" times.

With prototype (b), the procedure is executed once.

With prototype (c), the edit command string is executed.

With prototype (d), the contents of the visual-mode work area is executed.

Prototypes (c) and (d) are most useful when used within an edit procedure. However, edit-command messages such as "Search failed" and "File does not exist" will *not* be printed for commands executed from the procedure. The SUCCESS and FAILURE switches may be used to determine the status of the command executed (see the section "Edit Procedures" above).

Modifiers: None.

Conditions: SUCCESS if no error messages are generated.

Examples:

```
EXECUTE !PROCB 10
```

In the above example, the edit procedure !PROCB is executed 10 times.

```
EXECUTE 'SCAN 1 10 "ABC"'
```

In the above example, the SCAN command is executed.

EXPLAIN

Edit Command Description

Purpose: To describe editor commands during an editor session.

Prototype:
(a) EXPLAIN
(b) EXPLAIN command

Action: The EXPLAIN command provides information about the syntax and operation of editor commands. Prototype (a) provides general editor information. Prototype (b) provides information about the specified command.

Modifiers: None.

Conditions: SUCCESS if an explanation is given.

Example:

```
EXPLAIN ALTER
```

The above command provides information about the ALTER command.

GOTO

Edit Command Description

Purpose: To alter the flow of execution in an edit procedure.

Prototype:
(a) GOTO label
(b) GOTO label [ON] condition

where "condition" is any of SUCCESS, S, FAILURE, F, ENDOFFILE, EOF, or E.

Action: The GOTO command may used within an edit procedure to alter the sequential execution of commands. Prototype (a) specifies an unconditional transfer to the edit procedure command with the given label. Prototype (b) specifies a transfer if the specified condition is satisfied.

The label may be one to eight nonblank characters. There are two predefined edit procedure labels, COUNT and END. COUNT decrements the execution count by one and continues execution from the beginning of the edit procedure if the count is still positive. END terminates the edit procedure immediately, regardless of the execution count given.

Modifiers: None.

Examples:

```
GOTO LABEL1
```

The above command transfers execution to the label LABEL1.

```
GOTO END ON EOF
```

The above command terminates execution when an end-of-file condition occurs.

INSERT

Edit Command Description

Purpose: To insert new lines into the file.

Prototype:
(a) INSERT [linenumber] strexp
(b) INSERT [linenumber]
(c) INSERT [linenumber] INCREMENT=increment

Action: With prototype (a), the string specified by the string expression "strexp" is inserted as a new line in the file. Its line number becomes "linenumber" unless a line already exists with that number, in which case the new line is inserted immediately after "linenumber". If no space is available, the comment

```
No room left for insertion
```

is printed. The PATTERNS option must be ON if "strexp" is more complex than a simple string.

With prototype (b), the user is prompted for lines to be inserted at or after "linenumber". This is called "quick insertion mode." The user may enter an arbitrary number of lines in this manner. If the user attempts to enter more lines than will fit, the above error comment is printed. To leave quick insertion mode, a null line or an end-of-file may be given. Implicit concatenation is ignored in quick insertion mode. The inserted lines are spaced evenly between "linenumber" and the next line following it. The editor accomplishes this by periodically renumbering the lines within the insert space. The user need not be concerned with this automatic renumbering except when the access to the edit file is only WRITE (not WRITE/CHANGE). In this case, the user should use prototype (c).

With prototype (c), the user is prompted for lines to be inserted at or after "linenumber". This is an alternate form of prototype (b). The major difference between this form and prototype (b) is that the increment between lines is explicit, and the user is prompted for lines using the line number which will be assigned to the line being inserted. The lines inserted will start at "linenumber", if it is not already occupied, and continue with the increment given by "increment". If "linenumber" is occupied, the first line will be inserted at "linenumber+increment" and insertion will continue from there. Otherwise, prototype (c) works in exactly the same way as prototype (b), except that the error message given when no more insertions are possible is:

```
No more room for insertion with given increment
```

The line-number prompting may be disabled by specifying the modifier @NPLN or setting the PLN (PAGELINENUMBER) option to OFF. The current line pointer is set at the last line inserted.

Modifiers: @CH, @LEN, @LNR, @PLN, @MACRO, @MCL, @TR, @TX, @V, @VMV, @W, @X

Conditions: SUCCESS if at least on line is inserted.

Examples:

```
INSERT 123 'data'
```

In the above example, the line “data” is inserted at line 123 in the file.

```
INSERT 12
data 1
data 2
data 3
$ENDFILE
```

In the above example, lines “data 1”, “data 2”, and “data 3” are inserted at line 12 of the file. “data 3” is the current line (*).

```
INSERT 12 I=.01
 12    _ data 1
 12.01 _ data 2
 12.02 _ data 3
 12.03 _ $ENDFILE
```

This example is the same as the previous example except that the lines to be inserted are explicitly assigned to line numbers 12, 12.01, and 12.02.

```
INSERT 4 ASCII('Hi There!')
```

This example inserts the ASCII character equivalent of the string “Hi There!” at line 4.

JUSTIFY

Edit Command Description

Purpose: To right- or left-justify lines.

Prototype: JUSTIFY [lrange] [direction lmar rmar]

where

"lrange" is a contiguous line range specifying the text to be justified. The line range must be in the forward order.

"direction" is a keyword specifying the method of justification to be used. The values allowed for "direction" are LEFT, RIGHT, and BOTH.

"lmar" is the column to be used as the left margin for the justification.

"rmar" is the column to be used as the right margin for the justification.

Action: The line range specified by "lrange" will be justified to the columns "lmar" and "rmar" on the basis of the keyword "direction". LEFT means justify to the left margin (ragged-right), RIGHT means justify to the right margin (ragged-left), and BOTH means justify to both margins (block).

"direction", "lmar", and "rmar" must all be specified if any are specified. If all three are omitted, the previous values (i.e., the values appearing on the previous JUSTIFY command) will be used. If there was no previous JUSTIFY command, the command will be aborted. "lrange" defaults to the current line (*).

At the start of the justification process, the lines in "lrange" are subdivided into individual items, with each item being the string of characters up to the next editor fill character. The editor fill character is normally a blank, but may be redefined by the editor FILLCHAR option. For a block of written text, each item is synonymous with a word. After the text has been subdivided into individual items, the text is then rewritten into the line range specified by "lrange" according to the justification direction and margin specifications, replacing the previous unjustified lines. If extra lines are needed to accommodate the justified text, they will be inserted after the last line in the line-number range.

If the @OPL modifier is not specified (the default), the justified lines are packed as tightly as possible. Certain items may be moved up or down from their original lines to the end or beginning of a different line.

If @OPL is specified, the lines in "lrange" are processed individually, i.e., items from one line are not merged with items of another line. Thus, the justification is done on a line-by-line basis, rather than over a block of lines. If the original line is too long for the justification region, new lines will be created for the excess items.

The space to the left of the left margin “lmar” and the space between individual items in each rewritten line is filled with the editor pad character. The editor pad character is normally a blank, but may be redefined by the editor PADCHAR option. If LEFT or RIGHT is specified, only one pad character is inserted between each item. If BOTH is specified, one or more pad characters may be inserted between each item.

If a blank line is encountered in “lrange”, it is treated as a request to interrupt the current justification process and justify the accumulated items. A blank line is written after the justified lines. Justification then resumes with the next nonblank line. This provides a method of paragraphing by inserting blank lines where paragraph endings are desired.

If an item longer than the justification region is encountered, it will be rewritten as a single line without truncation (thus extending beyond the right margin “rmar”).

An error message is printed if “lmar” > “rmar”, or if either is less than 1 or greater than the maximum record length of the edit file. An error message is also printed if the resultant justified output will not fit in the space occupied by “lrange”. This may occur if the justification area as defined by “lmar” and “rmar” is substantially less than the line lengths of the original text.

Modifiers: @CH, @LEN, @LNR, @MCL, @OPL, @TR, @TX, @V, @VMV, @W

Conditions: SUCCESS if the region is justified.

Example:

```
JUSTIFY /FILE BOTH 1 65
```

In the above example, the contents of the entire edit file is justified between columns 1 and 65.

LET

Edit Command Description

Purpose: To create or redefine an editor variable.

Prototype: LET variable=value

Action: The LET command creates or redefines an editor variable. If the variable does not exist, it is created and assigned the value "value"; otherwise, the current value is replaced by "value".

Editor variables are useful in the construction of editor pattern structures to assign matched substrings, or are used to save complete pattern structures.

The variable name may consist of 1 to 255 characters, the first of which must be alphabetic (A-Z) while subsequent characters may be alphanumeric (A-Z, 0-9, and _). The value of the variable may be any one of the following data types:

(1) an integer, e.g., 5, –4

(2) a line number, e.g., *F, 1.5

(3) a string or string expression, e.g., "STRING1", 'string2', 'ABC' VARIABLE 'DEF'

(4) a pattern

A variable is not declared to be of a particular data type; the context in which the variable is used determines its type.

The primary use of editor variables is with pattern matching (see the section "Pattern Matching").

The editor has several predefined variables. These are listed in the section "Pattern Matching."

Modifiers: None.

Conditions: SUCCESS if no error messages are generated.

Example:

```
LET ALPHA='abcdef'
```

The above example creates the variable ALPHA with the string value "abcdef".

```
LET ED_VRULER=''
```

The above example sets the visual-mode ruler to the null string. The ruler area will now contain the default status line as specified by the variable ED_STATUS_DEF.

LINE

Edit Command Description

Purpose: To make a particular line the current line.

Prototype:
(a) LINE [linenumber]
(b) LINE linenumber increment
(c) LINE linenumber COUNT=n

Action: With prototype (a), the line number specified becomes the current line.

With prototype (b), the values of "linenumber" and "increment" are added to form a new line number, which becomes the current line. "increment" is an absolute integer value that is added to "linenumber".

With prototype (c), "n" is a displacement (counting the number of existing lines) from "linenumber". The primary use of prototype (c) is to define values for program-function keys in visual mode using the line numbers *VT and *VB, which represent the top and bottom of the visual screen, respectively. For example, the default definitions for several of the program-function keys are:

PF1 and PF13: LINE *VT COUNT=–10
PF4 and PF16: LINE *VT COUNT=10
PF7 and PF19: LINE *VT COUNT=1

Modifiers: @LEN, @LNR, @TX, @V, @VMV, @W, @X

Conditions: SUCCESS if the line is in the file. EOF if the line is not in the file.

Examples:

```
LINE 123
```

In the above example, line 123 becomes the current line.

```
LINE *L -10
```

In the above example, the tenth line from the end of the file becomes the current line.

```
LINE *VB COUNT=-5
```

In the above example, the visual-mode screen will be positioned so that the new top line is the fifth line from the bottom of the current screen.

MATCH

Edit Command Description

Purpose: To search for a line with a particular character string starting in a particular column.

Prototype:
(a) MATCH [lpar] 'string'
(b) MATCH [lpar]
(c) MATCH [lpar] pattern
(d) MATCH VARIABLE=vname pattern

Action: With prototype (a), the editor searches the given line-number range "lpar" for a line that matches "string" starting at the left column position. Positions in the string in which the current fill character occurs will match any character in the corresponding position in the column range.

With prototype (b), the most recent string given in a previous MATCH command is used.

With prototype (c), a pattern structure is used to search the line-number range "lpar" for the subject string. The fill character is ignored when using prototype (c). The pattern match is done in anchored mode.

With prototype (d), a pattern structure is used to search the value of the variable "vname" for the subject string. See the section "Pattern Matching" for details on using patterns with the editor.

If no line-number range is specified, the command searches from the current line to the last line in the file (* *L).

The current line is set to the last line with a successful match. The top of the visual-mode screen (*VT) is set to the first line with a successful match.

The initial fill character is the blank. It may be changed with the editor FILLCHAR option. The left column position may be changed by the COLUMN command.

The line or lines that are matched by the MATCH command may be referenced collectively by the /MATCH region.

Modifiers: @A, @AC, @COL, @LEN, @LNR, @MCL, @NOT, @OPL, @PC, @TX, @V, @VMV, @W, @X

Conditions: SUCCESS if a successful match was made. EOF if the search extends beyond the end of the file.

Examples:

```
MATCH@ALL /FILE 'A  D'
```

In the above example, the entire file is searched for all occurrences of the string "AxxD", where A begins in the leftmost column. The second and

third characters are arbitrary.

```
MATCH@ALL /TAIL ED_09 'B'
COPY /MATCH TO F=FILE1
```

In the above example, the edit file is searched from the next line after the current line to the end of the file for two-character strings that begin with a digit and are followed by “B”. All of the lines successfully matched are copied into the file FILE1 by using the COPY command with the /MATCH region.

MCMD

Edit Command Description

Purpose: To execute an MTS command.

Prototype: MCMD command

Action: The MTS command specified is executed and control is returned to edit mode.

Modifiers: None.

Conditions: SUCCESS if the command succeeded.

Example:

```
MCMD $DISPLAY COST
```

In the above example, the $DISPLAY COST command is executed to display the current cost of the terminal session.

MOVE

Edit Command Description

Purpose: To move a set of lines to another part of the file or to another file.

Prototype:
(a) MOVE [lpar] [TO] linenumber
(b) MOVE [lpar]
(c) MOVE [lpar] [TO] F=FDname

Action: With prototype (a), all of the lines in the line-number range "lpar" are inserted at "linenumber" and the lines in the original region "lpar" are deleted. The first line is inserted at "linenumber" unless a line already exists with that number, in which case the first line is inserted immediately after it. All of the lines are inserted before the next line following it. The word TO is optional, but caution must be used since prototype (a) may be confused with prototype (b).

With prototype (b), the moved lines are inserted immediately after the current line.

With prototype (c), the lines in "lpar" are moved to the FDname specified.

The MOVE command reads all of the lines to be moved before it writes them; thus, it is not possible to cause loops by specifying overlapping regions. The changes performed by writing to another file or device may not be reversed by the UNDO or RESTORE commands.

Changes to the external file cannot be undone. However, the UNDO command will place lines back into the edit file.

Modifiers: @CH, @LEN, @LNR, @MCL, @TR, @TX, @V, @VMV, @W, @X

Conditions: SUCCESS if the region is moved. EOF if the line-number range extends beyond the end of the file.

Examples:

```
MOVE 6 TO 2
```

In the above example, line 6 is inserted at line 2.

```
MOVE 4 9 *L
```

In the above example, lines 4 through 9 are appended to the end of the file.

MTS

Edit Command Description

Purpose: To return to the caller (normally MTS command mode).

Prototype:
(a) MTS
(b) MTS command

Action: With prototype (a), control returns to the caller (normally MTS command mode).

With prototype (b), control returns to the caller (normally MTS command mode) and the command specified is executed as an MTS command.

The edit session may be later resumed by issuing the $EDIT command.

If a patterned file name was specified on the EDIT command, the next file in the pattern sequence (if any) is edited.

Modifiers: None.

Conditions: SUCCESS always for prototype (a); SUCCESS for prototype (b) if the command succeeded.

Example:

```
MTS $DISPLAY COST
```

In the above example, control returns to MTS command mode and the cost of the terminal session is displayed.

OVERLAY

Edit Command Description

Purpose: To overlay certain portions of a line with other data.

Prototype:
(a) OVERLAY [lpar] strexp
(b) OVERLAY [lpar]

Action: With prototype (a), each line in the specified line-number range "lpar" is overlaid with the string specified by the string expression "strexp". The PATTERNS option must be ON if "strexp" is more complex than a simple string.

With prototype (b), each line is printed on the terminal and the user is prompted for a string to be overlaid on that line. If the user enters an end-of-file, the command is terminated and the last line printed is not modified.

Positions in the line corresponding to a nonfill character in the string are replaced by the corresponding character in the string; positions corresponding to a fill character are not changed. The overlaying is done with respect to the current column range.

If the columns being overlaid extend beyond the end of the line, pad characters are inserted from the end of the line to the right column pointer.

The initial fill character is the blank. It may be changed with the editor FILLCHAR option. The current column range may be changed by the COLUMN command.

Modifiers: @CH, @COL, @LEN, @LNR, @MCL, @TR @TX, @V, @VMV, @W, @X

Conditions: SUCCESS if at least one line is overlayed. EOF if a read beyond the end of the file is attempted.

Examples:

```
OVERLAY 123 '0000'
```

In the above example, the first four characters of the current column range (as set by the COLUMN command) in line 123 are overlaid with zeros.

```
OVERLAY 123 '   ABC'
```

In the above example, the fourth through sixth characters of the current column range (as set by the COLUMN command) in line 123 are replaced by the string 'ABC'; the first three characters are unchanged.

```
OVERLAY 19 COL=35 'comment'
```

In the above example, the seven characters in line 19 starting at column 35 are replaced by the string "comment". If line 19 is less than 34 characters in length, it will be padded to 34 characters with the pad character, and the string will be appended at column 35.

PRINT

Edit Command Description

Purpose: To print lines of the file or information about editor variables and options.

Prototype:
(a) PRINT [lpar]
(b) PRINT [lpar] CHECKPOINT
(c) PRINT COLUMNS
(d) PRINT FILENAME
(e) PRINT {MESSAGE | MSG} strexp
(f) PRINT !name
(g) PRINT {PROCEDURES | XECS}
(h) PRINT REGIONS
(i) PRINT SAVES
(j) PRINT VARS
(k) PRINT VARIABLE name
(l) PRINT VMSG strexp
(m) PRINT VPFS
(n) PRINT WINDOWS

Action: With prototype (a), all of the lines in the given line-number range "lpar" are printed.

With prototype (b), all of the lines in the given line-number range that have been changed since the last CHECKPOINT command are printed.

PRINT COLUMNS prints the values of the column pointers.

PRINT FILENAME prints the name of the file being edited.

PRINT MESSAGE or PRINT MSG prints the message given by "strexp". "strexp" may be a character string, a variable, or a combination of character strings and variables. If the command is executed in visual mode, the message will be intensified and placed in the ruler area (immediately below the work area); the VMV modifier must be used to write the message into the conversation buffer.

PRINT !name prints the commands of the edit procedure "!name".

PRINT PROCEDURES or PRINT XECS prints the names of the currently defined edit procedures.

PRINT REGIONS prints the names of the currently defined regions. The "/name" command may be used to print region values.

PRINT SAVES prints the names of all files that have been SAVEd along with their associated save names.

PRINT VARIABLE name prints the name, type, length, and value of the editor variable "name".

PRINT VARS prints the names of all editor variables that are currently defined.

PRINT VMSG prints the message given by "strexp" in the visual-mode error comment field below the vruler/status line. This allows, for example, a command macro to issue messages that will appear at the bottom of the visual screen and then enter visual mode.

PRINT VPFS prints the current definitions of all the visual-mode program-function keys.

PRINT WINDOWS prints the values of the window pointers.

Modifiers: @LEN, @LNR, @MCL, @TX, @VMV, @W, @X

Conditions: SUCCESS if the information is printed. EOF if a read beyond the end of the file is attempted.

Examples:

```
PRINT 123
```

In the above example, line 123 is printed.

```
PRINT /FILE CHECKPOINT
```

In the above example, all lines in the entire file that have been changed since the last CHECKPOINT command was issued are printed.

```
PRINT MSG "Did you know that this is a " WHAT
```

In the above example, the message "Did you know that this is a test" is printed assuming that the value of variable WHAT is "test".

PROCEDURE

Edit Command Description

Purpose: To create an edit procedure.

Prototype: PROCEDURE !name
XEC !name

Action: An edit procedure with the name "!name" is created, or emptied if it already exists.

The user is prompted to enter the command list. All edit commands entered become part of the command list. The command list is terminated by entering the pseudocommand END, a null line, or an end-of-file.

XEC is an alternate command name for PROCEDURE.

See the section "Edit Procedures" for further details.

Modifiers: @MACRO

If the @MACRO is specified, lines read during the procedure definition are read with the MTS FDname @MA CRO@MFR applied (any line beginning with the MTS command macro flag ">" will be processed by the MTS macro processor).

Conditions: None.

Examples:

```
PROCEDURE !PROC1
LINE@NV *F
LOOP: SCAN 'TENTATIVE'@NV
GOTO END ON FAILURE
DELETE@NV
GOTO LOOP
END
```

In the above example, the edit procedure !PROC1 is created which deletes from the entire file all lines containing the string "TENTATIVE".

```
PROCEDURE@MACRO !SETUP
>Set_Edit
END
```

In the above example, the MTS command macro named Set_Edit is used to define the edit procedure !SETUP.

REGION

Edit Command Description

Purpose: To define names which refer to certain sets of lines within the file.

Prototype: REGION /name lpar

Action: The REGION command defines the region name "/name" to be the line-number range "lpar".

"lpar" may consist of one or more previously defined region names. In this case, the region are concatenated to form the new region. Overlap may occur, e.g., the commands

```
REGION /REG1 1,2
REGION /REG2 2,3
REGION /REG3 /REG1,/REG2
```

will define a region REG3 which contains two occurrences of line 2.

The line numbers defining a region can be displayed by using the "/name" command, e.g.,

```
/REG1
```

Modifiers: None.

Conditions: SUCCESS if the region is created.

Examples:

```
REGION /REG1 100 200
```

The region name /REG1 is defined as the line-number range 100 to 200.

```
REGION /REG2 100,200
```

/REG2 is defined as a region consisting of two lines, 100 and 200.

REMEMBER

Edit Command Description

Purpose:	To restore a previous state of the edit session.
Prototype:	REMEMBER [name]
Action:	Information saved under "name" by the SAVE command is restored causing the editor to revert to that state. If "name" is omitted, the save information named by the current file name is used. See also the SET REMEMBER, SET SAVE, and SET SAVEREMEMBER descriptions.
Modifiers:	None
Conditions:	None

RENUMBER

Edit Command Description

Purpose: To renumber lines in the file.

Prototype:
(a) RENUMBER [first last beg inc]
(b) RENUMBER beg inc
(c) RENUMBER first last beg
(d) RENUMBER beg

Action: The lines in the region "first" through "last" are renumbered to have line numbers beginning with "beg" and successively incremented by "inc".

The defaults are: first=*F, last=*L, beg=1, and inc=1.

Modifiers: None.

Conditions: SUCCESS if the region is renumbered.

Examples:

```
RENUMBER 1 100 1
```

The above command renumbers lines 1 through 100 of the file beginning with 1 and incremented by 1.

```
RENUMBER 1000 10
```

The above command renumbers the entire file beginning with 1000 and incremented by 10.

```
RENUMBER 1
```

The above command renumbers the entire file beginning with 1.

REPLACE

Edit Command Description

Purpose: To replace lines in the file.

Prototype:
(a) REPLACE [lpar] strexp
(b) REPLACE [lpar]

Action: With prototype (a), each line within the line-number range "lpar" is replaced by the string specified by the string expression "strexp". The PATTERNS option must be ON if "strexp" is more complex than a simple string.

With prototype (b), each line is printed on the terminal and the user is prompted for a string to be used as a replacement for that line. If the user enters an end-of-file, the command is terminated and the last line printed is not modified. If the user enters a null line, the line is replaced by a null line; i.e., it is deleted.

Modifiers: @CH, @LEN, @LNR, @MCL, @TR, @TX, @V, @VMV, @W, @X

Conditions: SUCCESS if at least one line is replaced. EOF if a read beyond the end of the file is attempted.

Examples:

```
REPLACE 123 'newline'
```

In the above example, line 123 is replaced by the string "newline".

```
REPLACE 123
newline
```

In the above example, line 123 is replaced by the string "newline".

```
REPLACE 1 10 RPAD('B' ED_09,16) '*'
```

The above example replaces lines 1 through 10 with lines containing the string "B0123456789 ".

RESTORE

Edit Command Description

Purpose: To restore the status of the file to the point of the last CHECKPOINT command.

Prototype: RESTORE [lpar]

Action: The RESTORE command restores the portion of the file specified by "lpar" to its condition at the last CHECKPOINT ON command, if checkpointing is enabled. The specified portion of the file is restored to its status before the last command that changed it, if CHECKPOINT is OFF (the default). The RESTORE command has no effect if CHECKPOINT is NEVER.

If "lpar" is omitted, only the current line is restored. To restore the entire file, /FILE should be specified. The COUNT keyword in "lpar" is not allowed with this command.

See the section "Checkpoint/Restore Facility" for further details.

Modifiers: @CH, @LEN, @LNR, @TR, @TX, @V, @VMV, @W, @X

Conditions: None.

Examples:

```
RESTORE 123
```

In the above example, line 123 is restored to its status at the last CHECKPOINT command.

```
RESTORE /FILE
```

In the above example, the entire file is restored to its status at the last CHECKPOINT command.

RETURN

Edit Command Description

Purpose:	To return to the caller (normally MTS command mode).
Prototype:	RETURN
Action:	Control returns to the caller (normally MTS command mode). The edit session may be later resumed by issuing the $EDIT command. If a patterned file name was specified on the EDIT command, the next file in the pattern sequence (if any) is edited. Note: Many terminals have a RETURN key; pressing this key is *not* equivalent to typing the characters “RETURN”.
Modifiers:	None.
Conditions:	SUCCESS always.

SAVE

Edit Command Description

Purpose: To save information about the current state of the edit session.

Prototype: SAVE [name]

Action: Information about the current state of the edit session (*VT, current line, insertion point and count, etc.) is saved under "name".

If "name" is omitted, the saved information is named with the current file name.

Note that this command does not save any text from the file. The CHECKPOINT command must be used for this purpose.

Any regions that are defined and/or any checkpoint information is lost when a new file is edited, even though the current file may have been saved.

See also the SET REMEMBER, SET SAVE, and SET SAVEREMEMBER descriptions.

Modifiers: None

Conditions: None

SCAN

Edit Command Description

Purpose: To search for a line containing a particular character string.

Prototype:
(a) SCAN [lpar] 'string'
(b) SCAN [lpar]
(c) SCAN [lpar] pattern
(d) SCAN VARIABLE=vname pattern

Action: With prototype (a), each line in the given line-number range "lpar" is searched for the first occurrence of the character string "string".

With prototype (b), each line is searched for the first occurrence of the character string given on the last SCAN command that specified a string.

With prototype (c), a pattern structure is used to search the line-number range "lpar" for the subject string. The pattern match is done in unanchored mode.

With prototype (d), a pattern structure is used to search the value of the variable "vname" for the subject string. See the section "Pattern Matching" for details on using patterns with the editor.

If no line-number range is specified, the command searches from the current line to the last line of the file (* *L).

The current line is set to the last line in which there was a successful match. The top of the visual-mode screen (*VT) is set to the first line with a successful match.

The line or lines that are matched by the SCAN command may be referenced collectively by the /SCAN region.

Modifiers: @A, @AC, @COL, @LEN, @LNR, @MCL, @NOT, @OPL, @PC, @RTL, @TX, @V, @VMV, @W, @X

Conditions: SUCCESS if the search is successful. EOF if a read beyond the end of the file is attempted.

Examples:

```
SCAN /FILE 'TIGER'
```

In the above example, the entire file is searched for the first occurrence of the string "TIGER".

```
SCAN@ALL 1 100 'B' LEN(2) 'B'
MOVE /SCAN TO *L
```

In the above example, lines 1 through 100 are scanned for the pattern "BxxB", where "xx" is arbitrary. All the lines matching that pattern are moved to the end of the file by using the MOVE command with the /SCAN

region.

SET

Edit Command Description

Purpose: To change the settings of global editor options.

Prototype: SET keyword=value **...**

Action: The SET command is used to alter the values of a number of global editor options that govern the editing process.

{AC | ANYCASE}={ON | OFF} Default: OFF

If the ANYCASE option is ON, the ALTER, CHANGE, MATCH, and SCAN commands will match any occurrence of alphabetic characters in the pattern regardless of case. If the ANYCASE option is OFF, these commands will match only the exact alphabetic characters as specified in the pattern.

ALL={ON | OFF} Default: OFF

If the ALL option is ON, the ALTER, CHANGE, MATCH, and SCAN commands process all occurrences of the pattern throughout the specified range of the file. If the ALL option is OFF, these commands process only the first occurrence of the pattern.

BASE=[n] Default: 0.0

The line number to be used as line 0.0 in edit commands and editor output. The base does not affect MINLINE or MAXLINE. If "n" is omitted, the base is reset to the default.

CHECKPOINT={ON | OFF} Default: ON

If the CHECKPOINT option is ON and if the checkpoint buffer has been opened by the CHECKPOINT command, the results of edit commands are recorded in the buffer so that they may be undone, if necessary. See the section "Checkpoint/Restore Facility."

COLUMN={ON | OFF} Default: ON

If the COLUMN option is ON, the action of the edit command applies only to the column range set by the COLUMN command. See the description of the COLUMN command.

{DV | DELETEVERIFY}={n | OFF} Default: 5

The DELETEVERIFY option specifies the maximum number of lines to be deleted without confirmation by the DELETE command. This safeguards the user against the accidental deletion of a larger number of lines than intended. This number may be set to any positive integer. Requests for

confirmation may be suppressed by setting DELETEVERIFY to OFF.

{E | ECHO}={ON | OFF} Default: OFF (terminal)
ON (batch)

If the ECHO option is ON, edit commands are echoed (printed) as they are executed.

{ERR | ERROREXIT}={ON | OFF} Default: OFF (terminal)
ON (batch)

If the ERROREXIT option is ON, the edit session is terminated if an error occurs. The editor may be reentered by issuing another $EDIT command.

{F | FILL | FILLCHAR}=char Default: blank

The fill character may be set to any single character. The fill character is used by the BLANK, JUSTIFY, MATCH, and OVERLAY commands.

{HEX | HEXADECIMAL | X}={ON | OFF}
Default: OFF

If the HEXADECIMAL option is ON, all character strings given in the command are treated as hexadecimal strings. For example, the string

```
"C8C5D3D3D640E3C8C5D9C5"
```

is equivalent to

```
"HELLO THERE"
```

when this option is ON. Verification is printed in hexadecimal. This command is also effective in visual mode.

LC={ON | OFF} Default: ON

If the LC option is ON, input text entered on the screen in visual mode will not be converted to uppercase. If the LC option is OFF, input text is converted to uppercase. This command applies only to the IBM 3278 terminal.

{LEN | LENGTH}={ON | OFF} Default: OFF

If the LENGTH option is ON, verification of a line includes its length.

{LNR | LINENUMBER}={ON | OFF} Default: ON

If the LINENUMBER option is ON, verification of a line includes its line number.

MAXLINE[=n] Default: 2147483.647

The maximum line number to be accessed in the file. Lines may not be read, modified, or inserted above this line number. If "n" is omitted, the maximum line number is reset to the default.

{MCL | MOVECURRENTLINE}={ON | OFF}
Default: ON

If the MOVECURRENTLINE option is ON, the current-line pointer (*) is moved by the action of edit commands.

MINLINE[=n] Default: –2147483.648

The minimum line number to be accessed in the file. Lines may not be read, modified, or inserted below this line number. If "n" is omitted, the minimum line number is reset to the default.

NOT={ON | OFF} Default: OFF

If the NOT option is ON, the pattern search in the SCAN and MATCH commands is successful when the pattern is *not* found. That is, the search succeeds at any position in the line where the pattern string fails to match. For example, this function in conjunction with the COLUMN command may be useful in finding a line that does not contain a blank in a given column range.

{NPP | NONPRINTPROTECT}={ON | OFF}
Default: ON

If the NONPRINTPROTECT option is ON, intensified lines will not be modifiable when in visual mode. These are lines containing nonprintable characters, lowercase letters when LC=OFF, or the last line of the screen when truncated.

{OPL | ONCEPERLINE}={ON | OFF} Default: OFF

If the ONCEPERLINE option is ON, only one operation such as an alteration or successful scan is allowed per line. The setting of ONCEPERLINE to ON often implies the setting of the ALL option to ON.

{P | PAD | PADCHAR}=char Default: blank

The pad character may be set to any single character. The pad character is used by the APPEND, BLANK, JUSTIFY, OVERLAY, and SHIFT commands.

{PA | PRINTALL}={ON | OFF} Default: OFF

If the PRINTALL option is ON, verification of every alteration of a line is given when both the ALL and VERIFY options are ON. If this option is OFF, only one verification is printed per line regardless of the number of alterations made. The PRINTALL option only applies to the ALTER and CHANGE commands.

PATTERNS={ON | OFF} Default: OFF

If the PATTERNS option is ON, the pattern-matching facility is enabled. See the section "Pattern Matching" for a description of this facility.

{PC | PRINTCOLUMN={ON | OFF} Default: OFF

If the PRINTCOLUMN option is ON, the column number of any successful pattern search is printed. In visual mode, the column number is printed in the ruler area.

{PLN | PREFIXLINENUMBER}={ON | OFF}
Default: ON

If the PREFIXLINENUMBER option is ON and an increment has been specified on the INSERT command, the user will be prompted for insertion lines with the line numbers at which the lines are to be inserted.

{PMAR | PRINTMARGIN}=n

The PRINTMARGIN option specifies the right print margin. When verification is performed, lines exceeding the right print margin are continued on the next line with a minus sign prefix character. The default right print margin is set on initial entry to the editor and is equal to the *SINK* output length. The margin may be reset to an integer value in the range of 32 to 255.

REMEMBER={ON | OFF} Default: OFF

If the REMEMBER option is ON, an editor remember operation is automatically done when returning to edit a file on which a save operation was done. See the SAVE and REMEMBER commands for further information.

REPSCAN={ON | OFF} Default: OFF

If the REPSCAN option is ON, replacement scanning will be performed by edit commands. The variables which are to have their values inserted must be enclosed by the replacement-scan delimiters (see the RPSCDELIM option).

RPSCDELIM=c^1c^2 Default: None

"c^1" and "c^2" specify the replacement-scan delimiters which are used to enclose the variables that are to have their values replaced in a pattern. "c^1" and "c^2" must be different characters, e.g., { and }.

RPSCECHO={ON | OFF} Default: ON

If the RPSCECHO option is ON, the results of each replacement scan operation will be echoed.

{RS | RESCAN}={ON | OFF} Default: OFF

If both the RESCAN and ALL options are ON, the ALTER and CHANGE commands resume pattern scanning at the replaced string (rather than after it) after a successful alteration. If these options are ON or if they are issued as command modifiers, accidental infinite loops may be created. For example,

```
ALTER@A@RS 123 "the"then"
```

would cause the following line:

```
Farmer in the dell
```

to become:

```
Farmer in thennnnnnnnnnnnnnn ...
```

To limit accidental loops, the rescan count at the same column position is limited to a number equal to the maximum possible length of the line. One useful purpose for this feature is altering strings of blanks to one blank:

```
ALTER@A@RS /FILE "  " "
```

where all pairs of blanks are changed to a single blank.

{RTL | RIGHTTOLEFT}={ON | OFF} Default: OFF

If the RIGHTTOLEFT option is ON, pattern scanning is performed from right to left within a line, i.e., the last occurrence of the pattern in the line is matched first.

SAVE={ON | OFF} Default: OFF

If the SAVE option is ON, an editor save operation is automatically done when exiting the editor or when the EDIT command is used to edit a new file. See the SAVE and REMEMBER commands for further information.

{SAVEREM | SAVEREMEMBER}={ON | OFF}
Default: OFF

If the SAVEREMEMBER option is ON, an editor save operation is automatically done when exiting the editor or when the EDIT command is used to edit a new file. Similarly, an automatic remember operation is done when returning to edit a file on which a save operation was done. See the SAVE and REMEMBER commands for further information.

SPELLCOR={ON | OFF} Default: OFF

If the SPELLCOR option is ON, the editor will perform spelling correction on edit commands in the same manner as spelling correction is performed on MTS commands. If the SPELLCOR option is OFF, spelling correction

is not performed.

STOP={ON | OFF} Default: ON

If the STOP option is OFF, the execution of the edit STOP command or an end-of-file while reading a command is treated as the edit RETURN command. If the STOP option is ON, the editing session is terminated in the above situations.

TIMEINTERVAL={n | OFF} Default: 3 seconds

If the TIMEINTERVAL option is enabled, the editor will print a warning message after every "n" seconds of CPU time have been used for a single edit command. If TIMEINTERVAL is OFF, the warning message will be suppressed. "n" must be ≥ 0.

TRIM={ON | OFF} Default: OFF

If the TRIM option is ON, all lines written or changed in the file are trimmed to remove all but one trailing blank.

{TX | TEXT | LINETEXT}={ON | OFF} Default: ON

If the TEXT option is ON, verification of a line includes its contents.

VERIFY={ON | OFF} Default: ON

If the VERIFY option is ON, the resulting lines are printed after they are processed by an edit command. The following options govern the output produced by verification: LENGTH, LNR, PA, TEXT, WINDOW, and X.

VISUAL={ON | OFF} Default: OFF

If the VISUAL option is ON, the editor will be entered in visual mode instead of line mode. This is the same as if the :VISUAL command were appended to the EDIT command.

VMARKER={ON | OFF} Default: OFF

If the VMARKER option is ON, the current line (in visual mode) will be marked by adding as asterisk and highlighting the number field.

VMCOST={ON | OFF | NONE} Default: ON

If the VMCOST option is ON, the cost of the edit session is given in the cost field of the work area when in visual mode. Setting this option to OFF may be useful when using the command field of the work area to reposition the file to lines with large line numbers. When this option is OFF, a colon will separate the cost and work fields. When this option is NONE, the separating colon is omitted.

VML={ON | OFF | NONE} Default: ON

If the VML option is ON, line numbers are displayed in visual mode to the left of the text. If this option is OFF, periods will appear in place of line numbers. If this option is NONE, the line-number field is omitted.

{VMV | VISUALMODEVERIFY}={ON | OFF}
Default: OFF

If the VMV option is ON, verification of commands executed from the work area in visual mode will appear in the conversation buffer, i.e., when visual mode is exited, the verification will appear on the screen.

WINDOW={ON | OFF} Default: ON

If the WINDOW option is ON, the text of lines printed during verification includes only those columns specified by the WINDOW command.

Modifiers: None.

Conditions: None.

Example:

```
SET VERIFY=ON FILLCHAR=$
```

The above SET command sets the VERIFY option to ON and the fill character to "$".

SHIFT

Edit Command Description

Purpose: To shift parts of a line to the left or the right.

Prototype: SHIFT [lpar] direction count

where "direction" is LEFT, L, RIGHT, or R and "count" is an integer.

Action: The SHIFT command shifts the contents of the current column range in the given line-number range "lpar" in the specified direction by "count" characters. For a right shift, the pad character is inserted into the vacated positions. For a left shift, the pad character is inserted into the vacated positions if the line extends beyond the left column pointer. If the line does not extend beyond the left column pointer, the shift is suppressed for that line. Characters that are shifted out of the current column range are lost.

Modifiers: @CH, @COL, @LEN, @LNR, @MCL, @TR, @TX, @V, @VMV, @W

Conditions: SUCCESS if at least one line is shifted. EOF if a read beyond the end of the file is attempted.

Examples:

```
SHIFT /FILE LEFT 9
```

In the above example, every line of the file is shifted to the left by 9 positions. If the current column range is 1 to 32767, then the first 9 columns of the file are removed.

```
SHIFT /FILE COL=(12,16) RIGHT 2
```

In the above example, the characters in columns 12 through 16 of every line of the file are shifted to the right by two column positions. Columns 12 and 13 are filled with the pad character and the previous contents of columns 15 and 16 are lost. Column 17 does not change.

SPREAD

Edit Command Description

Purpose: To evenly renumber a region of the file.

Prototype: SPREAD [lpar]

Action: The lines in the region "lpar" are renumbered so that they are evenly spread with respect to the line-number range. This command is useful for obtaining more insertion room between lines when they are different by only .001.

The last line is not renumbered (unless *L is specified), e.g., the command

```
SPREAD 1 10
```

renumbers lines from 1 up to but not including 10.

Modifiers: None.

Conditions: SUCCESS if the region is successfully renumbered.

Example:

```
SPREAD 1056 1057
```

The above command evenly renumbers the lines in the file starting with line 1056 and ending with line preceding 1057.

STOP

Edit Command Description

Purpose:	To terminate the editing session and return to the caller.
Prototype:	STOP
Action:	The editing session is terminated and control is returned to the caller (normally MTS command mode). All editor workspace and buffers are released. An end-of-file also terminates the edit session. If the editor STOP option is ON, the STOP command becomes identical to the RETURN command, that is, the edit session is not terminated. If a patterned file name was specified on the EDIT command, the next file in the pattern sequence (if any) is edited.
Modifiers:	None.
Conditions:	None.

TRANSLATE

Edit Command Description

Purpose: To translate characters in one or more lines.

Prototype:
(a) TRANSLATE [lpar] 'string1'string2'
(b) TRANSLATE [lpar] [[FROM] fromset] [TO] toset
(c) TRANSLATE [lpar]
(d) TRANSLATE [lpar] strexp1=strexp2

Action: With prototype (a), each character in "string1" found in the line(s) is translated to the corresponding character in "string2". The correspondence is by character position in the strings. The line range translated is specified by "lpar". "string1" must be the same length as "string2".

With prototype (b), the characters in each line are taken from the character set "fromset" and the characters are translated to the corresponding characters in "toset". The legal "fromset-toset" pairs are:

fromset	*toset*	*Function*
EBCDIC	ASCII	EBCDIC to ASCII conversion
ASCII	EBCDIC	ASCII to EBCDIC conversion
UC	LC	uppercase to lowercase conversion
LC	UC	lowercase to uppercase conversion
BCD	EBCDIC	BCD to EBCDIC conversion
MTS	IBM	MTS EBCDIC to IBM Code Page 37 EBCDIC conversion
IBM	MTS	IBM Code Page 37 EBCDIC to MTS EBCDIC conversion

The "fromset" character set may be omitted in every case except for BCD to EBCDIC conversion.

The to/from sets MTS and IBM are for use in converting files created before February 22, 1988 on MTS or those saved on tape and restored. The MTS EBCDIC character set differs in only a few locations from that of the more widely used standard IBM Code Page 37 character set. The characters that will be changed by the command

TRANSLATE lpar FROM MTS TO IBM

are left and right square brackets, left and right curly braces, backslash, circumflex, grave, and tilde. Normally this command should only be applied once to a "source" file.

With prototype (c), the last character translation specification given in the most recent TRANSLATE command specifying character translations is used.

With prototype (d), the characters in the line range that are specified in "strexp1" are translated to the corresponding characters in "strexp2". The correspondence is by character position in the string specified in the string expressions. The length of the string from "strexp1" must be the same as in "strexp2".

Modifiers: @CH, @COL, @LEN, @LNR, @MCL, @TR, @TX, @V, @VMV, @W, @X

Conditions: SUCCESS if at least one line is translated. EOF if a read beyond the end of the file is attempted.

Examples:

```
TRANSLATE 91 '1234'ABCD'
```

All the 1's in line 91 are translated to A's, all 2's to B's, all 3's to C's, and 4's to D's.

```
TRANSLATE /FILE FROM UC TO LC
```

The entire file is translated from uppercase to lowercase alphabetics.

```
TRANSLATE /FILE FROM MTS TO IBM
```

The entire file is translated from MTS EBCDIC to IBM Code Page 37 EBCDIC.

UNDO

Edit Command Description

Purpose: To undo the action of the last command if the checkpoint feature is enabled.

Prototype: UNDO

Action: This command reverses the modifications made to the file for the last command issued which modified the file. For example, if lines are unintentionally destroyed by a previous command, they may be recovered provided no further commands have been issued which modified the file. To undo a particular range of lines, the RESTORE command should be used. The UNDO command may undo the action of a previous UNDO command. The @NV modifier is recommended for suppressing verification when a large number of lines are involved.

For this command to be operative, the checkpointing must be either ON or OFF (the default), but *not* NEVER.

See the section “Checkpoint/Restore Facility” for further details.

Modifiers: @CH, @LEN, @LNR, @TR, @TX, @V, @W

Conditions: None.

UNLOCK

Edit Command Description

Purpose: To close and unlock the edit file. This allows other users to lock the file.

Prototype:
UNLOCK
UNLK

Action: If the edit file is a disk file, all current buffers in virtual memory are written to the disk. The file is then unlocked. The action is the same as that of the MTS UNLK subroutine (see MTS Volume 3, *System Subroutine Descriptions*).

UNLOCK has no effect on checkpoint information; this information remains available to the RESTORE and UNDO commands.

VISUAL

Edit Command Description

Purpose: To enter visual mode. The file is displayed on the video screen of a display terminal. As changes are made to the screen, they are recorded in the file. This allows the user to edit the file using the local terminal features of the display terminal.

Prototype: VISUAL [linenumber]

Action: When the VISUAL command is issued, the contents of the file are displayed on the video screen starting at line "linenumber". If the line number is omitted, the current line is the first line displayed. In general, the current line is the last line edited or displayed by the file editor. Initially, the current line is the first line of the file.

See the section "Visual Mode" for further details on using visual mode in the editor.

Modifiers: @HEX

If @HEX is specified, the screen display will contain the hexadecimal representation of the data. The display remains in hexadecimal until another VISUAL(@NHEX) command is given.

Conditions: None.

WINDOW

Edit Command Description

Purpose: To set the column range for verification printing and the width of the data to be viewed in visual mode.

Prototype:
(a) WINDOW
(b) WINDOW left
(c) WINDOW left right
(d) WINDOW {LEFT | RIGHT}
(e) WINDOW {LEFT | RIGHT} count

Action: With prototype (a), the window pointers (the left and right column values) are set to 1 and 32767 (the defaults).

With prototype (b), the window pointers are set to "left" and 32767.

With prototype (c), the window pointers are set to "left" and "right".

With prototype (d), the window pointers are shifted to the left or right extreme so that either the leftmost or rightmost full-screen width is displayed. When the window is shifted to the left, the left column value is set to 1 and the right column value is set to the width of the screen. When the window is shifted to the right, the right column value is set to the last character of the longest line in the file and the left column is set to the right column value less the screen width.

With prototype (e), the window pointers are decremented (with LEFT) or incremented (with RIGHT) by the value of "count". This in effect shifts the window of the screen to the left or right by the value of "count".

The window pointers set by the WINDOW command remain in effect for both line mode and visual mode. That is, if a WINDOW LEFT command is executed in visual mode for a terminal that supports a window size of 68 characters, then only columns 1-68 would be printed when a subsequent PRINT command is executed in line mode.

Prototypes (a), (b), and (c) cause lines that are longer than the screen width to be wrapped in visual mode. Prototypes (d) and (e) truncate lines outside of the window area instead of wrapping them.

If a line being verified is less than "left" characters in length, then nothing is printed for that line.

In visual mode, lines that have their rightmost portions truncated because of a window setting are marked with ">" in the line-number field. Lines that cannot be seen because a window setting truncates them on the left are marked with "<". In both cases, the markers point towards the hidden part of the lines. In addition, truncated lines will be padded with the current pad character (initially a blank) if data is added on the screen to the line. This is convenient for adding comments to a program, for example.

Modifiers: None.

Conditions: SUCCESS if no error messages are generated.

Example:

```
WINDOW 2 4
```

In the above example, the window pointers are set to columns 2 and 4, respectively. This could be useful for printing only columns 2 through 4 of a file containing an object module. If visual mode is entered, only columns 2, 3, and 4 will be displayed for each line on the screen.

/name

Edit Command Description

Purpose: To print the line numbers defined by the region "/name".

Prototype: /name

Action: This command prints the line numbers that are the bounds for the region "/name".

Modifiers: None.

Conditions: None.

Example:

```
/ABC
```

The above command prints the line numbers that are the bounds for the region /ABC.

!name

Edit Command Description

Purpose: To execute an edit procedure.

Prototype:
(a) !name n
(b) !name

Action: With prototype (a), the edit procedure "!name" is executed "n" times.

With prototype (b), the edit procedure is executed once.

The action of this command is identical to that of the EXECUTE command.

Modifiers: None.

Conditions: None.

Example:

```
!PROCB 10
```

In the above example, the edit procedure !PROCB is executed 10 times.

±n

Edit Command Description

Purpose: To move the current-line pointer or visual-mode top line forward or backward "n" lines.

Prototype:
(a) n
(b) +n
(c) –n

where "n" is an integer.

Action: This command moves the current-line pointer (*) or visual-mode top line (*VT) forward or backward by "n" lines. The sign of the number indicates the direction to move.

Thus, if the current line is 4, the command

```
1
```

moves the current-line pointer or visual-mode top line to the next line (which is not necessarily 5). Likewise,

```
-1
```

moves the current-line pointer or visual-mode top line to the preceding line (which is not necessarily 3).

In an edit procedure, ±n only affects the current-line pointer.

Modifiers: @LEN, @LNR, @TX, @V, @W, @X

Conditions: SUCCESS if the current-line pointer is moved. EOF if the current-line pointer is moved beyond the end of the file.

Example:

```
10
```

The above command moves the visual-mode top line forward by 10 lines, if in visual mode, or the current-line pointer by 10 lines, otherwise.

±n.n

Edit Command Description

Purpose: To increment the current-line pointer or visual-mode top line by "±n.n".

Prototype: ±n.n

where "n.n" is a decimal number.

Action: This command increments the current-line pointer (*) or visual-mode top line (*VT) by "±n.n". The number "n.n" must contain a decimal point and must be in the range –99999.999 to 99999.999.

In an edit procedure, ±n only affects the current-line pointer.

Modifiers: @LEN, @LNR, @TX, @V, @W, @X

Conditions: EOF if the current-line pointer or visual-mode top line is set to a line which does not exist in the file.

Example:

```
10.0
```

The above example increments the visual-mode top line by 10.0, if in visual mode, or the current-line pointer by 10.0, otherwise.

VISUAL MODE

For the display-screen terminal user, the visual-mode feature provides a very powerful method for entering data into a newly created file or for editing an existing file. Visual mode is supported on several types of terminals including the Ontel, the DEC VT100, the IBM 3278, the Ontel Amigo, IBM PC and its compatibles, and the Apple Macintosh microcomputers.

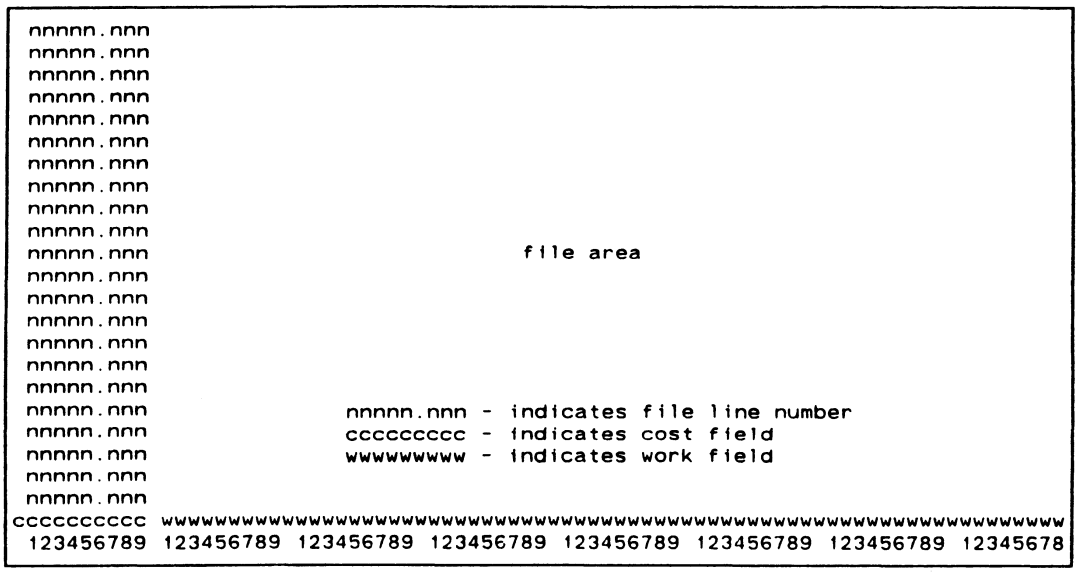


Figure 1. Visual-Mode Screen Format

In visual mode, the screen is divided into three areas:

(1) A "file area" consisting of several lines of the file with their associated file line numbers.

(2) A "work area" consisting of two fields: a 10-character "cost field" and a "work field," which may be extended for as long as needed.

(3) A "ruler area" that is the last line of the screen which is marked by default as a ruler for quick counting of column numbers and lengths in the file and work areas. This area may also be set to a status line as described in the section "Setting the Visual-Mode Ruler."

Visual mode is entered by first entering edit mode in the editor, i.e.,

EDIT filename

and then issuing the VISUAL command in the form

VISUAL n

where "n" is the line number of the first line in the file that is to appear at the top of the screen. If "n" is omitted, the current line (initially the first line of the file) appears at the top of the screen.

After entering visual mode, the user may make changes directly to the text of the file as it is displayed on the screen. These changes are recorded in a screen buffer within the terminal and are used by the editor at appropriate times to update the actual contents of the file.

Overview of Visual Mode with the Ontel

The following paragraphs briefly describe the use of visual mode with the Ontel terminal, one of the simplest terminals that can be used with visual mode. The complete descriptions of the use of visual mode with the Ontel and with other terminals follow at the end of this section.

The movement of the cursor on the Ontel is controlled by the CURSOR-RIGHT, CURSOR-LEFT, CURSOR-UP, and CURSOR-DOWN keys. These keys are labelled with arrows. Movement of the cursor is also controlled by the TAB and RETURN keys. The following keys have special function in visual mode:

(1) The INSERT MODE key may be used to insert characters into a line. This is done by moving the cursor to the position of the insert, pressing the INSERT MODE key to enter insert mode, and then entering the desired text. To exit insert mode after the insertion is completed, the INSERT MODE key may be pressed again.

(2) The DELETE key may be used to delete characters from a line. This is done by moving the cursor to the position of the deletion and pressing the DELETE key as many times as needed to delete the undesired characters.

(3) The ERASE TO EOL key may be used to delete all characters in a line to the right of the cursor. An entire line may be deleted by moving the cursor to the beginning of the line before pressing ERASE TO EOL.

The method of extending lines and inserting new lines into the file is given below in the discussion of the program-function keys.

As changes are made to the screen, they are stored within a buffer inside the terminal (at this point, they have not been transmitted to MTS and therefore, the file has not been updated by the editor). When the user either exits visual mode or causes the editor to rewrite the screen, the changes to the file are transmitted to MTS and the editor, and the file is updated at that time. The following operations cause the file to be updated by the editor:

(1) Pressing either the ATTN or EOF key which exits visual mode and returns to normal editing mode.

(2) Pressing the PF0 key which updates the file without exiting visual mode.

(3) Pressing any of the other program-function (PF) keys which executes an editor visual program function.

Pressing the FIX key deletes the current changes to the screen buffer and rewrites the screen to its previous status when the file was last updated.

Thirteen different editor visual program functions (VPFs) are predefined for the Ontel terminal. These program functions may be used to execute special editor visual mode commands. On the Ontel, these visual program functions may be accessed by pressing the numeric keys in conjunction with the

CTRL key or by pressing one of the program-function keys along the top row of the terminal.

VPF	*Ontel Key*	*Effect*
0	PF0	Update screen
1	CTRL-1 or PF1	Back up 10 lines
2	CTRL-2 or PF2	Append to work area
3	CTRL-3 or PF3	Memory 1
4	CTRL-4 or PF4	Advance 10 lines
5	CTRL-5 or PF5	Insert work area
6	CTRL-6 or PF6	Memory 2
7	CTRL-7 or PF7	Advance 1 line
8	CTRL-8 or PF8	Execute work area
9	CTRL-9	Memory 3
10	CTRL-0	Insert lines
11	CTRL-–	Extend line
12	CTRL-↑	Previous memory

Editor visual program functions 1, 4, and 7 are used to move new sections of the file onto the screen:

(1) VPF 1 moves 10 lines backward in the file. This is equivalent to the file editor command –10 or the device command %WB=10.

(2) VPF 4 moves 10 lines forward in the file. This is equivalent to the file editor command +10 or the device command %WF=10.

(3) VPF 7 moves 1 line forward in the file. This is equivalent to the file editor command +1 or the device command %WF=1.

Editor visual program functions 10 and 11 are used to insert a new line into the file or to extend a line beyond the right boundary of the file area:

(1) VPF 10 inserts a nonnumbered, null-length line (space to insert a new line) immediately after the cursor position. This line will be marked by a period in the line-number field. Information then may be inserted into this line space and it will be written as a new line when the file is updated. Consecutive pressing of VPF 10 doubles the number of lines available for insertion and decreases the line-number spacing. Note: If the line-number number spacing is .001, VPF 10 will not work and the file must then be renumbered.

(2) If, in the process of inserting characters into a line, the right margin of the screen is reached, VPF 11 will create another screen line of space so that the line may be continued into this region. Note: Occasionally, VPF 11 must be used before the right margin is reached; this is because the line has blanks appended to the end of it.

The work area is composed of a “cost field” and a “work field.” The work field may be used as a depository to either edit lines from the file or to compose edit commands to be subsequently applied to the file.

The work field is used by editor visual program functions 2, 5, and 8:

(1) VPF 2 appends to the work field the last part of the line in the file area pointed to by the cursor.

(2) VPF 5 inserts the contents of the work field into the line pointed to by the cursor at the position of the cursor. The user may make a copy of a line or a part of a line in the work field (VPF 2), modify it, and deposit the resulting line elsewhere in the file (VPF 5).

(3) VPF 8 takes the contents of the work field and applies it as an edit command to the file. For example, the user may enter the following command into the work field

```
SHIFT /FILE LEFT 5
```

and execute the VPF 8 command. This shifts the contents of the entire file to the left by 5 columns.

The cost field, just before the work field, normally contains the total cost of the terminal session. If this field is modified to a line number, the screen will be positioned to that line number after any visual program function is executed. Editor visual program functions 1, 4, and 7 (normally for file-positioning) are ignored in this case. If the first character of the cost field is set to "C", the contents of the "work field" is executed as an editor command; the line that the cursor is pointing to becomes the editor "current line" for the editor command in the work area (this is the same function as performed by VPF 8). The displaying of the cost of the edit session in the cost field of the work area may be suppressed by issuing the the command

SET VMCOST=OFF

This may be useful when using the cost field to reposition the file to lines with large line numbers. A colon identifies the cost work field when VMCOST is OFF.

Editor visual program functions 3, 6, 9, and 12 control the file-position memory feature of visual mode. These may be set to "remember" a particular position in the file so that the user can quickly return to it at a later time.

(1) VPF 3, VPF 6, and VPF 9 control memory positions 1, 2, and 3, respectively, which are used to define the "remembered" positions and return to them. To define the current file position on the screen as a remembered position, press the memory position key twice in succession. To return to the remembered position, press the key once. Memory position 1 is initially defined as the first line of the file, memory position 2 is defined as line 1000, and memory position 3 is defined as the last line of the file.

(2) VPF 12 moves the screen to the "previous" file position which is either

(a) the file position before either the last memory key or VPF 12 was pressed,
(b) the file position before a line number in the cost field was set,
(c) the file position when visual mode was last exited, or
(d) the file position before an edit command was executed via VPF 8 or the character "C" in the cost field.

Visual mode may be exited by pressing the ATTN key or the EOF key (see the descriptions of the individual terminals that follow for the correct definition of these keys). If the EOF key is used, the contents of the "work area" are written to *SINK* for further use. Both exit methods will update the editor current line to the line last pointed to by the cursor if MCL is in effect. If an error occurs (I/O error, for example), then the editor may also leave visual mode to print the error message.

The action of the entire VISUAL command may be undone by pressing the ATTN key and issuing the UNDO command. Note that if an edit command is executed during visual mode, the modifications done before that command may not be undone. An UNDO command will restore all modifications only after and including that command. If the user wants complete restoring ability, the CHECKPOINT command must be issued followed by the RESTORE command to undo the changes made.

On the Ontel terminal, a reverse-video line number is a warning that the user may not modify that line on the screen (other terminals may denote this by using an intensified line number). This occurs because either

(1) the line contains nonprinting characters which have been displayed with question marks inserted in their place, or

(2) the line contains lowercase letters and the LC option is set to OFF.

An update to such a line would either replace the nonprinting characters with question marks or replace the lowercase characters with uppercase characters. Visual mode normally should not be used for editing lines which contain characters that are not in the terminal keyboard character set. The user may override this restriction by specifying the editor NPP (NONPRINTPROTECT) option, e.g.,

```
SET NPP=OFF
```

Verification of edit commands executed from the work area is normally suppressed. Verification may be enabled by appending the @VMV (VISUALMODEVERIFY) modifier to the executed command or by specifying the VMV option, e.g.,

```
SET VMV=ON
```

In this case, verification will be written into the terminal conversation buffer and will be visible when visual mode is exited.

The user will be automatically signed off by the system in visual mode if the screen is not updated within a 30-minute interval.

If @HEX is specified, the screen display will contain the hexadecimal representation of the data. The display remains in hexadecimal until another VISUAL@NHEX command is given.

In the following Ontel example, Panels 1 and 2 illustrate the initial entering of text into an empty file. The initial command sequence is

```
EDIT filename
V
```

After entering visual mode, the cursor (shown by the underscore character) is pointing to the beginning of line 1 and the screen is filled with empty lines. The $.12 figure in the lower-left corner is the current cost of the terminal session.

After the text has been entered and the file is updated by executing VPF 0 (pressing the PF0 key), the screen will appear as shown in Panel 2. The current cost of the terminal session has now risen to $.24.

Panel 3 shows the file after some simple editing is done to lines 7, 8, and 9. In line 7, the word "goop" is changed to "good" by moving the cursor to the position of the "p" and typing "d". In line 8, the phrase " Zones," is inserted by moving the cursor to the position ",", pressing the INSERT MODE key, and typing the text to be inserted. In line 9, the double occurrence of "the " is corrected by moving the cursor to the "t" of the second "the" and pressing the DELETE key four times.

Panels 4 through 7 illustrate the addition of new text to the file. In this case, the new text is to be inserted at line 16. The screen as shown in Panel 4 is obtained by giving the command

```
v 14
```

followed by moving the cursor to line 16 and executing VPF 10 (the CTRL-0 key) five times. As text is inserted after line 16, it will go on lines 16.05, 16.1, 16.15, etc.

Panel 5 illustrates the effect when a data or text line is to be inserted that is longer than one screen line. In this case, when the user nears the end of typing on line 16.4, VPF 11 (the CTRL-– key) should be executed to extend the line by one screen line. When VPF 11 is executed, the file is automatically updated and line numbers are assigned to the new lines. After the completion of the line, the screen will appear as shown in Panel 5. Because of the effect of VPF 11, line 19 has now disappeared from the screen.

Panel 6 illustrates the appearance of the screen after the remainder of the text has been entered and before VPF 0 (the PF0 key) is executed; Panel 7 illustrates the screen after VPF 0 is executed. Note that line 16.82 contains at least one blank while line 16.88 was empty. At this point, the current cost of the session is $.45.

Panels 8 and 9 illustrate the use of the work field to execute an edit command on the file. In Panel 8, the cursor has been positioned to the beginning of the work field and then the text of the SHIFT command has been entered in the work field. Next, VPF 8 (the CTRL-8 key) has been invoked to execute the SHIFT command, which shifts the contents of the entire file to the left by 5 spaces; the result of this shift is shown in Panel 9.

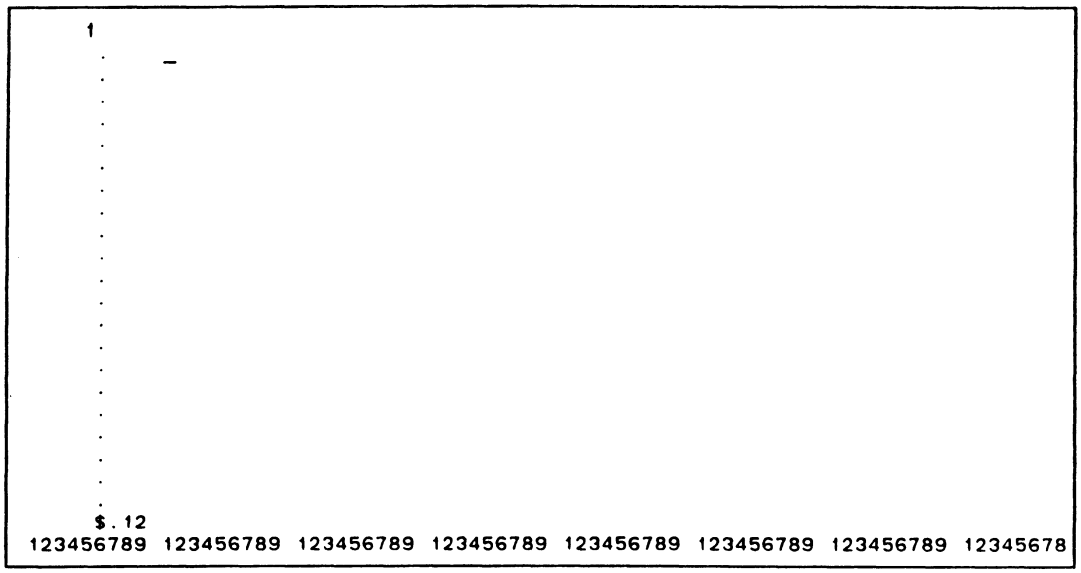


Figure 1. Screen Ready for Initial Text Entering

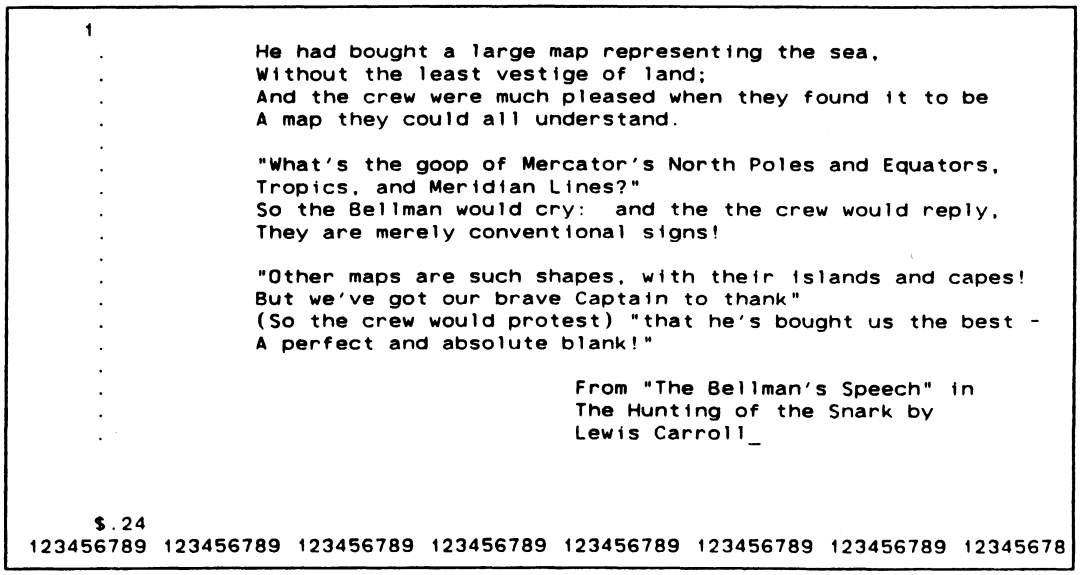


Figure 2. Screen After Initial Text Entering

```
     1
     2              He had bought a large map representing the sea,
     3              Without the least vestige of land;
     4              And the crew were much pleased when they found it to be
     5              A map they could all understand.
     6
     7              "What's the good of Mercator's North Poles and Equators,
     8              Tropics, Zones, and Meridian Lines?"
     9              So the Bellman would cry:  and_the crew would reply,
    10              They are merely conventional signs!
    11
    12              "Other maps are such shapes, with their islands and capes!
    13              But we've got our brave Captain to thank"
    14              (So the crew would protest) "that he's bought us the best -
    15              A perfect and absolute blank!"
    16
    17                                      From "The Bellman's Speech" in
    18                                      The Hunting of the Snark by
    19                                      Lewis Carroll



      $.26
123456789 123456789 123456789 123456789 123456789 123456789 123456789 12345678
```

Figure 3. Screen After Text Correction

```
    14              (So the crew would protest) "that he's bought us the best -
    15              A perfect and absolute blank!"
    16
       .     _
       .
       .
       .
       .
       .
       .
       .
       .
       .
       .
       .
       .
       .
    17                                      From "The Bellman's Speech" in
    18                                      The Hunting of the Snark by
    19                                      Lewis Carroll
      $.29
123456789 123456789 123456789 123456789 123456789 123456789 123456789 12345678
```

Figure 4. Screen Ready for Additional Text Insertion

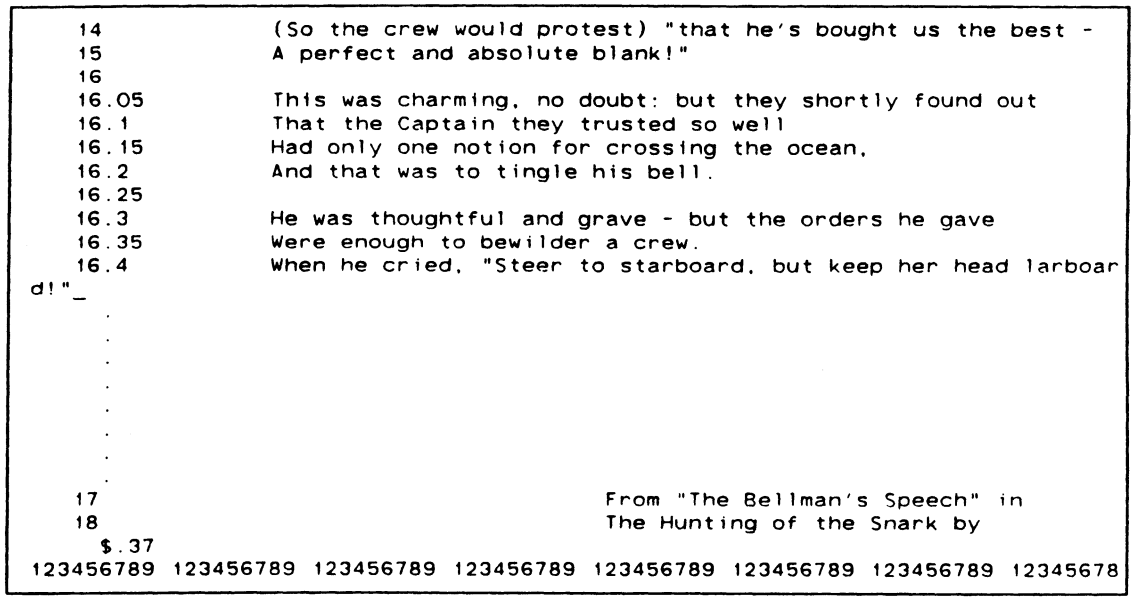


Figure 5. Screen After Extended Line Is Entered

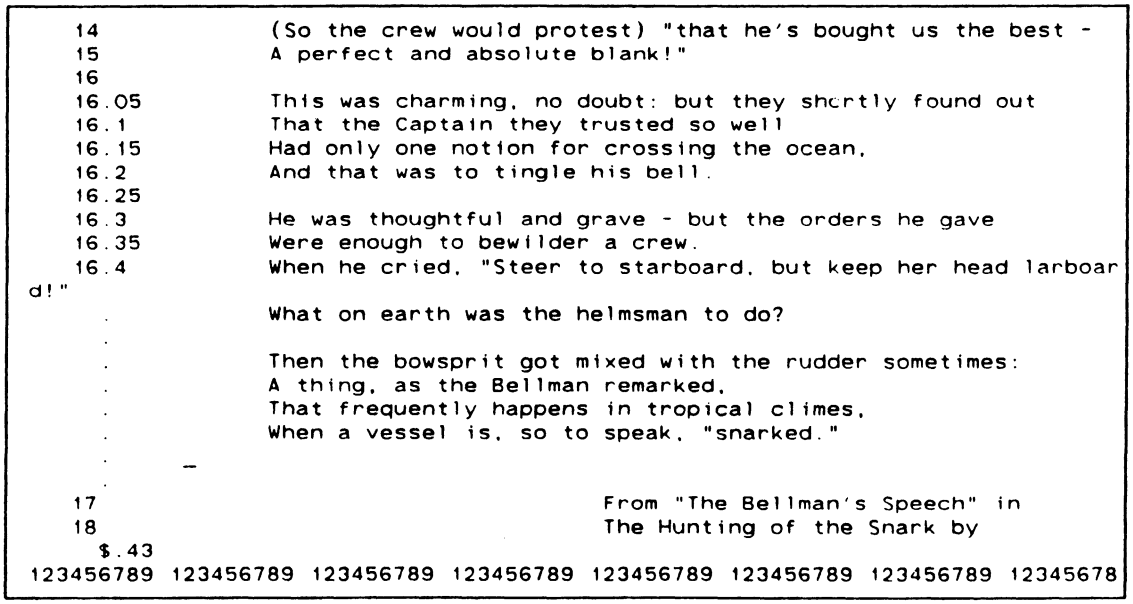


Figure 6. Screen After Additional Text Insertion

```
    14              (So the crew would protest) "that he's bought us the best -
    15              A perfect and absolute blank!"
    16
    16.05           This was charming, no doubt: but they shortly found out
    16.1            That the Captain they trusted so well
    16.15           Had only one notion for crossing the ocean,
    16.2            And that was to tingle his bell
    16.25
    16.3            He was thoughtful and grave - but the orders he gave
    16.35           Were enough to bewilder a crew.
    16.4            When he cried, "Steer to starboard, but keep her head larboar
d!"
    16.46           What on earth was the helmsman to do?
    16.52
    16.58           Then the bowsprit got mixed with the rudder sometimes:
    16.64           A thing, as the Bellman remarked,
    16.7            That frequently happens in tropical climes,
    16.76           When a vessel is, so to speak, "snarked."
    16.82
    17                                          From "The Bellman's Speech" in
    18                                          The Hunting of the Snark by
    19                                          Lewis Carroll
      $.45
123456789 123456789 123456789 123456789 123456789 123456789 123456789 12345678
```

Figure 7. Screen After File Is Updated

```
    14              (So the crew would protest) "that he's bought us the best -
    15              A perfect and absolute blank!"
    16
    16.05           This was charming, no doubt: but they shortly found out
    16.1            That the Captain they trusted so well
    16.15           Had only one notion for crossing the ocean,
    16.2            And that was to tingle his bell.
    16.25
    16.3            He was thoughtful and grave - but the orders he gave
    16.35           Were enough to bewilder a crew.
    16.4            When he cried, "Steer to starboard, but keep her head larboar
d!"
    16.46           What on earth was the helmsman to do?
    16.52
    16.58           Then the bowsprit got mixed with the rudder sometimes:
    16.64           A thing, as the Bellman remarked,
    16.7            That frequently happens in tropical climes,
    16.76           When a vessel is, so to speak, "snarked."
    16.82
    17                                          From "The Bellman's Speech" in
    18                                          The Hunting of the Snark by
    19                                          Lewis Carroll
      $.45 shift /file left 5
123456789 123456789 123456789 123456789 123456789 123456789 123456789 12345678
```

Figure 8. Screen Before SHIFT Command Execution

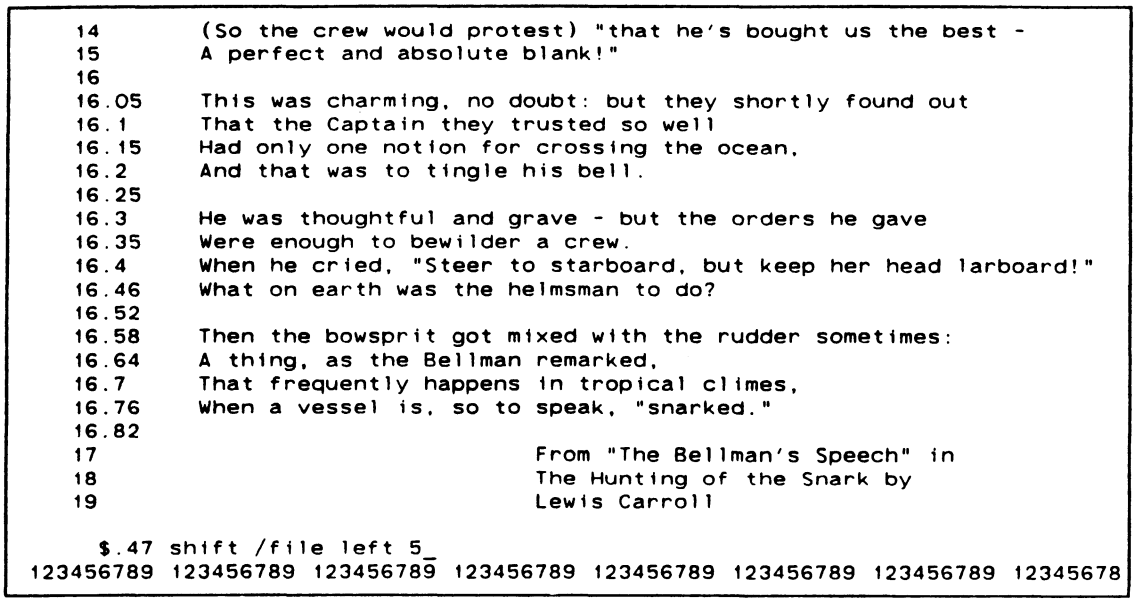
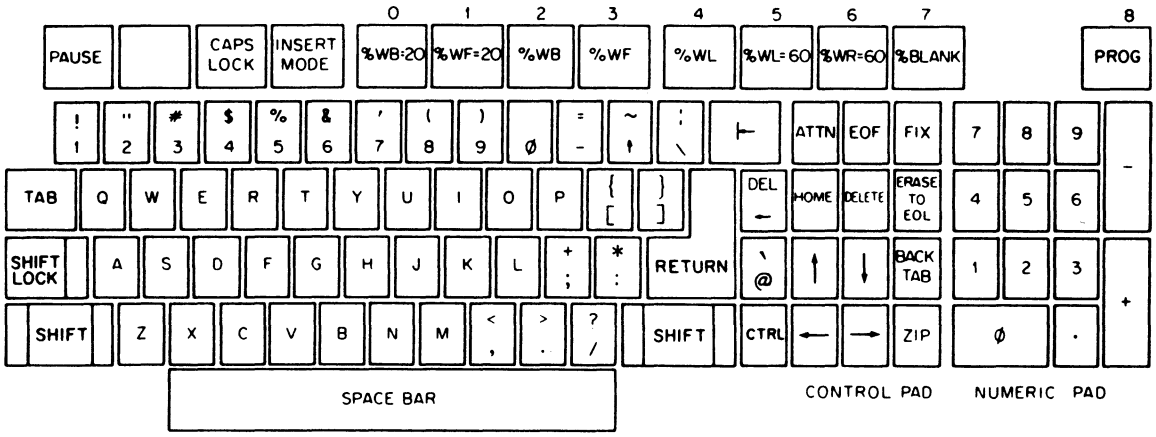


Figure 9. Final Screen Contents After Shifting

USING VISUAL MODE WITH THE ONTEL TERMINAL

The keyboard layout for the Ontel terminal is as follows:



The Ontel Keyboard - Model 1503

The movement of the cursor is controlled by the CURSOR-RIGHT, CURSOR-LEFT, CURSOR-UP, and CURSOR-DOWN keys. These keys are labelled with arrows. Movement of the cursor is also controlled by the TAB, BACKTAB, ZIP, HOME, and RETURN keys.

The following keys on the Ontel have special functions in visual mode. The actions of these keys are controlled only by the terminal routines and not by the editor program itself.

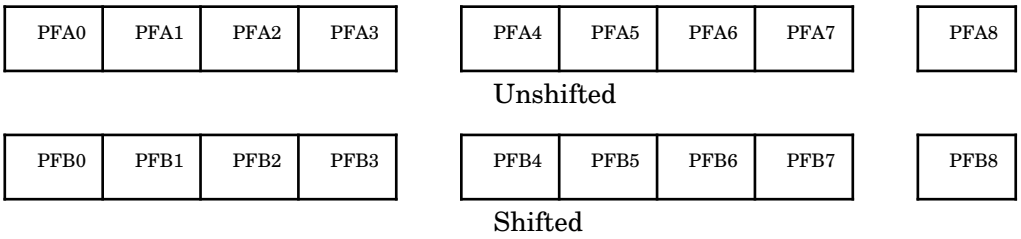
Ontel Key	*Effect*
INSERT MODE	Enter or exit insertion mode (toggle switch)
DELETE	Delete the character at the cursor
SHIFT-DELETE	Delete the work at the cursor
ERASE TO EOL	Delete all characters to the right of the cursor
FIX	Discard all current screen changes
SHIFT-HOME	Move cursor to work field

Across the top of the keyboard is the program-function pad. The nine rightmost keys are called program-function (PF) keys and are labelled with Ontel device commands. They are arranged in the following format:

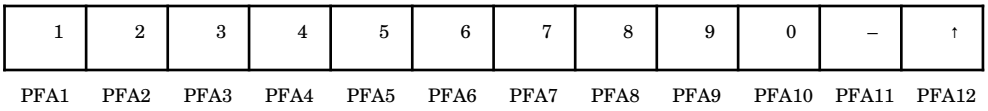
%WB=20	%WF=20	%WB	%WF	%WL	%WL=60	%WR=60	%BLANK	PROG
PF1	PF2	PF3	PF4	PF5	PF6	PF7	PF8	

On the Ontel Model OP-1/R (the gray models), the PROG key is labelled as %ENTER.

Each program-function key has an Ontel *program function* associated with it. These Ontel program functions are named PFA0 through PFA8 (for the unshifted keys) and PFB0 through PFB8 (for the shifted keys).



In addition, the top row of numeric keys, when depressed in combination with the CTRL key, have Ontel program functions assigned to them:



Normally (when not in visual mode), the program-function keys execute the Ontel device commands that are inscribed on the key tops, both in the unshifted and shifted positions. In visual mode, each of the program-function keys is assigned to an editor visual program function (VPF) for visual-mode editing. These visual program functions in turn are assigned to visual-mode commands. The default key assignments for the Ontel terminal are as follows:

VPF	*Ontel Key*	*Ontel PF*	*Default Assignment*
–2	EOF	-	VEXIT ED_WORK
–1	ATTN	-	VEXIT
0	PF0	PFA0	VUPDATE
1	CTRL-1 or PF1	PFA1	LINE *VT COUNT=–10
2	CTRL-2 or PF2	PFA2	VPFB
3	CTRL-3 or PF3	PFA3	VMEMORY 1
4	CTRL-4 or PF4	PFA4	LINE *VT COUNT=10
5	CTRL-5 or PF5	PFA5	VPFE
6	CTRL-6 or PF6	PFA6	VMEMORY 2
7	CTRL-7 or PF7	PFA7	LINE *VT COUNT=1
8	CTRL-8 or PF8	PFA8	VEXECUTE
9	CTRL-9	PFA9	VMEMORY 3
10	CTRL-0	PFA10	VINSERT 1 2
11	CTRL-–	PFA11	VEXTEND
12	CTRL-↑	PFA12	VPREVIOUS
13	-	PFA13	LINE *VT COUNT=–10
14	-	PFA14	VPFB
15	-	PFA15	VMEMORY 1
16	SHIFT-PF0	PFB0	LINE *VT COUNT=10
17	SHIFT-PF1	PFB1	VPFE
18	SHIFT-PF2	PFB2	VMEMORY 2
19	SHIFT-PF3	PFB3	LINE *VT COUNT=1
20	SHIFT-PF4	PFB4	VEXECUTE

21	SHIFT-PF5	PFB5	VMEMORY 3
22	SHIFT-PF6	PFB6	VINSERT 1 2
23	SHIFT-PF7	PFB7	VEXTEND
24	SHIFT-PF8	PFB8	VPREVIOUS
25	-	PFB9	Undefined
26	-	PFB10	Undefined
27	-	PFB11	Undefined
28	-	PFB12	Undefined
29	-	PFB13	Undefined
30	-	PFB14	Undefined
31	-	PFB15	Undefined

Note that the default assignments for editor visual program functions (VPFs) 1 through 12 are the same as for 13 through 24. Editor VPFs 13 through 24 do not have a default key assignment for the Ontel, but the user may assign the Ontel PF with an Ontel key (see the section "Redefining the Ontel Keyboard" below). Editor VPFs 25 through 31 are initially undefined, but they may be defined by the user (see the section "Reassigning Ontel Visual Program Functions" below).

Using the Numeric Pad

The numeric pad may be redefined to serve as a program-function pad by issuing the Ontel device command

%PAD=PFKEYS

The numeric pad keys then have the following configuration:

PFA13	PFA14	PFA15	PFA0
PFB0	PFB1	PFB2	
PFB3	PFB4	PFB5	PFB8
PFB6		PFB7	

Unshifted

PFA1	PFA2	PFA3	PFA0
PFA4	PFA5	PFA6	
PFA7	PFA8	PFA9	PFA12
PFA10		PFA11	

Shifted

This redefinition is effective *both* in visual mode and in normal terminal mode. The configuration of the numeric pad depends on whether the user is in visual mode or normal line mode.

The %PAD device command may be issued directly from the terminal keyboard or may be issued from the user's sigfile using the command

$CONTROL *MSOURCE* PAD=PFKEYS

When the user is not in visual mode, the numeric-pad keys either will be defaulted to Ontel device commands or will have definitions that are assigned by the user by the %FUNCTION device command (see MTS Volume 4, *Terminals and Networks in MTS*).

If the %PAD=PFKEYS device command is not given, the keys of the numeric pad retain their normal numeric-mode functions while in visual mode.

The function of the numeric pad may be restored to numeric mode by issuing the device command

%PAD=NUMERIC

Reassigning Ontel Visual Program Functions

The example below illustrates how the editor visual program functions may be reassigned to take advantage of the additional Ontel program functions that are available on the numeric pad.

```
VPF 13=LINE *VT COUNT=1          (assign PFA13)
VPF 14=LINE *VT COUNT=10         (assign PFA14)
VPF 15=LINE *VT COUNT=100        (assign PFA15)
VPF 16=LINE *VT COUNT=-1         (assign PFB0)
VPF 17=LINE *VT COUNT=-10        (assign PFB1)
VPF 18=LINE *VT COUNT=-100       (assign PFB2)
VPF 19=VMEMORY 1                 (assign PFB3)
VPF 20=VMEMORY 2                 (assign PFB4)
VPF 21=VMEMORY 3                 (assign PFB5)
VPF 22=LINE *F                   (assign PFB6)
VPF 23=LINE *L                   (assign PFB7)
VPF 24=VPREVIOUS                 (assign PFB8)
```

Editor visual program functions VPF 13 through VPF 24 are reassigned to different visual mode commands so that they do not duplicate VPF 1 through VPF 12. This in effect makes the Ontel unshifted numeric pad which uses the Ontel program functions PFA13 through PFA15 and PFB0 through PFB8 different from the shifted numeric pad which uses Ontel program functions PFA0 through PFA12. In this example, a selection of visual mode and normal edit commands is chosen so that the user may easily move around within the file.

As a second example, the following editor initialization file will reassign part of the program functions to visual-mode commands, part to normal editor commands, and part to edit procedures to provide more advanced capabilities for the program-function pad.

```
*
* Define a procedure to execute a %GRAB device
* command to initiate a second session
*
PROCEDURE !GRAB
UNLOCK
$CONTROL *MSOURCE* GRAB
END
*
* Define a procedure to execute a %FLIP device
* command to switch to the alternate session
*
PROCEDURE !FLIP
UNLOCK
$CONTROL *MSOURCE* FLIP
END
*
* Define a procedure to set variable A
* to beginning line for MOVE after
* moving the cursor to desired line
*
```

```
PROCEDURE !SETA
LET A=*
END
*
* Set variable B to ending line for MOVE
* after moving the cursor to desired line
*
PROCEDURE !SETB
LET B=*
END
*
* Redefine some PF keys
*
VPF 13=!GRAB                      (assign PFA13)
VPF 14=!FLIP                      (assign PFA14)
VPF 15=!SETA                      (assign PFA15)
VPF 16=!SETB                      (assign PFB0)
VPF 17=MOVE A B TO *              (assign PFB1)
VPF 18=COMBINE * *N LEN=65        (assign PFB2)
VPF 19=VSPLIT                     (assign PFB3)
VPF 20=COPY * TO *                (assign PFB4)
VPF 21=RENUMBER                   (assign PFA5)
```

The above example illustrates how edit procedures may be used to execute terminal device commands and more complex editor commands. PFA13 and PFA14 have been reassigned to execute the %GRAB and %FLIP terminal device commands. These commands are executed via the MTS $CONTROL command (see MTS Volumes 1 and 4 for further details about issuing terminal device commands via the $CONTROL command). The edit UNLOCK command writes the current changes on the screen into the edit file and unlocks the file before the device command is executed. PFA15, PFB0, and PFB1 have been reassigned to work in concert to easily move lines in the file from one location to another. PFA15 and PFB0 specify the beginning and ending bounds of the lines to be moved and PFB1 moves the text to the location specified by the current position of the cursor. PFB2 has been reassigned to the COMBINE command to combine the line pointed to by the cursor with the next line; the resultant lines are limited to 65 characters. PFB3 has been reassigned to the VSPLIT command which splits a line into two lines at the position of the cursor. PFB4 is reassigned to the COPY command to duplicate the line pointed to by the cursor. PFB5 is reassigned to the RENUMBER command.

Redefining the Ontel Keyboard

The Ontel terminal supports a total of 32 Ontel program functions, named PFA0 through PFA15 and PFB0 through PFB15. By default, program-function keys PF0 through PF8 are defined on the function pad across the top row of the keyboard as Ontel program functions PFA0 through PFA8, and the CTRL-1 through CTRL-↑ keys are defined as Ontel program functions PFA1 through PFA12. All other Ontel program functions must be assigned to specific terminal keys by the user.

The %FUNCTION device command may be used to assign the remaining Ontel program functions PFB9 through PFB15 or to reassign Ontel program functions PFA0 through PFA15 and PFB0 through PFB8. The %FUNCTION device command is given in the form

%FUNCTION n m

where “n” is a hexadecimal value describing the key to be reassigned and “m” is a hexadecimal value describing the function to be assigned to the key.

The hexadecimal values for program-function keys PF0 through PF8 on the function pad are as follows:

Key	*Hex Value Unshifted*	*Hex Value Shifted*
PF0	E4	F4
PF1	E5	F5
PF2	E6	F6
PF3	E7	F7
PF4	CE	DE
PF5	CA	DA
PF6	CC	DC
PF7	EE	FE
PF8	EF	FF

The hexadecimal values for the keys on the numeric pad are as follows:

Key	*Hex Value Unshifted*	*Hex Value Shifted*
0	C0	D0
1	C1	D1
2	C2	D2
3	C3	D3
4	C4	D4
5	C5	D5
6	C6	D6
7	C7	D7
8	C8	D8
9	C9	D9
+	CD	DD
–	CF	DF
.	CB	DB

The hexadecimal values for the remainder of the keys on the Ontel keyboard are given in MTS Volume 4, *Terminals and Networks in MTS*.

The hexadecimal values for the Ontel program functions are as follows:

Program Function	*Hex Value*	*Editor VPF*
PFA0 through PFA15	80 through 8F	0 through 15
PFB0 through PFB15	90 through 9F	16 through 31

As an example of assigning the Ontel program functions PFB9 through PFB15 for use with visual mode from the Ontel terminal, the following editor initialization file could be constructed.

```
$CONTROL *MSOURCE* FUNCTION F5 99
$CONTROL *MSOURCE* FUNCTION F6 9A
$CONTROL *MSOURCE* FUNCTION F7 9B
$CONTROL *MSOURCE* FUNCTION DE 9C
$CONTROL *MSOURCE* FUNCTION DA 9D
```

```
$CONTROL *MSOURCE* FUNCTION DC 9E
$CONTROL *MSOURCE* FUNCTION FE 9F
VPF 25=VSPLIT
VPF 26=RENUMBER
VPF 27=COPY * TO *
VPF 28=LINE *VT COUNT=-100
VPF 29=LINE *VT COUNT=100
VPF 30=LINE *VT COUNT=-200
VPF 31=LINE *VT COUNT=200
```

In the above example, the $CONTROL *MSOURCE* FUNCTION command redefines the shifted program-function keys PF1 through PF7 across the top row of the keyboard to execute Ontel program functions PFB9 through PFB15 instead of the default Ontel program functions PFA1 through PFA7. Keys PF0 and PF8 and the unshifted PF1 through PF7 keys are unchanged. These keys are subsequently assigned by the VPF command to the editor visual program functions VPF 25 through VPF 31. By issuing the %PAD=PFKEYS device command, the user will then have available 32 different visual program functions for use with visual mode, VPF 0 through VPF 24 on the numeric pad and VPF 25 through VPF 31 along the top row of the keyboard.

USING VISUAL MODE WITH THE ONTEL AMIGO

The keyboard layout for the Ontel Amigo terminal is the same as for the IBM PC.

The movement of the cursor is controlled by the CURSOR-RIGHT, CURSOR-LEFT, CURSOR-UP, and CURSOR-DOWN keys. These keys are labelled with arrows. Movement of the cursor is also controlled by the TAB, HOME, and RETURN keys.

The following keys on the Ontel Amigo have special functions in visual mode. The actions of these keys are controlled only by the terminal routines and not by the editor program itself.

Amigo Key	*Effect*
Ins	Enter or exit insertion mode (toggle switch)
Del	Delete the character at the cursor
SHIFT-Del	Delete the word at the cursor
SHIFT-Bkspc (←)	Delete all characters to the right of the cursor
Esc	Discard all current line changes
SHIFT-Esc	Discard all current screen changes
SHIFT-Home	Move cursor to work field

Along the left side of the keyboard is the program-function pad. These ten keys are called program-function keys and are labelled F1 through F10. Each program-function key has four Amigo *program functions* associated with it, one for each of the four possible key positions (unshifted-Fx, SHIFT-Fx, CTRL-Fx, and CTRL-SHIFT-Fx). These Amigo program functions are named PFA0 through PFA15, PFB0 through PFB15, and PFC0 through PFC8.

Normally (when not in visual mode), the program-function keys execute terminal device commands. In visual mode, each of the program-function keys is assigned to an editor visual program function (VPF) for visual-mode editing. These visual program functions in turn are assigned to visual-mode commands. The default key assignments for the Ontel Amigo terminal are as follows:

VPF	*Amigo Key*	*Amigo PF*	*Default Assignment*
–2	CTRL-C	-	VEXIT ED_WORK
–1	CTRL-E	-	VEXIT
0	PrtSc	PFA0	VUPDATE
1	F1	PFA1	LINE *VT COUNT=–10
2	F2	PFA2	VPFB
3	F3	PFA3	VMEMORY 1
4	F4	PFA4	LINE *VT COUNT=10
5	F5	PFA5	VPFE
6	F6	PFA6	VMEMORY 2
7	F7	PFA7	LINE *VT COUNT=1
8	F8	PFA8	VEXECUTE
9	F9	PFA9	VMEMORY 3
10	F10	PFA10	VINSERT 1 2
11	SHIFT-F1	PFA11	VEXTEND
12	SHIFT-F2	PFA12	VPREVIOUS
13	SHIFT-F3	PFA13	LINE *VT COUNT=–10
14	SHIFT-F4	PFA14	VPFB
15	SHIFT-F5	PFA15	VMEMORY 1

16	SHIFT-F6	PFB0	LINE *VT COUNT=10
17	SHIFT-F7	PFB1	VPFE
18	SHIFT-F8	PFB2	VMEMORY 2
19	SHIFT-F9	PFB3	LINE *VT COUNT=1
20	SHIFT-F10	PFB4	VEXECUTE
21	CTRL-F1	PFB5	VMEMORY 3
22	CTRL-F2	PFB6	VINSERT 1 2
23	CTRL-F3	PFB7	VEXTEND
24	CTRL-F4	PFB8	VPREVIOUS
25	CTRL-F5	PFB9	Undefined
26	CTRL-F6	PFB10	Undefined
27	CTRL-F7	PFB11	Undefined
28	CTRL-F8	PFB12	Undefined
29	CTRL-F9	PFB13	Undefined
30	CTRL-F10	PFB14	Undefined
31	CTRL-SHIFT-F1	PFB15	Undefined
32	CTRL-SHIFT-F2	PFC0	Undefined
33	CTRL-SHIFT-F3	PFC1	Undefined
34	CTRL-SHIFT-F4	PFC2	Undefined
35	CTRL-SHIFT-F5	PFC3	Undefined
36	CTRL-SHIFT-F6	PFC4	Undefined
37	CTRL-SHIFT-F7	PFC5	Undefined
38	CTRL-SHIFT-F8	PFC6	Undefined
39	CTRL-SHIFT-F9	PFC7	Undefined
40	CTRL-SHIFT-F10	PFC8	Undefined

Note that the default assignments for editor visual program functions (VPFs) 1 through 12 are the same as for 13 through 24. Editor VPFs 25 through 40 are initially undefined, but they may be defined by the user (see the section “Reassigning Amigo Visual Program Functions” below).

Reassigning Amigo Visual Program Functions

The example below illustrates how the editor visual program functions may be reassigned to make the Amigo program-function pad more versatile.

```
VPF  1=VPFB                    (reassign F1 (PFA1))
VPF  2=VPFE                    (reassign F2 (PFA2))
VPF  3=VEXECUTE                (reassign F3 (PFA3))
VPF  4=VSPLIT                  (reassign F4 (PFA4))
VPF  5=VINSERT 1 2             (reassign F5 (PFA5))
VPF  6=VEXTEND                 (reassign F6 (PFA6))
VPF  7=VMEMORY 1               (reassign F7 (PFA7))
VPF  8=VMEMORY 2               (reassign F8 (PFA8))
VPF  9=VMEMORY 3               (reassign F9 (PFA9))
VPF 10=VPREVIOUS               (reassign F10 (PFA10))
VPF 11=LINE *VT COUNT=-1       (reassign SHIFT-F1 (PFA11))
VPF 12=LINE *VT COUNT=1        (reassign SHIFT-F2 (PFA12))
VPF 13=LINE *VT COUNT=-10      (reassign SHIFT-F3 (PFA13))
VPF 14=LINE *VT COUNT=10       (reassign SHIFT-F4 (PFA14))
VPF 15=LINE *VT COUNT=-100     (reassign SHIFT-F5 (PFA15))
VPF 16=LINE *VT COUNT=100      (reassign SHIFT-F6 (PFB0))
VPF 17=LINE *VT COUNT=-1000    (reassign SHIFT-F7 (PFB1))
VPF 18=LINE *VT COUNT=1000     (reassign SHIFT-F8 (PFB2))
VPF 19=LINE *F                 (reassign SHIFT-F9 (PFB3))
VPF 20=LINE *L                 (reassign SHIFT-F10 (PFB4))
```

Editor visual program functions VPF 1 through VPF 12 are rearranged so that they more logically fit the Amigo program-function pad. Editor visual program functions VPF 13 through VPF20 are reassigned to different visual mode commands so that they do not duplicate VPF 1 through VPF 12. In this example, a selection of visual mode and normal edit commands is chosen so that the user may easily move around within the file.

As a second example, the following editor initialization file will reassign part of the program functions to visual-mode commands, part to normal editor commands, and part to edit procedures to provide more advanced capabilities for the program-function pad.

```
*
* Define a procedure to execute a %GRAB device
* command to initiate a second session
*
PROCEDURE !GRAB
UNLOCK
$CONTROL *MSOURCE* GRAB
END
*
* Define a procedure to execute a %FLIP device
* command to switch to the alternate session
*
PROCEDURE !FLIP
UNLOCK
$CONTROL *MSOURCE* FLIP
END
*
* Define a procedure to set variable A
* to beginning line for MOVE after
* moving the cursor to desired line
*
PROCEDURE !SETA
LET A=*
END
*
* Set variable B to ending line for MOVE
* after moving the cursor to desired line
*
PROCEDURE !SETB
LET B=*
END
*
* Define a procedure to clear work area
* before executing a VPFB command
*
PROCEDURE !VPFB
LET ED_WORK=''
VPFB
END
*
* Redefine some PF keys
*
VPF 21=!GRAB                    (reassign CTRL-F1 (PFB5))
VPF 22=!FLIP                    (reassign CTRL-F2 (PFB6))
VPF 23=!SETA                    (reassign CTRL-F3 (PFB7))
VPF 24=!SETB                    (reassign CTRL-F4 (PFB8))
VPF 25=MOVE A B TO *            (reassign CTRL-F5 (PFB9))
VPF 26=COMBINE * *N LEN=65      (reassign CTRL-F6 (PFB10))
VPF 27=COPY * TO *              (reassign CTRL-F7 (PFB11))
VPF 28=RENUMBER                 (reassign CTRL-F8 (PFB12))
```

```
VPF 29=!VPFB                    (reassign CTRL-F9 (PFB13))
VPF 30=VPFE                     (reassign CTRL-F10 (PFB14))
```

The above example illustrates how edit procedures may be used to execute terminal device commands and more complex editor commands. PFB5 and PFB6 have been reassigned to execute the %GRAB and %FLIP terminal device commands. These commands are executed via the MTS $CONTROL command (see MTS Volumes 1 and 4 for further details about issuing terminal device commands via the $CONTROL command). The edit UNLOCK command writes the current changes on the screen into the edit file and unlocks the file before the device command is executed. PFB7, PFB8, and PFB9 have been reassigned to work in concert to easily move lines in the file from one location to another. PFB7 and PFB8 specify the beginning and ending bounds of the lines to be moved and PFB9 moves the text to the location specified by the current position of the cursor. PFB10 has been reassigned to the COMBINE command to combine the line pointed to by the cursor with the next line; the resultant lines are limited to 65 characters. PFB11 is reassigned to the COPY command to duplicate the line pointed to by the cursor. PFB12 is reassigned to the RENUMBER command. PFB13 has altered the VPFB command so that the work area is emptied before the insertion is performed.

Redefining the Amigo Keyboard

The Ontel Amigo terminal supports a total of 48 Ontel program functions, named PFA0 through PFA15, PFB0 through PFB15, and PFC0 through PFC15. By default, program-function keys F0 through F10 are defined as Amigo program functions PFA0 through PFA15, PFB0 through PFB15, and PFC0 through PFC8. The other Ontel program functions PFC9 through PFC15 must be assigned to specific terminal keys by the user.

The %FUNCTION device command may be used to assign the Amigo program functions. The %FUNCTION device command is given in the form

%FUNCTION n m

where “n” is a hexadecimal value describing the key to be reassigned and “m” is a hexadecimal value describing the function to be assigned to the key.

The hexadecimal values for program-function keys F0 through F10 on the program-function pad are as follows:

Key	*Hex Value Unshifted*	*Hex Value Shifted*	*Hex Value CTRLed*	*Hex Value CTRL/Shifted*
F1	80	90	A0	B0
F2	81	91	A1	B1
F3	82	92	A2	B2
F4	83	93	A3	B3
F5	84	94	A4	B4
F6	85	95	A5	B5
F7	86	96	A6	B6
F8	87	97	A7	B7
F9	88	98	A8	B8
F10	89	99	A9	B9

The hexadecimal values for the keys on the Amigo numeric pad are as follows:

Key	*Hex Value Unshifted*	*Hex Value Shifted*	*Hex Value CTRLed*	*Hex Value CTRL/Shifted*
(Ins)	C0	D0	E0	F0
1 (End)	C1	D1	E1	F1
2	C2	D2	E2	F2
3 (Pg Dn)	C3	D3	E3	F3
4	C4	D4	E4	F4
5	C5	D5	E5	F5
6	C6	D6	E6	F6
7 (Home)	C7	D7	E7	F7
8	C8	D8	E8	F8
9 (Pg Up)	C9	D9	E9	F9
Scrl Lock	CA	DA	EA	FA
–	CB	DB	EB	FB
PrtSc	CC	-	-	-
+	CD	DC	EC	FC
. (Del)	CE	DD	ED	FD

The hexadecimal values for the Ontel program functions are as follows:

Program Function	*Hex Value*	*Editor VPF*
PFA0 through PFA15	80 through 8F	0 through 15
PFB0 through PFB15	90 through 9F	16 through 31
PFC0 through PFC15	A1 through AF	32 through 48

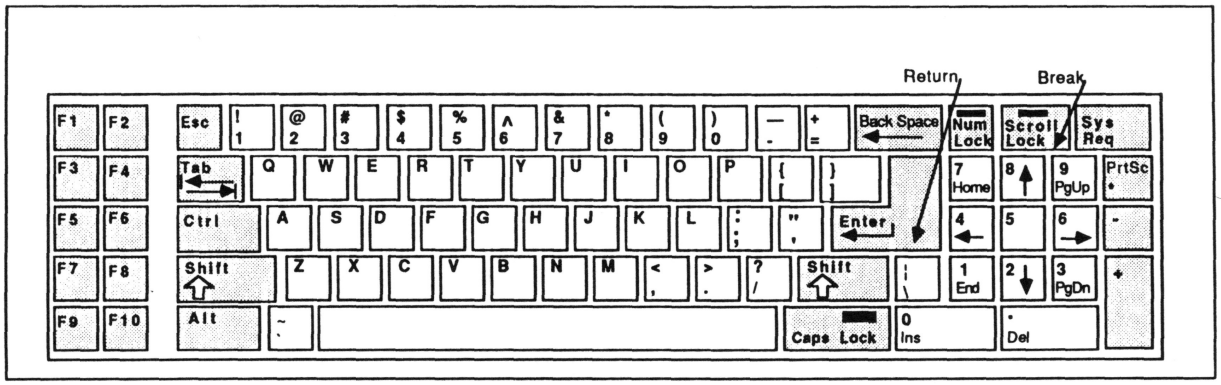
As an example of assigning the Amigo program functions PFB8 through PFC0 for use with visual mode from the Ontel Amigo terminal, the following editor initialization file could be constructed.

```
$CONTROL *MSOURCE* FUNCTION C1 A1
$CONTROL *MSOURCE* FUNCTION C2 A2
$CONTROL *MSOURCE* FUNCTION C3 A3
$CONTROL *MSOURCE* FUNCTION C4 A4
$CONTROL *MSOURCE* FUNCTION C5 A5
$CONTROL *MSOURCE* FUNCTION C6 A6
$CONTROL *MSOURCE* FUNCTION C7 A7
$CONTROL *MSOURCE* FUNCTION C8 A8
$CONTROL *MSOURCE* FUNCTION C9 A9
VPF 31=LINE *VT COUNT=100       (reassign 1 key (PFC1))
VPF 32=LINE *VT COUNT=200       (reassign 2 key (PFC2))
VPF 33=LINE *VT COUNT=300       (reassign 3 key (PFC3))
VPF 34=LINE *VT COUNT=400       (reassign 4 key (PFC4))
VPF 35=LINE *VT COUNT=500       (reassign 5 key (PFC5))
VPF 36=LINE *VT COUNT=600       (reassign 6 key (PFC6))
VPF 37=LINE *VT COUNT=700       (reassign 7 key (PFC7))
VPF 38=LINE *VT COUNT=800       (reassign 8 key (PFC8))
VPF 39=LINE *VT COUNT=900       (reassign 9 key (PFC9))
```

In the above example, the $CONTROL *MSOURCE* FUNCTION command redefines the Ontel Amigo numeric pad keys to Amigo program functions PFC1 through PFC9. These keys are subsequently assigned by the VPF command to the editor visual program functions VPF 31 through VPF 39. This converts the numeric-pad number keys into keys that advance the screen by a count of (key-number*100).

USING VISUAL MODE WITH THE IBM PC

The keyboard layout for the IBM PC and other IBM-compatible terminals such as the Zenith 150 is as follows:



The IBM PC Keyboard

The basic movement of the cursor is controlled by the CURSOR-RIGHT, CURSOR-LEFT, CURSOR-UP, and CURSOR-DOWN keys. These keys are labelled with arrows. The movement of the cursor may also be controlled by other keys as listed in the table below.

IBM-PC Key	*Effect*
Return	Move cursor to next line or field
CTRL-I or Tab	Move cursor to next tab position
CTRL-O or SHIFT-Tab	Move cursor to previous tab position
CTRL-F	Move cursor to next word
CTRL-R	Move cursor to previous word
CTRL-A or CTRL-Y	Move cursor to beginning of line or beginning of previous line
CTRL-Z	Move cursor to end of line or end of next line
Home	Move cursor to upper left corner
CTRL-B	Move cursor to work field

The following keys on the IBM PC have special functions in visual mode. The actions of these keys are controlled only by the terminal routines and not by the editor program itself.

IBM-PC Key	*Effect*
Ins	Enter or exit insertion mode (toggle switch)
Del or CTRL-D	Delete the character at the cursor
Backspace (←)	Delete the previous character or CTRL-H
CTRL-X	Delete the entire line
CTRL-U	Delete all characters to the right of the cursor
CTRL-V	Discard current screen changes

Along the left side of the keyboard is the program-function pad. These ten keys are called program-function keys and are labelled F1 through F10. Each program-function key has four *program functions* associated with it, one for each of the four possible key positions (unshifted-Fx, SHIFT-Fx, ALT-Fx, and CTRL-Fx). These program functions are supported by the Window program and are named PF1 through PF40.

Normally (when not in visual mode), the program-function keys execute terminal device commands. In visual mode, each of the program-function keys is assigned to an editor visual program function (VPF) for visual-mode editing. These visual program functions in turn are assigned to visual-mode commands. The default key assignments for the IBM PC terminal are as follows:

VPF	*IBM PC Key*	*Window PF*	*Default Assignment*
–2	CTRL-C	PFC	VEXIT ED_WORK
–1	CTRL-E	PFE	VEXIT
0	End	PF0	VUPDATE
1	F1	PF1	LINE *VT COUNT=–10
2	F2	PF2	VPFB
3	F3	PF3	VMEMORY 1
4	F4	PF4	LINE *VT COUNT=10
5	F5	PF5	VPFE
6	F6	PF6	VMEMORY 2
7	F7	PF7	LINE *VT COUNT=1
8	F8	PF8	VEXECUTE
9	F9	PF9	VMEMORY 3
10	F10	PF10	VINSERT 1 2
11	ALT-F1	PF11	VEXTEND
12	ALT-F2	PF12	VPREVIOUS
13	ALT-F3	PF13	LINE *VT COUNT=–10
14	ALT-F4	PF14	VPFB
15	ALT-F5	PF15	VMEMORY 1
16	ALT-F6	PF16	LINE *VT COUNT=10
17	ALT-F7	PF17	VPFE
18	ALT-F8	PF18	VMEMORY 2
19	ALT-F9	PF19	LINE *VT COUNT=1
20	ALT-F10	PF20	VEXECUTE
21	SHIFT-F1	PF21	VMEMORY 3
22	SHIFT-F2	PF22	VINSERT 1 2
23	SHIFT-F3	PF23	VEXTEND
24	SHIFT-F4	PF24	VPREVIOUS
25	SHIFT-F5	PF25	Undefined
26	SHIFT-F6	PF26	Undefined
27	SHIFT-F7	PF27	Undefined

28	SHIFT-F8	PF28	Undefined
29	SHIFT-F9	PF29	Undefined
30	SHIFT-F10	PF30	Undefined
31	CTRL-F1	PF31	Undefined
32	CTRL-F2	PF32	Undefined
33	CTRL-F3	PF33	Undefined
34	CTRL-F4	PF34	Undefined
35	CTRL-F5	PF35	Undefined
36	CTRL-F6	PF36	Undefined
37	CTRL-F7	PF37	Undefined
38	CTRL-F8	PF38	Undefined
39	CTRL-F9	PF39	Undefined
40	CTRL-F10	PF40	Undefined

Note that the default assignments for editor visual program functions (VPFs) 1 through 12 are the same as for 13 through 24. Editor VPFs 25 through 40 are initially undefined, but they may be defined by the user (see the section "Reassigning IBM PC Visual Program Functions" below).

Reassigning IBM PC Visual Program Functions

The example below illustrates how the editor visual program functions may be reassigned to make the IBM PC program-function pad more versatile.

```
VPF  1=VPFB                     (reassign F1 (PF1))
VPF  2=VPFE                     (reassign F2 (PF2))
VPF  3=VEXECUTE                 (reassign F3 (PF3))
VPF  4=VSPLIT                   (reassign F4 (PF4))
VPF  5=VINSERT 1 2              (reassign F5 (PF5))
VPF  6=VEXTEND                  (reassign F6 (PF6))
VPF  7=VMEMORY 1                (reassign F7 (PF7))
VPF  8=VMEMORY 2                (reassign F8 (PF8))
VPF  9=VMEMORY 3                (reassign F9 (PF9))
VPF 10=VPREVIOUS                (reassign F10 (PF10))
VPF 11=LINE *VT COUNT=-1        (reassign ALT-F1 (PF11))
VPF 12=LINE *VT COUNT=1         (reassign ALT-F2 (PF12))
VPF 13=LINE *VT COUNT=-10       (reassign ALT-F3 (PF13))
VPF 14=LINE *VT COUNT=10        (reassign ALT-F4 (PF14))
VPF 15=LINE *VT COUNT=-100      (reassign ALT-F5 (PF15))
VPF 16=LINE *VT COUNT=100       (reassign ALT-F6 (PF16))
VPF 17=LINE *VT COUNT=-1000     (reassign ALT-F7 (PF17))
VPF 18=LINE *VT COUNT=1000      (reassign ALT-F8 (PF18))
VPF 19=LINE *F                  (reassign ALT-F9 (PF19))
VPF 20=LINE *L                  (reassign ALT-F10 (PF20))
```

Editor visual program functions VPF 1 through VPF 12 are rearranged so that they more logically fit the IBM PC program-function pad. Editor visual program functions VPF 13 through VPF 20 are reassigned to different visual mode commands so that they do not duplicate VPF 1 through VPF 12. In this example, a selection of visual mode and normal edit commands is chosen so that the user may easily move around within the file.

As a second example, the following editor initialization file will reassign part of the program functions to visual-mode commands, part to normal editor commands, and part to edit procedures to provide more advanced capabilities for the program-function pad.

```
*
```

```
* Define a procedure to execute a %GRAB device
* command to initiate a second session
*
PROCEDURE !GRAB
UNLOCK
$CONTROL *MSOURCE* GRAB
END
*
* Define a procedure to execute a %FLIP device
* command to switch to the alternate session
*
PROCEDURE !FLIP
UNLOCK
$CONTROL *MSOURCE* FLIP
END
*
* Define a procedure to set variable A
* to beginning line for MOVE after
* moving the cursor to desired line
*
PROCEDURE !SETA
LET A=*
END
*
* Set variable B to ending line for MOVE
* after moving the cursor to desired line
*
PROCEDURE !SETB
LET B=*
END
*
* Define a procedure to clear work area
* before executing a VPFB command
*
PROCEDURE !VPFB
LET ED_WORK=''
VPFB
END
*
* Redefine some PF keys
*
VPF 21=!GRAB                    (reassign SHIFT-F1 (PF21))
VPF 22=!FLIP                    (reassign SHIFT-F2 (PF22))
VPF 23=!SETA                    (reassign SHIFT-F3 (PF23))
VPF 24=!SETB                    (reassign SHIFT-F4 (PF24))
VPF 25=MOVE A B TO *            (reassign SHIFT-F5 (PF25))
VPF 26=COMBINE * *N LEN=65      (reassign SHIFT-F6 (PF26))
VPF 27=COPY * TO *              (reassign SHIFT-F7 (PF27))
VPF 28=RENUMBER                 (reassign SHIFT-F8 (PF28))
VPF 29=!VPFB                    (reassign SHIFT-F9 (PF29))
VPF 30=VPFE                     (reassign SHIFT-F10 (PF30))
```

The above example illustrates how edit procedures may be used to execute terminal device commands and more complex editor commands. PF21 and PF22 have been reassigned to execute the %GRAB and %FLIP terminal device commands. These commands are executed via the MTS $CONTROL command (see MTS Volumes 1 and 4 for further details about issuing terminal device commands via the $CONTROL command). The edit UNLOCK command writes the current changes on the screen into the edit file and unlocks the file before the device command is executed. PF23, PF24, and PF25 have been reassigned to work in concert to easily move lines in the file from one location to another. PF23 and PF24 specify the beginning and ending bounds of the lines to be moved and PF25 moves the

text to the location specified by the current position of the cursor. PF26 has been reassigned to the COMBINE command to combine the line pointed to by the cursor with the next line; the resultant lines are limited to 65 characters. PF27 is reassigned to the COPY command to duplicate the line pointed to by the cursor. PF28 is reassigned to the RENUMBER command. PF29 has altered the VPFB command so that the work area is emptied before the insertion is performed.

Redefining the IBM PC Keyboard

The IBM PC terminal supports a total of 126 Window program functions, named PF1 through PF100 and PFA through PFZ. These program functions are assigned to the keyboard as follows:

PF1-PF10	Function keys F1-F10
PF11-PF20	ALT-Function keys F1-F10
PF21-PF30	SHIFT-Function keys F1-F10
PF31-PF40	CTRL-Function keys F1-F10
PF41-PF44	Numeric Pad keys 1-4
PF45	Del (.) key
PF46-PF49	Numeric Pad keys 6-9
PF50	Ins (0) key
PF51	CTRL-Numeric Pad key 1
PF52	CTRL-PrtSc key
PF53-PF54	CTRL-Numeric Pad keys 3-4
PF55	Non-existent
PF56-PF57	CTRL-Numeric Pad keys 6-7
PF58	CTRL-Break
PF59	CTRL-Numeric Pad key 9
PF60	Non-existent
PF61-PF70	ALT-1 through ALT-0 (nonnumeric pad)
PF71-PF72	ALT-– and ALT-=
PF73-PF98	ALT-alphabetic keys (l to r, t to b)
PF99-PF100	Non-existent
PFA-PFZ	CTRL-alphabetic keys (a to z)

The %VPF device command may be used to assign the Window program functions. The %VPF device command is given in the form

%VPF n=%! m

where “n” is the Window program function number corresponding to a particular key and “m” is an editor visual program function (VPF) number (from 0 through 99) Although the Window program function number need not be the same as the editor VPF number, it is recommended for the sake of clarity and simplicity that the Window and editor VPF numbers be the same.

Note: Window program functions PF55, PF60, PF99, PF100, and PFA through PFZ currently are not available for assignment.

The Window command %VPF? may be given to display all of the current assignments of editor VPFs to Window program functions.

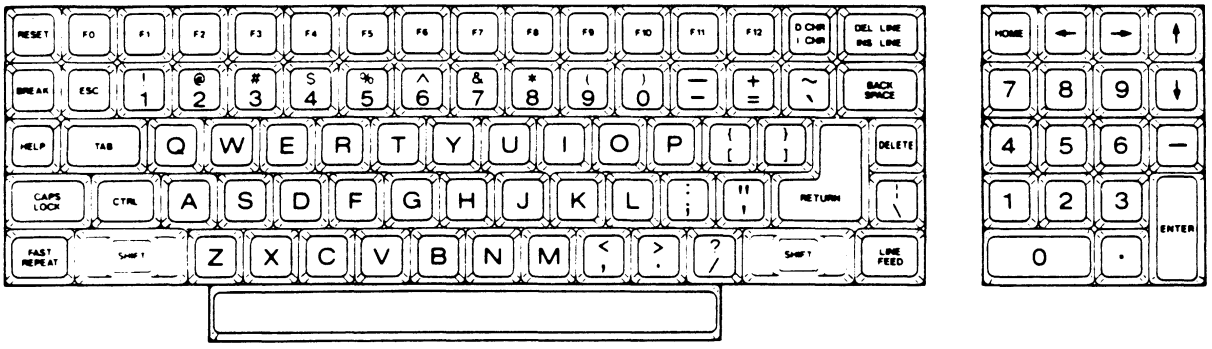
As an example of assigning the Window program functions PF41 through PF49 for use with visual mode from the IBM PC terminal, the following editor initialization file could be constructed.

```
$CONTROL *MSOURCE* VPF 41=%! 41
$CONTROL *MSOURCE* VPF 42=%! 42
$CONTROL *MSOURCE* VPF 43=%! 43
$CONTROL *MSOURCE* VPF 44=%! 44
$CONTROL *MSOURCE* VPF 45=%! 45
$CONTROL *MSOURCE* VPF 46=%! 46
$CONTROL *MSOURCE* VPF 47=%! 47
$CONTROL *MSOURCE* VPF 48=%! 48
$CONTROL *MSOURCE* VPF 49=%! 49
VPF 41=LINE *VT COUNT=100       (reassign 1 key)
VPF 42=LINE *VT COUNT=200       (reassign 2 key)
VPF 43=LINE *VT COUNT=300       (reassign 3 key)
VPF 44=LINE *VT COUNT=400       (reassign 4 key)
VPF 45=LINE *VT COUNT=500       (reassign . key)
VPF 46=LINE *VT COUNT=600       (reassign 6 key)
VPF 47=LINE *VT COUNT=700       (reassign 7 key)
VPF 48=LINE *VT COUNT=800       (reassign 8 key)
VPF 49=LINE *VT COUNT=900       (reassign 9 key)
```

In the above example, the $CONTROL *MSOURCE* VPF command assigns the IBM PC numeric pad keys to editor visual program functions VPF 41 through VPF 49. Then by issuing the editor VPF commands, the numeric-pad keys are converted into keys that advance the screen by a count of (key-number*100).

USING VISUAL MODE WITH THE ZENITH 100

The keyboard layout for the Zenith 100 terminals is as follows:



The Zenith 100 Keyboard

The basic movement of the cursor is controlled by the CURSOR-RIGHT, CURSOR-LEFT, CURSOR-UP, and CURSOR-DOWN keys. These keys are labelled with arrows. The movement of the cursor may also be controlled by other keys as listed in the table below.

Z-100 Key	*Effect*
Return	Move cursor to next line or field
CTRL-I or Tab	Move cursor to next tab position
CTRL-A or CTRL-Y	Move cursor to beginning of line or beginning of previous line
CTRL-Z	Move cursor to end of line or end of next line
Home	Move cursor to upper left corner
CTRL-B	Move cursor to work field

The following keys on the Zenith 100 have special functions in visual mode. The actions of these keys are controlled only by the terminal routines and not by the editor program itself.

Z-100 Key	*Effect*
I CHR	Enter or exit insertion mode (toggle switch)
D CHR	Delete the character at the cursor
BACKSPACE	Delete the previous character
CTRL-X or DEL LINE	Delete the entire line
CTRL-U	Delete all characters to the right of the cursor
CTRL-V	Discard current screen changes

Along the top of the keyboard is the program-function pad. These twelve keys are called program-function keys and are labelled F1 through F12. Each program-function key has two *program*

functions associated with it, one for each of the two possible key positions (unshifted-Fx and SHIFT-Fx) These program functions are supported by the Window program and are named PF1 through PF24.

Normally (when not in visual mode), the program-function keys execute terminal device commands. In visual mode, each of the program-function keys is assigned to an editor visual program function (VPF) for visual-mode editing. These visual program functions in turn are assigned to visual-mode commands. The default key assignments for the Zenith 100 terminal are as follows:

VPF	*Z-100 Key*	*Window PF*	*Default Assignment*
–2	CTRL-C	PFC	VEXIT ED_WORK
–1	CTRL-E	PFE	VEXIT
0	-	-	VUPDATE
1	F1	PF1	LINE *VT COUNT=–10
2	F2	PF2	VPFB
3	F3	PF3	VMEMORY 1
4	F4	PF4	LINE *VT COUNT=10
5	F5	PF5	VPFE
6	F6	PF6	VMEMORY 2
7	F7	PF7	LINE *VT COUNT=1
8	F8	PF8	VEXECUTE
9	F9	PF9	VMEMORY 3
10	F10	PF10	VINSERT 1 2
11	F11	PF11	VEXTEND
12	F12	PF12	VPREVIOUS
13	SHIFT-F1	PF13	LINE *VT COUNT=–10
14	SHIFT-F2	PF14	VPFB
15	SHIFT-F3	PF15	VMEMORY 1
16	SHIFT-F4	PF16	LINE *VT COUNT=10
17	SHIFT-F5	PF17	VPFE
18	SHIFT-F6	PF18	VMEMORY 2
19	SHIFT-F7	PF19	LINE *VT COUNT=1
20	SHIFT-F8	PF20	VEXECUTE
21	SHIFT-F9	PF21	VMEMORY 3
22	SHIFT-F10	PF22	VINSERT 1 2
23	SHIFT-F11	PF23	VEXTEND
24	SHIFT-F12	PF24	VPREVIOUS

Note that the default assignments for editor visual program functions (VPFs) 1 through 12 are the same as for 13 through 24.

Reassigning Zenith 100 Visual Program Functions

The example below illustrates how the editor visual program functions may be reassigned to make the Zenith 100 program-function pad more versatile.

```
VPF  1=VPFB                    (reassign F1 (PF1))
VPF  2=VPFE                    (reassign F2 (PF2))
VPF  3=VEXECUTE                (reassign F3 (PF3))
VPF  4=VSPLIT                  (reassign F4 (PF4))
VPF  5=VINSERT 1 2             (reassign F5 (PF5))
VPF  6=VEXTEND                 (reassign F6 (PF6))
VPF  7=VMEMORY 1               (reassign F7 (PF7))
```

```
VPF  8=VMEMORY 2                (reassign F8 (PF8))
VPF  9=VMEMORY 3                (reassign F9 (PF9))
VPF 10=VPREVIOUS                (reassign F10 (PF10))
VPF 11=VCURSOR WORK             (reassign F11 (PF11))
VPF 12=VUPDATE                  (reassign F12 (PF12))
VPF 13=LINE *F                  (reassign SHIFT-F1 (PF13))
VPF 14=LINE *L                  (reassign SHIFT-F2 (PF14))
VPF 15=LINE *VT COUNT=-1        (reassign SHIFT-F3 (PF15))
VPF 16=LINE *VT COUNT=1         (reassign SHIFT-F4 (PF16))
VPF 17=LINE *VT COUNT=-10       (reassign SHIFT-F5 (PF17))
VPF 18=LINE *VT COUNT=10        (reassign SHIFT-F6 (PF18))
VPF 19=LINE *VT COUNT=-100      (reassign SHIFT-F7 (PF19))
VPF 20=LINE *VT COUNT=100       (reassign SHIFT-F8 (PF20))
VPF 21=LINE *VT COUNT=-1000     (reassign SHIFT-F9 (PF21))
VPF 22=LINE *VT COUNT=1000      (reassign SHIFT-F10 (PF22))
VPF 23=VCURSOR WORK             (reassign SHIFT-F11 (PF23))
VPF 24=VUPDATE                  (reassign SHIFT-F12 (PF24))
```

Editor visual program functions VPF 1 through VPF 12 are rearranged so that they more logically fit the Zenith 100 program-function pad. Editor visual program functions VPF 13 through VPF 22 are reassigned to different visual mode commands so that they do not duplicate VPF 1 through VPF 12. In this example, a selection of visual mode and normal edit commands is chosen so that the user may easily move around within the file.

As a second example, the following editor initialization file will reassign part of the program functions to visual-mode commands, part to normal editor commands, and part to edit procedures to provide more advanced capabilities for the program-function pad.

```
*
* Define a procedure to execute a %GRAB device
* command to initiate a second session
*
PROCEDURE !GRAB
UNLOCK
$CONTROL *MSOURCE* GRAB
END
*
* Define a procedure to execute a %FLIP device
* command to switch to the alternate session
*
PROCEDURE !FLIP
UNLOCK
$CONTROL *MSOURCE* FLIP
END
*
* Define a procedure to set variable A
* to beginning line for MOVE after
* moving the cursor to desired line
*
PROCEDURE !SETA
LET A=*
END
*
* Set variable B to ending line for MOVE
* after moving the cursor to desired line
*
PROCEDURE !SETB
LET B=*
END
*
```

```
* Define a procedure to clear work area
* before executing a VPFB command
*
PROCEDURE !VPFB
LET ED_WORK=''
VPFB
END
*
* Redefine some PF keys
*
VPF 15=!GRAB                    (reassign SHIFT-F3 (PF15))
VPF 16=!FLIP                    (reassign SHIFT-F4 (PF16))
VPF 17=!SETA                    (reassign SHIFT-F5 (PF17))
VPF 18=!SETB                    (reassign SHIFT-F6 (PF18))
VPF 19=MOVE A B TO *            (reassign SHIFT-F7 (PF19))
VPF 20=COMBINE * *N LEN=65      (reassign SHIFT-F8 (PF20))
VPF 21=COPY * TO *              (reassign SHIFT-F9 (PF21))
VPF 22=RENUMBER                 (reassign SHIFT-F10 (PF22))
VPF 23=!VPFB                    (reassign SHIFT-F11 (PF23))
VPF 24=VPFE                     (reassign SHIFT-F12 (PF24))
```

The above example illustrates how edit procedures may be used to execute terminal device commands and more complex editor commands. PF15 and PF16 have been reassigned to execute the %GRAB and %FLIP terminal device commands. These commands are executed via the MTS $CONTROL command (see MTS Volumes 1 and 4 for further details about issuing terminal device commands via the $CONTROL command). The edit UNLOCK command writes the current changes on the screen into the edit file and unlocks the file before the device command is executed. PF17, PF18, and PF19 have been reassigned to work in concert to easily move lines in the file from one location to another. PF17 and PF18 specify the beginning and ending bounds of the lines to be moved and PF19 moves the text to the location specified by the current position of the cursor. PF20 has been reassigned to the COMBINE command to combine the line pointed to by the cursor with the next line; the resultant lines are limited to 65 characters. PF21 is reassigned to the COPY command to duplicate the line pointed to by the cursor. PF22 is reassigned to the RENUMBER command. PF23 has altered the VPFB command so that the work area is emptied before the insertion is performed.

Redefining the Zenith 100 Keyboard

The Zenith 100 terminal supports a total of 65 Window program functions, named PF1 through PF39 and PFA through PFZ. These program functions are assigned to the keyboard as follows:

PF1-PF12	Function keys F1-F12
PF13-PF24	SHIFT-Function keys F1-F12
PF25	Function key 0
PF26	SHIFT-Function key 0
PF27	SHIFT-BREAK
PF28	HELP
PF29	Non-existent
PF30-PF39	Numeric Pad keys 0-9
PF31	INS LINE
PF32	Down Arrow (↓)
PF33	DEL LINE
PF34	Left Arrow (←)
PF35	HOME
PF36	Right Arrow (→)

PF37	I CHR
PF38	Up Arrow (↑)
PF39	D CHR
PFA-PFZ	CTRL-Alphabetic keys (a-z)

The %VPF device command may be used to assign the Window program functions. The %VPF device command is given in the form

%VPF n=%! m

where “n” is the Window program function number corresponding to a particular key and “m” is an editor visual program function (VPF) number (from 0 through 99) Although the Window program function number need not be the same as the editor VPF number, it is recommended for the sake of clarity and simplicity that the Window and editor VPF numbers be the same.

Note: Window program functions PF29, and PFA through PFZ currently are not available for assignment.

As an example of assigning the Window program functions PF31 through PF39 for use with visual mode from the Zenith 100 terminal, the following editor initialization file could be constructed.

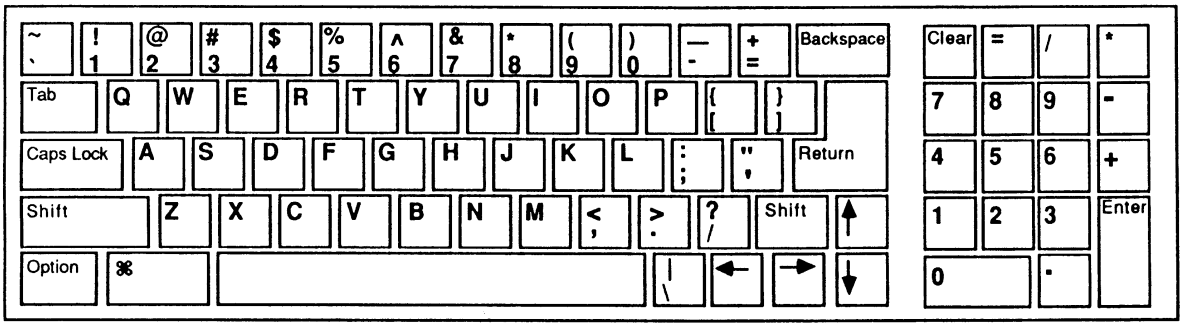
```
$CONTROL *MSOURCE* VPF 31=%! 31
$CONTROL *MSOURCE* VPF 32=%! 32
$CONTROL *MSOURCE* VPF 33=%! 33
$CONTROL *MSOURCE* VPF 34=%! 34
$CONTROL *MSOURCE* VPF 35=%! 35
$CONTROL *MSOURCE* VPF 36=%! 36
$CONTROL *MSOURCE* VPF 37=%! 37
$CONTROL *MSOURCE* VPF 38=%! 38
$CONTROL *MSOURCE* VPF 39=%! 39
VPF 31=LINE *VT COUNT=100       (reassign 1 key)
VPF 32=LINE *VT COUNT=200       (reassign 2 key)
VPF 33=LINE *VT COUNT=300       (reassign 3 key)
VPF 34=LINE *VT COUNT=400       (reassign 4 key)
VPF 35=LINE *VT COUNT=500       (reassign 5 key)
VPF 36=LINE *VT COUNT=600       (reassign 6 key)
VPF 37=LINE *VT COUNT=700       (reassign 7 key)
VPF 38=LINE *VT COUNT=800       (reassign 8 key)
VPF 39=LINE *VT COUNT=900       (reassign 9 key)
```

In the above example, the $CONTROL *MSOURCE* VPF command assigns the Zenith 100 numeric pad keys to editor visual program functions VPF 31 through VPF 39. Then by issuing the editor VPF commands, the numeric-pad keys are converted into keys that advance the screen by a count of (key-number*100).

The Window command %VPF? may be given to display all of the current assignments of editor VPFs to Window program functions.

USING VISUAL MODE WITH THE APPLE MACINTOSH

The keyboard layout for the Apple Macintosh terminal is as follows:



The Apple Macintosh Keyboard

The basic movement of the cursor is controlled by pressing the CTRL key (labelled with the quadrille symbol) in conjunction with the CURSOR-RIGHT, CURSOR-LEFT, CURSOR-UP, and CURSOR-DOWN keys (labelled with arrows). The cursor also may be moved by pressing the OPTION key and using the mouse to click the cursor to the desired location. The movement of the cursor may also be controlled by other keys as listed in the table below.

Mac Key	*Effect*
CTRL-H or Backsp	Move cursor to previous character
CTRL-M or Return	Move cursor to beginning of next line
CTRL-I or Tab	Move cursor to next tab position
CTRL-B	Move cursor to beginning of line or beginning of previous line
CTRL-K	Move cursor to beginning of previous line
CTRL-N	Move cursor to last column of line
CTRL-Z	Move cursor to end of line or end of next line
CTRL-T	Move cursor to upper left corner
CTRL-V	Move cursor to work field

The following keys on the Macintosh have special functions in visual mode. The actions of these keys are controlled only by the terminal routines and not by the editor program itself.

Mac Key	*Effect*
CTRL-A	Enter or exit insertion mode (toggle switch)
CTRL-D	Delete the character at the cursor
CTRL-W	Delete the word at the cursor
CTRL-X	Delete all characters to the right of the cursor
CTRL-G	Discard current line changes
CTRL-L	Discard current screen changes
CTRL-R	Rewrite the screen (with changes)

The Macintosh control program assigns Macintosh program functions (PFs) to several of the keys, primarily on the numeric pad keyboard. These are named PF0 through PF17. These in turn are assigned to editor program functions VPF 0 through VPF 17. The default key assignments for the Macintosh terminal are as follows:

VPF	*Mac Key*	*Mac PF*	*Default Assignment*
–2	CTRL-C	-	VEXIT ED_WORK
–1	CTRL-E	-	VEXIT
0	Numpad 1	PF0	VUPDATE
1	Numpad 1	PF1	LINE *VT COUNT=–10
2	Numpad 2	PF2	VPFB
3	Numpad 3	PF3	VMEMORY 1
4	Numpad 4	PF4	LINE *VT COUNT=10
5	Numpad 5	PF5	VPFE
6	Numpad 6	PF6	VMEMORY 2
7	Numpad 7	PF7	LINE *VT COUNT=1
8	Numpad 8	PF8	VEXECUTE
9	Numpad 9	PF9	VMEMORY 3
10	Numpad CLEAR	PF10	VINSERT 1 2
11	Numpad –	PF11	VEXTEND
12	Numpad +	PF12	VPREVIOUS
13	Numpad *	PF13	LINE *VT COUNT=–10
14	Numpad /	PF14	VPFB
15	Numpad ,	PF15	VMEMORY 1
16	Numpad ENTER	PF16	LINE *VT COUNT=10
17	Numpad .	PF17	VPFE

Note that the default assignments for editor visual program functions (VPFs) 1 through 5 are the same as for 13 through 17.

Reassigning Macintosh Visual Program Functions

The example below illustrates how the editor visual program functions may be reassigned to make the Macintosh numeric pad more versatile.

```
VPF  0=LINE *VT COUNT=-10        (reassign 0 key)
VPF  1=LINE *VT COUNT=1          (reassign 1 key)
VPF  2=LINE *VT COUNT=10         (reassign 2 key)
VPF  3=LINE *VT COUNT=100        (reassign 3 key)
VPF  4=VINSERT 1 2               (reassign 4 key)
VPF  5=VEXTEND                   (reassign 5 key)
VPF  6=VSPLIT                    (reassign 6 key)
```

```
VPF  7=VPFB                      (reassign 7 key)
VPF  8=VPFE                      (reassign 8 key)
VPF  9=VEXECUTE                  (reassign 9 key)
VPF 10=VMEMORY 1                 (reassign CLEAR key)
VPF 11=VMEMORY 2                 (reassign - key)
VPF 12=VMEMORY 3                 (reassign + key)
VPF 13=VPREVIOUS                 (reassign * key)
VPF 14=LINE *F                   (reassign / key)
VPF 15=LINE *L                   (reassign , key)
VPF 16=VUPDATE                   (reassign ENTER)
VPF 17=LINE *VT COUNT=-100       (reassign . key)
```

Editor visual program functions VPF 1 through VPF 17 are rearranged so that they more logically fit the Macintosh numeric pad. Editor visual program functions VPF 13 through VPF 17 are reassigned to different visual mode commands so that they do not duplicate VPF 1 through VPF 5. In this example, a selection of visual mode and normal edit commands is chosen so that the user may easily move around within the file.

As a second example, the following editor initialization file will reassign part of the program functions to visual-mode commands, part to normal editor commands, and part to edit procedures to provide more advanced capabilities for the program-function pad.

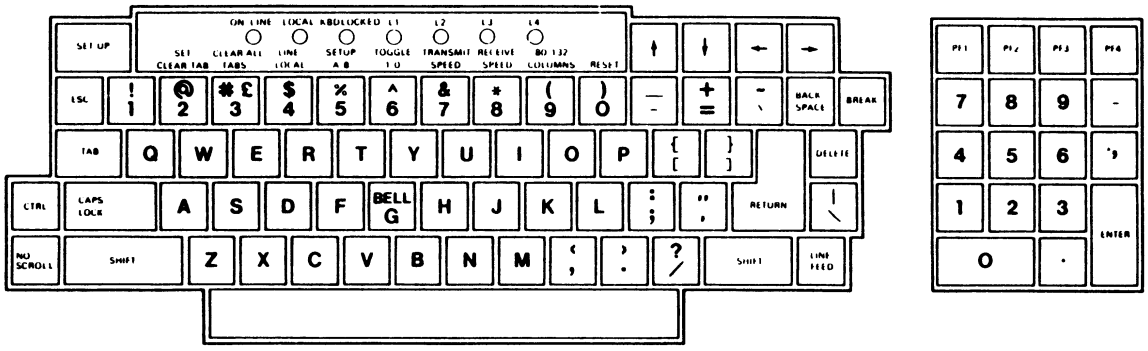
```
*
* Define a procedure to set variable A
* to beginning line for MOVE after
* moving the cursor to desired line
*
PROCEDURE !SETA
LET A=*
END
*
* Set variable B to ending line for MOVE
* after moving the cursor to desired line
*
PROCEDURE !SETB
LET B=*
END
*
* Define a procedure to clear work area
* before executing a VPFB command
*
PROCEDURE !VPFB
LET ED_WORK=''
VPFB
END
*
* Redefine some PF keys
*
VPF 10=!SETA                     (reassign CLEAR key (PF10))
VPF 11=!SETB                     (reassign - key (PF11))
VPF 12=MOVE A B TO *             (reassign + key (PF12))
VPF 13=COMBINE * *N LEN=65       (reassign * key (PF13))
VPF  7=!VPFB                     (reassign 7 key (PF7))
```

The above example illustrates how edit procedures may be used to execute terminal device commands and more complex editor commands. PF10, PF11, and PF12 have been reassigned to work in concert to easily move lines in the file from one location to another. PF10 and PF11 specify the beginning and ending bounds of the lines to be moved and PF12 moves the text to the location specified by the current

position of the cursor. PF13 has been reassigned to the COMBINE command to combine the line pointed to by the cursor with the next line; the resultant lines are limited to 65 characters. PF7 has altered the VPFB command so that the work area is emptied before the insertion is performed.

USING VISUAL MODE WITH THE DEC VT100 TERMINAL

The procedures for using visual mode with DEC VT100 terminals is similar to those for using the Apple Macintosh terminal. The keyboard layout for the DEC VT100 is as follows:



The DEC VT100 Keyboard

The basic movement of the cursor is controlled by pressing the the CURSOR-RIGHT, CURSOR-LEFT, CURSOR-UP, and CURSOR-DOWN keys (labelled with arrows). The movement of the cursor may also be controlled by other keys as listed in the table below.

VT100 Key	*Effect*
CTRL-H or Backsp	Move cursor to previous character
CTRL-M or Return	Move cursor to beginning of next line
CTRL-I or Tab	Move cursor to next tab position
CTRL-B	Move cursor to beginning of line or beginning of previous line
CTRL-K	Move cursor to beginning of previous line
CTRL-N	Move cursor to last column of line
CTRL-Z	Move cursor to end of line or end of next line
CTRL-T	Move cursor to upper left corner
CTRL-V	Move cursor to work field

The following keys on the DEC VT100 have special functions in visual mode. The actions of these keys are controlled only by the terminal routines and not by the editor program itself.

VT100 Key	*Effect*
CTRL-A	Enter or exit insertion mode (toggle switch)
CTRL-D	Delete the character at the cursor
CTRL-W	Delete the word at the cursor
CTRL-X	Delete all characters to the right of the cursor
CTRL-G	Discard current line changes
CTRL-L	Discard current screen changes
CTRL-R	Rewrite the screen (with changes)

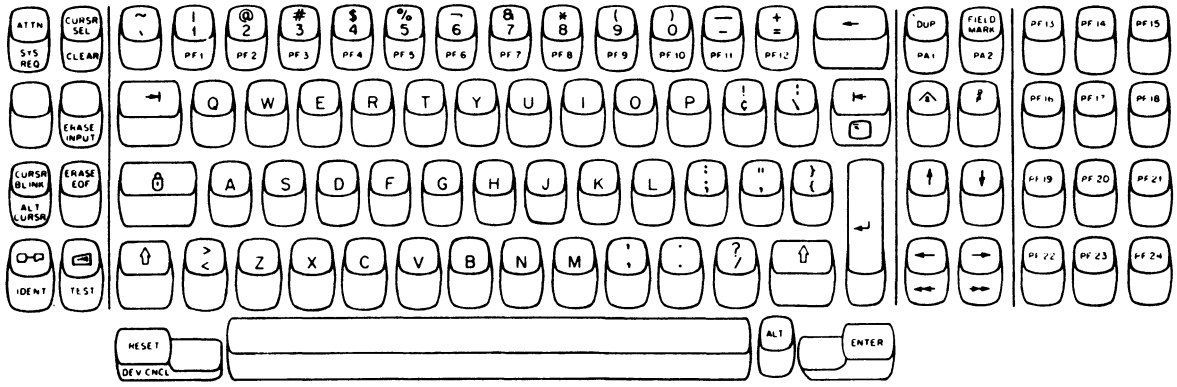
The VT-100 control program assigns VT100 program functions (PFs) to several of the keys, primarily on the numeric pad keyboard. These are named PF0 through PF17. These in turn are assigned to editor program functions VPF 0 through VPF 17. The default key assignments for the VT100 terminal are as follows:

VPF	*VT100 Key*	*VT100 PF*	*Default Assignment*
–2	CTRL-C	-	VEXIT ED_WORK
–1	CTRL-E	-	VEXIT
0	Numpad 0	PF0	VUPDATE
1	Numpad 1	PF1	LINE *VT COUNT=–10
2	Numpad 2	PF2	VPFB
3	Numpad 3	PF3	VMEMORY 1
4	Numpad 4	PF4	LINE *VT COUNT=10
5	Numpad 5	PF5	VPFE
6	Numpad 6	PF6	VMEMORY 2
7	Numpad 7	PF7	LINE *VT COUNT=1
8	Numpad 8	PF8	VEXECUTE
9	Numpad 9	PF9	VMEMORY 3
10	Numpad PF1	PF10	VINSERT 1 2
11	Numpad PF2	PF11	VEXTEND
12	Numpad PF3	PF12	VPREVIOUS
13	Numpad PF4	PF13	LINE *VT COUNT=–10
14	Numpad –	PF14	VPFB
15	Numpad ,	PF15	VMEMORY 1
16	Numpad ENTER	PF16	LINE *VT COUNT=10
17	Numpad .	PF17	VPFE

Note that the default assignments for editor visual program functions (VPFs) 1 through 5 are the same as for 13 through 17.

USING VISUAL MODE WITH THE IBM 3278 TERMINAL

The keyboard layout for the IBM 3278 terminal is as follows:



The IBM 3278 Keyboard

The movement of the cursor is controlled by the CURSOR-RIGHT, CURSOR-LEFT, CURSOR-UP, and CURSOR-DOWN keys. These keys are labelled with arrows. Movement of the cursor is also controlled by the TAB, BACKTAB, and RETURN keys.

The following keys on the IBM 3278 have special functions in visual mode. The actions of these keys are controlled only by the terminal routines and not by the editor program itself.

3278 Key	*Effect*
Insert (a)	Enter insertion mode
RESET	Exit insertion mode
Delete (a)	Delete the character at the cursor
ERASE EOF	Delete all characters to the right of the cursor
CLEAR	Discard all current screen changes

Across the top and to the right side of the keyboard are the program-function pads. These keys are labelled PF1 through PF12 and PF13 through PF24. The PF1 through PF12 keys must be pressed in conjunction with the ALT key. Each program-function key executes a *program function*.

Normally (when not in visual mode), the program functions are defined as IBM 3278 device commands. In visual mode, each of the programs function is assigned to an editor visual program function (VPF) for visual-mode editing. These visual program functions in turn are defined as visual-mode commands. The default key assignments for the IBM 3278 terminal are as follows:

VPF	*3278 Key*	*Default Assignment*
–3	SYS REQ	VCURSOR WORK
–2	PA2	VEXIT ED_WORK
–1	PA1	VEXIT
0	ENTER	VUPDATE
1	PF1	LINE *VT COUNT=–10
2	PF2	VPFB
3	PF3	VMEMORY 1
4	PF4	LINE *VT COUNT=10
5	PF5	VPFE
6	PF6	VMEMORY 2
7	PF7	LINE *VT COUNT=1
8	PF8	VEXECUTE
9	PF9	VMEMORY 3
10	PF10	VINSERT 1 2
11	PF11	VEXTEND
12	PF12	VPREVIOUS
13	PF13	LINE *VT COUNT=–10
14	PF14	VPFB
15	PF15	VMEMORY 1
16	PF16	LINE *VT COUNT=10
17	PF17	VPFE
18	PF18	VMEMORY 2
19	PF19	LINE *VT COUNT=1
20	PF20	VEXECUTE
21	PF21	VMEMORY 3
22	PF22	VINSERT 1 2
23	PF23	VEXTEND
24	PF24	VPREVIOUS

Note that the default assignments for editor visual program functions (VPFs) 1 through 12 are the same as for 13 through 24. The editor VPFs may be redefined by using the editor VPF command.

SETTING THE VISUAL-MODE RULER

Normally, each time the visual-screen is displayed, the last line of the screen shows the current contents of the predefined variable ED_VRULER. By default, this ruler contains column markers that may be used for aligning text on the screen.

The predefined variable ED_STATUS contains a string expression that is evaluated each time the visual-mode screen is displayed. The initial value is copied from the read-only predefined variable ED_STATUS_DEF. The string result of this evaluation is displayed in the screen position normally occupied by the visual-mode ruler (ED_VRULER) provided that this ruler has been set to the null string by the edit command

LET ED_VRULER=''

When non-null, the visual-mode ruler takes precedence over the contents of ED_STATUS. The contents of ED_STATUS may be tailored to fit the user's needs; however, the righthand assignment to ED_STATUS must be a string in order to prevent evaluation before the assignment; for example,

```
LET ED_STATUS='"Cursor at line: " ED_CURSOR_LINE'
```

The default visual-mode ruler may be restored by giving the edit command

LET ED_VRULER=ED_VRULER_DEF

VISUAL-MODE COMMANDS

Several special visual-mode commands are available to video-screen terminal users while editing in visual mode. These commands perform such functions as rapidly moving the cursor on the screen, executing commands from the work field, extending screen lines, inserting new screen lines, and splitting lines into shorter segments.

The visual-mode commands normally are executed either

(1) from the work field by depressing an appropriately assigned program-function key (see below), or

(2) by depressing a program-function key that has been assigned as a visual-mode command.

Executing Commands from the Work Field

A visual-mode command may be executed from the work field by inserting the command into the work field and then executing the VEXECUTE visual mode command. By default, this command is assigned to editor visual program functions VPF 8 and VPF 20.

For example with the Ontel terminal, the VSPLIT command may be used to split a line into two shorter lines. First, the cursor must be moved to the work field, either manually by moving the cursor with the cursor-positioning keys or directly by depressing the SHIFT-HOME key. After the VSPLIT command has been entered into the work field, the cursor is moved to the line to be split and positioned at the column where the split is to be made. Finally, either the CTRL-8 or PF8 key is pressed.

Executing Commands via Program-Function Keys

Many terminals have a set of program-function and control keys that are specially configured for use in visual mode. Details on using these program-function keys are given with the individual terminal descriptions above.

Visual-Mode Command Summary

The following commands affect the operation of the editor in visual mode. The same notation is used for the visual-mode commands as is used for the edit command definitions.

VCURSOR LINE [linenumber [{col | ZIP}]]
VCURSOR WORK [{col | ZIP}]
VCURSOR HOME
VCURSOR COST
VEXECUTE
VEXIT [strexp]
VEXTEND [n]
VINSERT [init] [mult]
VMARK [{RESET | FIRST}]
VMEMORY [n]
VPF {SAVE | RESTORE} pfname
VPF {n | ATTN | RETURN}=command
VPFB
VPFE

VPREVIOUS
VSPLIT
VTEST
VUPDATE

VCURSOR

Visual Edit Command Description

Purpose: To set the position of the visual-mode cursor.

Prototype:
(a) VCURSOR LINE [linenumber [{col | ZIP}]]
(b) VCURSOR WORK [{col | ZIP}]
(c) VCURSOR HOME
(d) VCURSOR COST

where

"linenumber" is the line number of the line the cursor is to appear in.

"col" is the column number within the line or work area the cursor is to appear in. The first character of a line is column 1.

Action: If prototype (a) is specified, the cursor is placed on the line with line number "linenumber". The cursor is placed either at the column given by "col" or after the last character in the line, if ZIP is given. If neither "col" nor ZIP is specified, the cursor is placed at the beginning of the line. The line number is defaulted to the current line (*), if omitted. If the specified line does not exist, the cursor is placed on the next previous line to the specified line, if it appears on the screen; otherwise, the cursor is placed in the home position at the top of the screen.

If prototype (b) is specified, the cursor is placed in the work area. The cursor is placed either at the column given by "col" or after the last character in the line, if ZIP is given. If neither "col" nor ZIP is specified, the cursor is placed at the beginning of the line.

If prototype (c) is specified, the cursor is placed in the home position, which is the first column of data of the first line of the screen (the top line).

If prototype (d) is specified, the cursor is positioned at the beginning of the cost field.

Conditions: SUCCESS if no error messages are generated.

Example:

```
VCURSOR LINE 312.2 19
```

The above example positions the cursor to line 312.2 at data column 19.

VEXECUTE

Visual Edit Command Description

Purpose: To execute the contents of the work area as an edit command.

Prototype: VEXECUTE

Action: The VEXECUTE command executes the contents of the work area as an edit command. By default, the execution of the command is not verified in the conversation buffer unless the @VMV modifier is appended to the command or the editor VMV option is ON.

If the command is not a valid edit command, an error message is printed in the ruler area.

The default settings of VPF 8 and VPF 20 execute the VEXECUTE command.

When VEXECUTE is used in an edit procedure, the execution of the work area is delayed until all of the commands of the procedure have been executed so that edit-command messages such as “Search failed” and “File does not exist” may be printed (these messages are suppressed when they occur within the procedure). The EXECUTE ED_WORK command may be used in an edit procedure to execute the work area immediately.

Conditions: SUCCESS if no error messages are generated.

VEXIT

Visual Edit Command Description

Purpose: To exit visual mode and optionally print a string expression on the terminal screen (conversation buffer).

Prototype: VEXIT [strexp]

where

> "strexp" is an editor string expression which is to be printed on the terminal screen after exiting visual mode. The string expression may contain or consist solely of an editor variable, in which case the contents of the variable is inserted into the string.

Action: The string expression specified by "strexp" is evaluated and printed in the terminal conversation buffer and visual mode is exited. By printing the string in the conversation buffer, it will appear on the screen after visual mode is exited.

If "strexp" is omitted, visual mode is exited without printing a string in the conversation buffer.

The default setting for VPF –2 is the command VEXIT ED_WORK, where ED_WORK is an editor variable containing the current contents of the work area.

The file buffers in memory are written to disk. This is the same as if the CLOSE command were executed.

Conditions: SUCCESS if no error messages are generated.

Example:

```
VEXIT 'Finished with source update'
```

> Visual mode is exited and the string "Finished with source update" is printed on the user's terminal.

VEXTEND

Visual Edit Command Description

Purpose: To extend the line containing the cursor or the work area by one screen line.

Prototype: VEXTEND [n]

Action: If the cursor is on a line on the screen, the line is extended with null characters into the next "n" screen lines. If the cursor is in the work area, the work area is extended by an additional screen line of nulls. Note that if there is already an additional screen line of nulls allotted to the line the cursor is on, this command has no effect unless a window has been set which causes parts of lines not to be wrapped.

The VEXTEND command toggles between wrapping and not wrapping a line which extends beyond the view of the window. When VEXTEND is first executed, an extension of the line will appear and the line remains unwrapped. If the user types in the extension area, the data will be inserted at those column ranges and any hidden data in the line (which cannot be seen due to the window) will follow the newly inserted data. Executing a second VEXTEND command will cause the line to wrap so that data may be appended to the line.

The line may be further extended after one or more nonnull characters (including blanks) are inserted into the first extension by again issuing the VEXTEND command.

A line that has been extended by the VEXTEND command will remain extended until one of the following conditions occurs:

(1) A VUPDATE command is executed (the default for the ENTER key).

(2) A VEXIT command is executed.

(3) A VEXTEND command is executed on a different line.

(4) An EDIT, RENUMBER, or SPREAD command is executed.

The default settings of VPF 11 and VPF 23 execute the VEXTEND command.

Conditions: SUCCESS if no error messages are generated.

VINSERT

Visual Edit Command Description

Purpose: To insert null lines on the screen after the line containing the cursor.

Prototype: VINSERT [init] [mult]

where

"init" is the number of null lines to insert after the current line the first time VINSERT is executed.

"mult" is the multiplier for the number of null lines for subsequent executions of VINSERT.

Action: If VINSERT is executed and the cursor is not on a null line, "init" null lines are inserted after the line containing the cursor, and the cursor is placed at the first character of the first null line. These null lines have a "." in the line-number field instead of the line number, so that they can be differentiated from lines that are entirely blank. If VINSERT was previously executed and the cursor is already on a null line, the null lines remaining on the screen (call their number "n") are replaced by ("n" * "mult") null lines. For example, suppose that VPF 22 is assigned VINSERT 3 4. Then if VPF 22 is executed, three null lines will appear on the screen. Then if the first null line is filled with text and VPF 22 is again executed, the remaining two null lines will be replaced by eight null lines (2 * 4).

"init" defaults to 1 and "mult" defaults to 2.

Null lines added with the VINSERT command will continue to be displayed until one of the following conditions occurs:

(1) A VUPDATE command is executed.

(2) A VEXIT command is executed.

(3) A VINSERT command is executed with the cursor outside the current insertion area.

(4) An EDIT or SPREAD command is executed.

The default settings of VPF 10 and VPF 22 execute the command VINSERT 1 2.

Modifiers:

@OPL If @OPL is specified, the insertion space will disappear when any program-function key is pressed. The default is @NOPL.

@PLN If @PLN is specified, the inserted null lines will be prefixed with their line numbers. The default is @NPLN.

Conditions: SUCCESS if no error messages are generated.

Example:

```
VINSERT 4 3
```

The first time this command is executed, 4 null lines are added after the line containing the cursor. If it is executed again without moving the cursor (or adding data to the null line), 12 null lines will appear after the line containing the cursor; if it is executed again, 36 null lines will appear, and so on.

VMARK

Visual Edit Command Description

Purpose: To define a region in visual mode.

Prototype: VMARK [{RESET | FIRST}]

Action: The first time VMARK is executed, the line containing the cursor will be marked with an "F" in the first column to indicate the first line in the VMARK region. The next time VMARK is executed, the line containing the cursor will be marked with an "L" to indicate that last line in the region.

The marked lines delineate a region with the name /VMARK which can be used as an editor region.

There can be only one VMARK region defined at a time. Before defining a new VMARK region, the VMARK RESET command must be given to clear the previous region.

The VMARK FIRST command is equivalent to issuing the sequence VMARK RESET, VMARK.

Conditions: None

VMEMORY

Visual Edit Command Description

Purpose: To "remember" a file position for subsequent restoration.

Prototype: VMEMORY [n]

where

"n" is an integer from 1 to 6 specifying the memory position.

Action: The VMEMORY command "remembers" a file position so that it may be subsequently restored.

There are six memory positions available, numbered 1 through 6. The initial definitions of the memory positions are as follows:

Position 1:	The first line of the file (*F)
Position 2:	Line 1000
Position 3:	The last line of the file (*L)
Position 4:	Line 1500
Position 5:	Line 2000
Position 6:	Line 3000

A memory position may be reset by issuing the VMEMORY command twice in succession without any intervening commands or movement of the cursor. To return to a "remembered" position, the VMEMORY command is issued once. For example, the command sequence

```
VMEMORY 1
VMEMORY 1
```

will set memory position 1 to the current position on the screen. Subsequently issuing the command

```
VMEMORY 1
```

will return the screen to the position "remembered" when the two VMEMORY commands were issued.

If the VMEMORY command is given without a memory-position number, the previous file position is restored. This is the same as the VPREVIOUS command (see the VPREVIOUS command description).

The default settings of several editor VPFs also perform the same function. They are

VPFs 3 and 15:	VMEMORY 1
VPFs 6 and 18:	VMEMORY 2
VPFs 9 and 21:	VMEMORY 3
VPFs 12 and 24:	VMEMORY (same as VPREVIOUS)

Conditions: SUCCESS if no error messages are generated.

VPF

Visual Edit Command Description

Purpose: To save or restore the settings of the editor visual program functions as a cluster, or to assign an edit command to an editor visual program function.

Prototype:
(a) VPF SAVE pfname
(b) VPF RESTORE pfname
(c) VPF {n | ATTN | RETURN}=command

where

"pfname" is the name for the program-function cluster (1 to 7 characters).

"n" is an editor visual program-function number.

"command" is an edit command to be assigned to the editor visual program function. The command starts immediately after the "=" sign.

Action: If prototype (a) is specified, the current settings of the editor visual program functions are saved as a cluster named "pfname". The current settings of the visual program functions remain the same. A saved program-function cluster may be restored using prototype (b).

If prototype (c) is specified, an edit command is assigned to the editor visual program function "n", or to the ATTN or RETURN keys. The number of visual program functions is dependent on the type of terminal being used (see the individual terminal descriptions for visual mode earlier). The default editor visual program-function assignments are given below.

The default editor visual program-function definitions are always available in the cluster DEFAULT and may be restored using prototype (b), e.g., VPF RESTORE DEFAULT.

Conditions: SUCCESS for prototypes (a) and (b) if the cluster is saved or restored; SUCCESS for prototype (c) if the specified key is set.

Visual Prog. Func.	*Default Assignment*
–1	VEXIT
–2	VEXIT ED_WORK
–3	VCURSOR WORK
0	VUPDATE
1	LINE *VT COUNT=–10
2	VPFB
3	VMEMORY 1
4	LINE *VT COUNT=10
5	VPFE
6	VMEMORY 2
7	LINE *VT COUNT=1
8	VEXECUTE
9	VMEMORY 3
10	VINSERT 1 2
11	VEXTEND
12	VPREVIOUS
13	LINE *VT COUNT=–10
14	VPFB
15	VMEMORY 1
16	LINE *VT COUNT=10
17	VPFE
18	VMEMORY 2
19	LINE *VT COUNT=1
20	VEXECUTE
21	VMEMORY 3
22	VINSERT 1 2
23	VEXTEND
24	VPREVIOUS
25-100	Undefined

VPFB

Visual Edit Command Description

Purpose: To append the remainder of the line containing the cursor to the end of the work area.

Prototype: VPFB

Action: The VPFB command appends the remainder of the line containing the cursor to the end of the work area. The remainder of the line starts at the cursor position.

If the cursor is in the work area, this command has no effect.

The default settings of VPF 2 and VPF 14 execute the VPFB command.

Conditions: SUCCESS if no error messages are generated.

VPFE

Visual Edit Command Description

Purpose: To insert the contents of the work area before the character in a line that contains the cursor.

Prototype: VPFE

Action: The VPFE command inserts the contents of the work area into the line that contains the cursor. The insertion is made at the position pointed to by the cursor so that the characters before the cursor compose the start of the line, the work area composes the middle of the line, and the characters from the cursor onward compose the end of the line.

If the cursor is in the work area, this command has no effect.

The default settings of VPF 5 and VPF 17 execute the VPFE command.

Conditions: SUCCESS if no error messages are generated.

VPREVIOUS

Visual Edit Command Description

Purpose: To move the screen to the "previous" file position.

Prototype: VPREVIOUS

Action: The VPREVIOUS command moves the screen to the "previous" file position, which is either

(1) the file position before the last VMEMORY command was issued,

(2) the file position before the line number in the visual-mode command field was set,

(3) the file position when visual mode was last exited, or

(4) the file position before an edit command was executed via the VEXECUTE command or the character "C" in the cost field.

The default settings of VPF 12 and VPF 24 execute the VPREVIOUS command.

Conditions: SUCCESS if no error messages are generated.

VSPLIT

Visual Edit Command Description

Purpose:	To split a line starting at the character the cursor is on.
Prototype:	VSPLIT
Action:	The VSPLIT command splits the line containing the cursor into two lines. The split is made so that the characters before the cursor remain in the current line, and a new line is inserted after the current line. The new line consists of the characters starting at the cursor through the end of the line.
Conditions:	SUCCESS if no error messages are generated.

VTEST

Edit Command Description

Purpose: To test if the editor is currently executing in visual mode.

Prototype: VTEST

Action: The execution of this command will set the condition to SUCCESS if the editor is currently executing in visual mode; otherwise, the condition is set to FAILURE.

Modifiers: None.

Conditions: SUCCESS if in visual mode; FAILURE otherwise.

Example:

```
PROCEDURE !SET_WINDOW
VTEST
GOTO END ON FAILURE
WINDOW LEFT
END
```

In the above example, the procedure uses the VTEST command to determine whether a window is to be set on the file. If the editor is in visual mode, a window will be set to be the size of the display screen. This has the effect of causing lines longer than the width of the screen to be extended to the right of the screen instead of being wrapped. In line mode, the window setting will not be changed.

VUPDATE

Visual Edit Command Description

Purpose: To update the contents of the edit file in accordance with the changes made to the visual-mode screen.

Prototype: VUPDATE

Action: The VUPDATE command updates the contents of the edit file in accordance with the changes made to the visual-mode screen. The screen is then rewritten using the updated contents of the edit file. The file buffers in memory are written to disk. This is the same as if the CLOSE command were executed.

The default setting of VPF 0 executes the VUPDATE command.

Conditions: SUCCESS if no error messages are generated.

PATTERN MATCHING

The pattern-matching facility of the editor allows the user to perform more sophisticated operations on a file than mere string searching or replacement. The pattern-matching facility is based on the algorithms for pattern matching that are used by the SNOBOL4 programming language. Currently, pattern matching may be done by the ALTER, CHANGE, MATCH, and SCAN commands. The PATTERNS option must be set to ON to enable the pattern-matching facility, e.g.,

```
SET PATTERNS=ON
```

In simple terms, pattern matching is the process of examining a subject for a substring or set of substrings specified by a pattern structure. The subject is a line or set of lines in the file being edited. The pattern structure is a set of substrings against which the subject is to be examined; each substring in the pattern is called a pattern element. For example, if the first line (line 1) of the file contains the line

```
THIS IS THE FIRST LINE OF MY FILE.
```

the command

```
SCAN 1 "FIRST"
```

will examine the first line for the string "FIRST". In this case, the subject is line 1 of the file and the pattern is composed of a single element, the string "FIRST". If the line contains the string, the pattern match is successful. If the line does not contain the string, the match fails. The above pattern match succeeds since the string "FIRST" is contained in the first line of the file. However, if the command

```
SCAN 1 "SECOND"
```

were executed, the pattern match would fail since the line does not contain the string "SECOND". The above two patterns are each composed of a single element, i.e., a simple string.

The concatenation and alternation operators may be used to construct more complicated patterns composed of several elements. The concatenation operator is indicated by a blank between pattern elements. Concatenation is used to combine several pattern elements into an ordered sequence that is to be scanned for in the subject. These pattern elements must occur in immediate succession within the subject. For example, the command

```
SCAN 1 "FIRST" " LINE"
```

would scan the line for the string "FIRST" followed immediately by the string " LINE"; in this case, the pattern match again succeeds. However, if the command

```
SCAN 1 "FIRST" "FILE"
```

were executed, the pattern match would fail since the subject does not contain the string "FIRST" followed immediately by the string "FILE".

The alternation operator is indicated by a vertical bar (|) or an exclamation point (!) between pattern elements. Alternation is used to indicate that one of several pattern elements is to be matched in the subject. For example, the command

```
SCAN 1 "FIRST" | "SECOND"
```

would scan the line for the string “FIRST” or the string “SECOND”; in this case the pattern match again would succeed. When a series of pattern elements is used as an alternative via the alternation operator, the entire set of alternatives is treated as a single pattern element; if one of the alternatives succeeds, the entire set is successful.

Concatenation and alternation may be used together to build more complex patterns. Since concatenation has higher precedence than alternation, parentheses may be used to control the structure of the pattern. For example, the command

```
SCAN 1 "FIRST" " LINE" | "SECOND" " LINE"
```

may be used to scan the subject for the string “FIRST LINE” or “SECOND LINE”. This command may also be written as

```
SCAN 1 ("FIRST" | "SECOND") " LINE"
```

In this case, the parentheses are needed to override the higher precedence of the concatenation operator.

The editor uses a pattern-matching scanner for matching the subject string against the pattern structure. The pattern structure is used like a small program or procedure that instructs the scanner how to examine the subject. At any instant during scanning, the scanner uses two pieces of information:

(1) where in the subject string it should be looking, and

(2) what element of the pattern structure it should be matching.

The scanner maintains a pointer called a cursor, which is positioned at the character in the subject that the scanner is trying to match against an element in the pattern structure.

When scanning commences, the cursor is positioned at the first character in the subject. The scanner then attempts to match the subject against the first element of the pattern.

(1) If the match is successful, the cursor is moved past the characters that matched the pattern element and is positioned at the character that must match the next element in the pattern structure, and the process is repeated. When all elements in the pattern structure have been successfully matched, the pattern match succeeds.

(2) If the match against the pattern element was not successful, the scanner backs up and attempts to match any alternatives to previous elements in the pattern structure, if they exist. If any of the alternatives provides a successful match, the cursor is moved past the matched characters and is positioned at the character that must match the next element in the pattern structure, and the pattern-match process continues as above. If none of the alternatives provides a successful match, the pattern match fails.

Each pattern element in the pattern must be matched against the subject in order for the pattern match to be successful. In addition, the matched pattern elements must occur in immediate succession within the subject.

As pattern matching proceeds, the cursor is moved from left to right along the subject string until the entire pattern match is successful (all elements of the pattern structure are matched against the subject string) or until the cursor reaches the end of the subject string without finding any possible successful matches. In the latter case, the entire pattern match fails although there may be successful matches by individual elements in the pattern.

The pattern-matching algorithm is more graphically illustrated by considering what happens when the pattern structure

```
("P" | "M") ("A" | "AT") ("CH" | "CHING")
```

is used to match the subject string “MATCHING”. Initially, the cursor is positioned at the “M” in the subject string

```
|
MATCHING
```

and a match is attempted against the first element “P” of the pattern. Since this match is unsuccessful, a match is tried against the alternative “M” to the first element, which does produce a successful match. The cursor is then positioned past the matched character “M” to the next character “A” in the subject

```
 |
MATCHING
```

and a match is attempted against the second element of the pattern structure “A”. This also produces a successful match, and the cursor is positioned to the next character “T” in the subject

```
  |
MATCHING
```

and a match is attempted against the third element in the pattern. However, since the third element “CH” or its alternative “CHING” does not produce a successful match, the scanner backs up in the pattern in order to try alternatives to previously matched pattern elements to determine if they will produce successful matches. In this case, the alternative “AT” to the second element “A” also produces a successful match and the cursor is positioned past the “AT” in the subject to “C”,

```
   |
MATCHING
```

which does produce a successful match in the third pattern element. At this point, the entire pattern has successfully matched the string “MATCH” and the pattern match succeeds. The elements of the pattern that produced the match were “M”, “AT”, and “CH”. Notice that the alternative to the third element “CHING” was not used during the match since the entire pattern match succeeded before that alternative was attempted.

The above example also illustrates an important control that the user has over the scanner. In a pattern structure, alternatives are matched by the scanner in left-to-right order. Therefore, by positioning alternatives correctly, the user can control the order in which the scanner matches them.

Anchoring is another concept that is important in pattern matching. Both the SCAN command and the MATCH command may be used for scanning the subject string for a pattern. The only difference between the SCAN command and the MATCH command is that the SCAN command allows the pattern to be matched anywhere in the subject while the MATCH command requires that the

pattern match the subject string beginning at the first character position of the current column range. For example,

```
SCAN 1 "FIRST"
```

successfully matches the string “FIRST” in the above subject string, while the command

```
MATCH 1 "FIRST"
```

fails to match the subject since the string “FIRST” is not at the beginning of the subject.

Predefined Pattern Elements

The editor pattern-matching facility has several special predefined pattern elements which may be used in pattern structures. Most of these special elements are designed to match a certain group of character strings instead of a specified character string. Some of the most commonly used pattern elements are described below; the complete descriptions of all the pattern elements are given at the end of this volume. The previous line containing the string

```
THIS IS THE FIRST LINE OF MY FILE.
```

should be used as the subject in the following examples.

(1) LEN

LEN is a pattern element that may be used to match a character string of a specified length within the subject. The command

```
SCAN 1 LEN(4)
```

will match the first four characters of the first line which is the string “THIS”. The last word in the line could be matched by the command

```
SCAN 1 LEN(4) "."
```

In this case, the pattern instructs the scanner to match a string of four characters which is immediately followed by the character “.”.

(2) SPAN, BREAK, SKIP

SPAN and BREAK are pattern elements that are used to match a run of characters. Patterns used to match

(a) a run of blanks,
(b) a run of digits, or
(c) a run of letters (a word)

may be formed by using

(a) SPAN(" ")
(b) SPAN("0123456789")
(c) SPAN("ABCDEFGHIJKLMNOPQRSTUVWXYZ")

The pattern element SPAN matches the longest string in the subject starting at the current cursor position that consists solely of the characters appearing in the argument for SPAN. For example, the command

```
SCAN 1 SPAN("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

will match the first word "THIS" of the subject. The SPAN pattern element must match at least one character in the subject, or else it fails.

Patterns used to match

(a) everything up to the next blank,
(b) everything up to the next punctuation mark, or
(c) everything up to the next number

may be formed by using

(a) BREAK(" ")
(b) BREAK(",.;:!?")
(c) BREAK("+−0123456789")

The pattern element BREAK matches the longest string in the subject starting at the current cursor position that does not contain a character from the argument. In other words, BREAK matches everything up to but not including the first "break" character. For example, the command

```
SCAN 1 BREAK(" ")
```

also will match the first word "THIS" of the subject. The BREAK pattern element must find at least one break character in the subject, or else it fails.

The SKIP pattern element is similar to SPAN except that it does not require that any characters be matched from the subject.

The following example illustrates how the SPAN and BREAK pattern elements may be used to decompose the beginning of a sentence into its individual words.

```
SCAN 1 BREAK(" ") SPAN(" ") BREAK(" ")
```

In this case, the first occurrence of BREAK in the pattern matches the word "THIS", and the second BREAK matches the word "IS"; the SPAN pattern element matches all of the blanks that occur between the first two words.

(3) ANY, NOTANY

ANY and NOTANY are pattern elements that are used to match single characters. ANY matches any character appearing in its argument. For example, the command

```
SCAN 1 ANY("AEIOU")
```

will match the first vowel that appears in the subject, in this case the letter I in the first word "THIS". NOTANY matches any character that does not appear in its argument; for example, the command

```
SCAN 1 NOTANY("AEIOU")
```

will match the first character that is not a vowel, in this case the letter "T".

(4) TAB, RTAB, REM

TAB and RTAB are pattern elements that are used to match all characters from the current cursor position up to a specified column in the subject. TAB(n) matches up through the *n*th character in the subject; RTAB(n) matches up to but not including the *n*th character from the right end of the subject. For example, the TAB pattern element in the command

```
SCAN 1 BREAK(" ") TAB(11)
```

will match the string " IS THE", while the RTAB pattern element in the command

```
SCAN 1 "LINE" RTAB(1)
```

will match the string " OF MY FILE". The pattern element RTAB(0) is useful for matching the remainder of the subject; this pattern element also may be written as REM. For example, REM in the command

```
SCAN 1 "LINE" REM
```

will match the string " OF MY FILE.".

(5) POS, RPOS

POS and RPOS are pattern elements that are used to check whether the cursor is at a certain character position in the subject. POS(n) succeeds only if the cursor is positioned immediately past the *n*th character; RPOS(n) succeeds if the cursor is positioned immediately before the *n*th character from the right end of the subject. Both POS and RPOS match the null string, i.e., they do not match any of the actual characters in the subject. For example, the POS pattern element in the command

```
SCAN 1 BREAK(" ") POS(4)
```

will check to ensure that the cursor is just past the fourth character in the subject, i.e., that the first word of the sentence is a four-character word. The pattern elements POS(0) and RPOS(0) are particularly useful for checking if the cursor is at the beginning or end of the subject. For example, the command

```
SCAN 1 POS(0) "THIS"
```

succeeds only because the word "THIS" is the first word in the subject, while the command

```
SCAN 1 "LINE" RPOS(0)
```

fails because the word "LINE" is not the last word of the subject. The use of POS(0) or RPOS(0) in effect anchors the scanning to the left or right end of the subject. The MATCH command also may be used to anchor the pattern match to the left end of the subject, e.g.,

```
MATCH 1 "THIS"
```

(6) ARB

ARB is a pattern element that matches zero or more arbitrary characters. When first encountered by the scanner moving from left to right, ARB matches the null string. When backed into on subsequent scans, ARB increases the number of characters matched by one. ARB fails only when it no longer can increase the number of characters matched. For example, the ARB pattern element in the command

```
SCAN 1 BREAK(" ") ARB SPAN("ABCDEFGHIJKLMNOPQRSTUVWXYZ") "."
```

will match everything between the first and last words of the sentence in the subject. In this case, the match of ARB is increased one character at a time starting with the first character after the word “THIS” until the pattern element SPAN matches a sequence of letters followed by a period at the end of the sentence. In effect, the ARB pattern element moves the cursor across the remainder of the subject from the current cursor position until the rest of the pattern succeeds (or until the end of the subject is encountered).

Variables

The editor allows the user to define variables for use in the construction of pattern structures. These variables may be used by the pattern structure to assign matched substrings of the subject, or may be used to save complete pattern structures.

A variable is initially created by the LET command, which has the syntax

LET variable=value

The variable name consists of 1 to 255 characters, the first of which must be alphabetic (A-Z) with the subsequent ones being alphanumeric (A-Z, 0-9, and _). The value may be any one of the following types:

(1) an integer, e.g., 5, –4.

(2) a line number, e.g., *F, 1.5

(3) a string or string expression (see below)

(4) a pattern (see below).

A variable is not declared to be of a single data type; the context in which the variable is used determines its data type.

If the value of the variable is a string, the string must be enclosed by editor string delimiters, e.g., “STRING1”, 'string2'. The null string, which is a string of zero characters, also may be assigned to a variable by giving adjacent string delimiters, e.g., “” or ''. A *string expression*, which is the combination of one or more strings and variables whose values are strings, may be the value of a variable. Each string and/or variable in the expression must be separated by one or more blanks. The value is the concatenation of the values of the individual components taken from left to right. For example, if X=“ABC” and Y=“123”, then

```
LET W = X 'DEF' Y ' 45' "6"
```

would result in W="ABCDEF123 456".

The value of a variable may be a pattern structure. For example, the command

```
LET PAT1 = BREAK(" ") SPAN(" ") BREAK(" ")
```

may be used to assign the above pattern to the variable PAT1 which then may be used subsequently by another editor command, e.g.,

```
SCAN 1 PAT1
```

This SCAN command is equivalent to the SCAN command given above for decomposing a sentence into its individual words.

Assigning patterns to variables is very useful for preserving commonly used pattern structures. Some common examples are:

```
LET ALPHA = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
LET LCALPHA = "abcdefghijklmnopqrstuvwxyz"
LET NUM = "0123456789"
LET ALPHANUM = ALPHA LCALPHA NUM
LET PUNC = ".,;:!?"
LET ANYALPHA = ANY(ALPHA)
LET ANYNUM = ANY(NUM)
LET SPNALPHANUM = SPAN(ALPHANUM)
LET BKPUNC = BREAK(PUNC)
```

The value, type, and length of a variable may be displayed by issuing the command

```
PRINT VAR=variable
```

The names of all defined editor variables may be displayed by issuing the command

```
PRINT VARS
```

Predefined Variables

The editor has several predefined variables. These are listed below. The variables marked as read-only (RO) cannot be directly modified by the user.

ED_AZ	Upper and lowercase letters (RO)
ED_COUNT	Count produced by COUNT command (RO)
ED_CURSOR_COL	Column of cursor in visual mode (RO)
ED_CURSOR_FIELD	Type of line pointed to by cursor (LINE, WORK, or RULER) (R0)
ED_CURSOR_LINE	Line number of cursor in visual mode (RO)
ED_CURSOR_POS	POS value (column – 1) of cursor in visual mode (RO)
ED_DATA_COLS	Number of data column positions (RO)
ED_FILENAME	Edit file name (without trailing blanks) (RO)
ED_LCAZ	Lowercase letters (RO)
ED_LEFT_COLUMN	Left value of COLUMN command (RO)
ED_LEFT_WINDOW	Left value of WINDOW command (RO)
ED_MSINK	Writes a string to MSINK
ED_MSOURCE	Reads a line from MSOURCE and returns result as a string

	(RO)
ED_PF_NAME	PF-key cluster name (RO)
ED_RIGHT_COLUMN	Right value of COLUMN command (RO)
ED_RIGHT_WINDOW	Right value of WINDOW command (RO)
ED_SAVENAME	Name associated with the current SAVE/REMEMBER commands (RO)
ED_SCAN_COL	Column number of last scan match (RO)
ED_SCREEN_LINES	Number of lines on screen (RO)
ED_SINK	Writes a string to SINK
ED_SOURCE	Reads a line from SOURCE and returns result as a string (RO)
ED_STATUS	Status expression for visual mode
ED_STATUS_DEF	Default status expression for visual mode (file name, column values, window values, PF-key cluster name) (RO)
ED_TIME	Current time of day (RO)
ED_UCAZ	Uppercase letters (RO)
ED_VMARK_FIRST_COL	Column number associated with VMARK first line (RO)
ED_VMARK_FIRST_LINE	First line of VMARK region (RO)
ED_VMARK_LAST_COL	Column number associated with VMARK last line (RO)
ED_VMARK_LAST_LINE	Last line of VMARK region (RO)
ED_VRULER	Current visual-mode ruler
ED_VRULER_DEF	Default visual-mode ruler (RO)
ED_WORK	Contents of visual-mode work area
ED_09	Numeric characters (RO)

As an example, the following SCAN command will place the first complete “word” (string of nonblank characters) of the current line into the visual editor work area.

```
SCAN * SKIP(" ") (SPAN(ED_AZ ED_09) $ ED_WORK)
```

Pattern-Match Value Assignment

Variables may be assigned values during the pattern-matching process via the $ assignment operator. The form of the assignment is

pattern-element $ variable

If “pattern-element” is matched during the pattern-matching process, the matched substring will be assigned to “variable”. This assignment is done immediately. Thus, regardless if the entire pattern match subsequently fails, “variable” will be assigned a value if “pattern-element” matches a substring of the subject. For example, using the previous line containing the string

```
THIS IS THE FIRST LINE OF MY FILE.
```

the command

```
MATCH 1 BREAK(" ") $ WORD SPAN(" ") "IS"
```

assigns the first word “THIS” to the variable WORD. If the pattern were changed to

```
MATCH 1 BREAK(" ") $ WORD SPAN(" ") "WAS"
```

the variable WORD would still be assigned the string “THIS” even though the entire pattern match fails.

The precedence of the $ operator is higher than that of either the concatenation or alternation operators. Parentheses may be used in the pattern to override the precedence. For example, the command

```
MATCH 1 BREAK(" ") SPAN(" ") BREAK(" ") $ WORD
```

will assign the string "IS" to the variable WORD, while the command

```
MATCH 1 (BREAK(" ") SPAN(" ") BREAK(" ")) $ WORD
```

will assign the string "THIS IS" to the variable WORD. The above examples use the MATCH command to anchor the pattern match to the beginning of the string. Note that the value of the variable cannot be used as a pattern element, e.g.,

```
MATCH 1 BREAK(" ") $WORD SPAN(" ") WORD
```

is invalid.

TAB and REM may be used in conjunction with value assignment when scanning files containing structured data such as data that is ordered by column number. For example, if line 10 of a file contains

```
JON Q. DOUGH 123 DOLLAR ST. BUCKMUCH, TENN.
```

the command

```
SCAN 10 TAB(20) $ NAME TAB(40) $ ADDR REM $ TOWN
```

can be used to break up the subject line into its constituent components, e.g., name, street address, and town. In this case, the value assignments would be

```
NAME = "JON Q. DOUGH         "
ADDR = "123 DOLLAR ST.       "
TOWN = "BUCKMUCH, TENN.      "
```

Pattern-Match Replacement

A matched substring in the subject may be replaced by another string or a string expression via the = replacement operator. The form of the replacement is

(pattern-element = expression)

If "pattern-element" is successfully matched during the pattern-matching process, the matched string is replaced in the subject by "expression". The replacement is done only if the entire pattern match succeeds. "expression" may be either a single string or a string expression as defined above under "Variables". The string expression may not contain parentheses. "pattern-element" consists of all of the pattern elements to the left of the = operator until the beginning of the pattern or an unbalanced left parenthesis "(" is encountered. The replacement operator may only be used in patterns with the ALTER and CHANGE commands. For example, the command

```
ALTER 1 ("FIRST" = "NEXT")
```

may be used to alter the first line of the file to

```
THIS IS THE NEXT LINE OF MY FILE.
```

"expression" consists of all strings and variables after the = operator until an unbalanced right parenthesis ")" is encountered. More than one replacement is allowed in a pattern if the replacements do not overlap. For example, the command

```
ALTER 1 ("FIRST" = "NEXT") ARB ("FILE" = "DATA")
```

may be used to alter the first line to

```
THIS IS THE NEXT LINE OF MY DATA.
```

In this case, parentheses are needed around both replacement operations in order to exclude the substring matched by ARB from the replacements.

"expression" is evaluated after the pattern match occurs and before the replacement operation is done. Therefore, any variables whose values are changed by the $ assignment operator will have their new values used in the replacement. For example, the command

```
ALTER 1 ("FIRST" = "NEXT") ARB ("FILE" $ WORD = "DATA " WORD)
```

may be used to alter the first line to

```
THIS IS THE NEXT LINE OF MY DATA FILE.
```

In this case, the variable WORD is assigned the value of the matched string "FILE" and then used as part of the replacement operation.

The ALTER and CHANGE commands normally function in unanchored mode, that is, they allow the pattern match to occur anywhere in the subject. Therefore, POS(0) or RPOS(0) must be used in the pattern structure to force anchoring. For example, the command

```
ALTER 1 POS(0) ("THIS" = "THAT")
```

will change the word "THIS" to "THAT" in the subject only if it is at the beginning of the subject line. On the other hand, the command

```
ALTER 1 ("THIS" = "THAT")
```

will change the first occurrence of the word "THIS" to "THAT" regardless of where it appears in the line, and the command

```
ALTER@ALL 1 ("THIS" = "THAT")
```

will change all occurrences of "THIS" to "THAT" in line 1.

Other Commands and String Expressions

When the pattern matching facility is enabled, string expressions (as defined above in "Variables") are allowed in an edit command wherever a string is allowed, except for the ALTER, CHANGE, MATCH, SCAN, and TRANSLATE commands.

The ALTER, CHANGE, MATCH, and SCAN commands require patterns (which may contain string expressions). The TRANSLATE command syntax becomes

TRANSLATE lpar string-exp1=string-exp2

An example of the use of string expressions in place of strings is

```
INSERT 10 'THE NUMBERS ' ED_09 '.'
```

which inserts the string “THE NUMBERS 0123456789.” at line 10.

EDITOR PATTERN-ELEMENT DEFINITIONS

The same notation is used for the edit pattern-element definitions as is used for the edit command descriptions.

Summary of Pattern Elements

* string
ABORT
ANY(string)
ANYCASE(string)
ARB
* ASCII(string)
BREAK(string)
BREAKX(string)
* DUPL(string,count)
* EBCDIC(string)
FAIL
FENCE
* HEX(string)
* LC(string)
LEN(length)
* LPAD(string,len[,padch])
NOTANY(string)
POS(column)
REM
* RPAD(string,len[,padch])
RPOS(column)
RTAB(column)
SIZE(string)
SKIP(string)
SKIPNOT(string)
SPAN(string)
* SUBSTR(string,beg[,end])
* SUBSTR(string,beg | len)
TAB(column)
* UC(string)

The pattern elements marked with an asterisk are functions which modify a string and return a result. Therefore, they may be used in a string expression (strexp) in any edit command that accepts string expressions.

string

Edit Pattern Description

Purpose: To match a specified character string.

Prototype: string

Action: The subject is compared with *string* starting at the current cursor position. If the subject contains *string*, the pattern match is successful; otherwise, the pattern match fails.

Examples: The resulting pattern match in the subject line after matching the pattern:

```
"versus"

Comments on the case,  Regina versus Christensen,
                                    └─────┘
                                      └──versus
```

The following ALTER command may be used with the above subject line to change "versus" to "vs.":

```
ALTER * 'versus' = 'vs.'
```

The result is

```
Comments on the case,  Regina vs. Christensen,
```

ABORT

Edit Pattern Description

Purpose: To cause a permanent failure of the entire pattern-matching sequence.

Prototype: ABORT

Action: ABORT is a pattern element that will always cause the entire pattern-matching sequence to fail. ABORT will prevent the scanner from seeking alternative matches.

Examples: The resulting pattern match in the subject line after matching the pattern:

```
(POS(0) "*" ABORT) | "BALR"

STMT1  BALR  14,15
       └──┘
        └──BALR
```

The above example tests the subject line for the string “BALR” if and only if the first character of the line is not “*”.

The following SCAN command scans the entire file for all BALR assembler instructions (while ignoring 370 Assembler “*” comment lines):

```
SCAN@ALL /FILE (POS(0) "*" ABORT) | "BALR"
```

ANY

Edit Pattern Description

Purpose: To match a single character which is selected from a given set of characters.

Prototype: ANY(string)

Parameter:

string is a string giving the set of characters from which a character is to be matched.

Action: Starting at the current cursor position, ANY will match any single character in the subject which is one of those given in *string*. If the next character in the subject does not contain a character given in *string*, ANY will fail.

Examples: The resulting pattern match in the subject line after matching the pattern:

```
ANY("aeiou")

Comments on the case,  Regina versus Christensen,
 │
 └──ANY("aeiou")
```

The resulting pattern match in the subject line after matching the pattern:

```
BREAK(",") ARB ANY("aeiou")

Comments on the case,  Regina versus Christensen,
└──────────────────┘└─┘│
   └──BREAK(",")     │  └──ANY("aeiou")
                     └──ARB
```

The following SCAN commands will assign the first matched vowel in the above subject line to the variable VOWEL:

```
SCAN * ANY("aeiou") $ VOWEL


SCAN * BREAK(",") ARB ANY("aeiou") $ VOWEL
```

With the first SCAN command, the value of VOWEL is set to “o”; with the second SCAN command, the value of VOWEL is set to “e”.

ANYCASE

Edit Pattern Description

Purpose: To match a specified character string.

Prototype: ANYCASE(string)

Parameter:

string is the character string to be matched in the subject.

Action: The subject is compared with *string* starting at the current cursor position. If the subject contains *string*, the pattern match is successful; otherwise, the pattern match fails. The match is done regardless of the case of the characters in the subject and *string*. For example, ANYCASE("BLERF") matches “bLeRf”, “BLErf”, and “blerf”.

For very simple cases, alternation patterns are more efficient than ANYCASE patterns. For example,

```
SCAN /FILE ("EDITOR" | "Editor" | "editor")
```

is more efficient than

```
SCAN /FILE ANYCASE("editor")
```

This is because the processing necessary for using ANYCASE involves converting each string with the proper first letter (“E” or “e” in the example above) to uppercase and then comparing it against the uppercase representation stored for ANYCASE. Note, however, for cases involving more than three or four alternatives, ANYCASE is more efficient. The alternation given above only matches three possibilities out of the sixty-four permutations of the upper and lowercase characters in “editor”.

Examples: The resulting pattern match in the subject line after matching the pattern:

```
ANYCASE("regina")

Comments on the case,  regina versus Christensen,
                       └────┘
                         └──Regina
```

The following ALTER command alters all case forms of the word “regina” to “Regina” throughout the entire file:

```
ALTER@ALL /FILE ANYCASE("regina") = "Regina"
```

ARB

Edit Pattern Description

Purpose: To match an arbitrary number of characters.

Prototype: ARB

Action: ARB matches an arbitrary number of characters in the subject starting at the current cursor position. This may be zero or more characters and may include the remainder of the subject.

ARB is normally used in conjunction with other pattern elements. ARB increases the number of characters matched by moving the cursor one character position at a time until the entire pattern succeeds. If the end of the subject is reached before the entire pattern succeeds, then ARB will fail.

In effect, the ARB pattern element moves the cursor from the current cursor position across the remainder of the subject until the rest of the pattern succeeds (or until the end of the subject is encountered).

An alternate name for ARB is three dots (**...**).

Examples: The resulting pattern match in the subject line after matching the pattern:

```
"Comments" ARB ","

Comments on the case,  Regina versus Christensen,
└───────┘└──────────┘│
  └─Comments └─ARB  └─,
```

The following ALTER command alters the phrase “on the case” to “about the case” in the subject line:

```
ALTER * "Comments" (ARB = " about the case") ","
```

The result is

```
Comments about the case,  Regina versus Christensen,
```

ASCII, EBCDIC

Edit Pattern Description

Purpose: To convert an EBCDIC character string into its ASCII equivalent value, or to convert an ASCII character string into its EBCDIC equivalent value.

Prototype: ASCII(string)

EBCDIC(string)

Parameter:

string is the string to be converted.

Action: The pattern element ASCII converts the EBCDIC character string in *string* into its ASCII equivalent value.

The pattern element EBCDIC converts the ASCII character string in *string* into its EBCDIC equivalent value.

The null string is not allowed as an argument to ASCII or EBCDIC.

BREAK

Edit Pattern Description

Purpose: To match zero or more contiguous characters which are not part of a given set of characters.

Prototype: BREAK(string)

Parameter:

string is a string giving the set of characters to stop matching on.

Action: Starting at the current cursor position, BREAK will match all the characters in the subject up to but not including a character in the subject which is in *string*.

BREAK does not match the character it stops on. If the character at the current cursor position is a member of *string*, BREAK will match the null string. BREAK will fail if it does not find a character in the subject which is also in *string*.

Examples: The resulting pattern match in the subject line after matching the pattern:

```
BREAK(" ")

Comments on the case,  Regina versus Christensen,
└──────┘
  └──BREAK(" ")
```

The following SCAN command assigns the portion of the subject line matched by BREAK to the variable FIRST (in this case the string “Comments”):

```
SCAN * BREAK(" ") $ FIRST
```

The resulting pattern match in the subject line after matching the pattern:

```
BREAK(",")

Comments on the case,  Regina versus Christensen,
└────────────────────┘
    └──BREAK(",")
```

The following SCAN command assigns the portion of the subject line matched by BREAK to the variable FIRST and the remainder of the string after the break character to the variable LAST:

```
SCAN * (BREAK(" ") $ FIRST) "," (REM $ LAST)
```

In this case, FIRST is set to “Comments on the case” and LAST is set to “Regina versus Christensen,”.

The resulting pattern match in the subject line after matching the pattern:

```
"case" BREAK(",")
```

```
Comments on the case,  Regina versus Christensen,
                  └──┘
                   └──case
```

In the above example, BREAK matches the null string since the cursor is positioned at the break character "," after matching "case" assuming the pattern match is performed in unanchored mode.

BREAKX

Edit Pattern Description

Purpose: To match zero or more contiguous characters which are not part of a given set of characters.

Prototype: BREAKX(string)

Parameter:

string is a string giving the set of characters to stop matching on.

Action: Starting at the current cursor position, BREAKX will match all the characters in the subject up to but not including a character in the subject which is in *string*.

BREAKX does not match the character it stops on. BREAKX will fail if it does not find a character in the subject which is also in *string*.

BREAKX is the same as BREAK except that BREAKX has implicit alternatives which are obtained by scanning past the first break character found and scanning to the next break character. In other words, should the pattern fail, BREAKX will force scanning past the current break character and match (like BREAK) at the next break character.

Note that BREAKX may be used to replace ARB in many situations where BREAK cannot be easily used. For example, the pattern

```
"PETS" BREAKX("CD") ("CATS"|"DOGS")
```

may be used to replace

```
"PETS" ARB ("CATS"|"DOGS")
```

which produces in some circumstances a more efficient pattern when scanning a subject line such as

```
MY FAVORITE PETS ARE CANARIES, TURTLES, AND DOGS.
```

In this case, BREAKX only has to try two alternatives while ARB must try several alternatives before “DOGS” is matched.

Examples: Resulting pattern match in the subject line after matching the pattern:

```
POS(0) BREAKX(" ") " versus"

Comments on the case,  Regina versus Christensen,
└──────────────────────────────┘└──────┘
   └──BREAKX(" ")                └──" versus"
```

If BREAK were used in the above pattern instead of BREAKX, the pattern match would fail.

DUPL

Edit Pattern Description

Purpose: To match (or duplicate) a string by a specified number of times.

Prototype: DUPL(string,count)

Parameters:

string is the string to be matched or duplicated.
count is the number of times that *string* is to be duplicated. *count* must be greater than or equal to zero.

Action: Starting at the current cursor position, DUPL will match the subject string by the number of times specified by the count.

The DUPL pattern may be used in conjunction with the SIZE pattern to replace a string with an equivalent number of blanks, e.g.,

```
ALTER * 'string' $ A = DUPL(' ',SIZE(A))
```

FAIL

Edit Pattern Description

Purpose: To cause a local failure in the pattern-matching sequence.

Prototype: FAIL

Action: FAIL is a pattern element that will always cause the pattern-matching sequence to fail. FAIL is useful for forcing the scanner to seek alternative matches.

FENCE

Edit Pattern Description

Purpose: To prevent an alternative pattern match from backing up past a certain position in the subject.

Prototype: FENCE

Action: FENCE matches the null string when encountered by the pattern-match scanner when moving left to right through a pattern. If a subsequent failure causes the scanner to back up to try an alternative pattern, FENCE prevents the scanner from trying alternatives prior to its own occurrence in the pattern.

HEX

Edit Pattern Description

Purpose: To convert a character representation of a hexadecimal string to its hexadecimal value.

Prototype: HEX(string)

Parameter:

string is the character representation of the hexadecimal string to be converted.

Action: The character representation of the hexadecimal string is converted to its hexadecimal value.

The null string is not allowed as an argument to HEX.

Example: The following command will scan the file for the character string "ABC" followed by the hexadecimal constant "01".

```
SCAN /FILE "ABC" HEX("01")
```

LC, UC

Edit Pattern Description

Purpose: To convert a character string into lowercase, or to convert a character string into uppercase.

Prototype: LC(string)

UC(string)

Parameter:

string is the character string to be converted into lower- or uppercase.

Action: The pattern element LC converts the character string in *string* into lowercase characters.

The pattern element UC converts the character string in *string* into uppercase characters.

The null string is not allowed as an argument to LC or UC.

Example: The following command will alter all occurrences of the string “fdub”, “Fdub”, “FDub”, etc. to the string “FDUB”.

```
ALTER@ALL /FILE ANYCASE("fdub") = UC("fdub")
```

LEN

Edit Pattern Description

Purpose: To match an arbitrary string of characters of a certain length.

Prototype: LEN(length)

Parameter:

length is a nonnegative integer value specifying the number of characters to match.

Action: Starting at the current cursor position, LEN will match the next *length* characters in the subject. If there are not *length* characters remaining in the subject, LEN will fail.

Examples: The resulting pattern match in the subject line after matching the pattern:

```
LEN(5)

Comments on the case,  Regina versus Christensen,
└────┘
  └──LEN(5)
```

The following SCAN command assigns the first five characters of the subject line to the variable FIRST:

```
SCAN * LEN(5) $ FIRST
```

The resulting pattern match in the subject line after matching the pattern:

```
LEN(5) RPOS(0)

Comments on the case,  Regina versus Christensen,
                                                └────┘
                                          LEN(5)──┘
```

The following SCAN command assigns the last five characters of the subject line to the variable LAST:

```
SCAN * LEN(5) $ LAST RPOS(0)
```

LPAD

Edit Pattern Description

Purpose: To pad a string to a given length. The padding is done on the left end of the string.

Prototype: LPAD(string,len[,padch])

Parameters:

string is the string to be padded.
len is the string length desired.
padch is the string to be used as the pad character. Only the first character will be used. If this parameter is omitted, the current editor pad character will be used (defaults to blank).

Action: The LPAD pattern element pads *string* on the left with the pad character to make it *len* characters long. If *string* is already more than *len* characters in length, it is left unchanged; in this case, LPAD still succeeds.

Examples: The LPAD pattern gives the following results:

```
LPAD("abc",5)   yields   "  abc"
LPAD("abc",2)   yields   "abc"
```

NOTANY

Edit Pattern Description

Purpose: To match a single character which is not one of a given set of characters.

Prototype: NOTANY(string)

Parameter:

string is a string giving the set of characters from which a character is not to be matched.

Action: Starting at the current cursor position, NOTANY will match the character in that cursor position in the subject which is not one of those given in *string*.

Examples: The resulting pattern match in the subject line after matching the pattern:

```
NOTANY("aeiou")

Comments on the case,  Regina versus Christensen,
│
└──NOTANY("aeiou")
```

The resulting pattern match in subject line after matching the pattern:

```
BREAK(",") SPAN(" ,") NOTANY("aeiou")

Comments on the case,  Regina versus Christensen,
└────────────────────┘└─┘│
      └──BREAK(",")   │ └──NOTANY("aeiou")
                      └──SPAN(" ,")
```

The following SCAN commands will assign the first matched consonant in the above subject line to the variable CONS:

```
SCAN * NOTANY("aeiou") $ CONS


SCAN * BREAK(",") SPAN(" ,") NOTANY("aeiou") $ CONS
```

With the first SCAN command, the value of CONS is set to “C”; with the second SCAN command, the value of CONS is set to “R”.

POS

Edit Pattern Description

Purpose: To test if the current cursor position in the subject is at a certain column with respect to the beginning (left-hand edge) of the subject.

Prototype: POS(column)

Parameter:

column is a nonnegative integer value specifying the column position to be tested.

Action: The current cursor position relative to the beginning of the subject is compared with *column*. If they are equal, then POS succeeds. If they are not equal, then POS fails. POS never moves the current cursor position; when it succeeds it matches the null string.

POS(0) tests for the current cursor position immediately before the first character in the string. POS(n) tests for the cursor position immediately after the last character in the subject, if the subject is "n" characters long.

POS is normally used to anchor the pattern match to a certain column position. This is useful when the pattern is being used to test for the occurrence of a character string starting at a certain column position.

Examples: The resulting pattern match in the subject line after matching the pattern:

```
POS(14) "matter"

The case is a matter of public record; we do not
              └────┘
              │ └──matter
              └──POS(14)
```

The resulting pattern match in the subject line after matching the pattern:

```
SKIPNOT(";") POS(37)

The case is a matter of public record; we do not
└──────────────────────────────────────┘ │
             └──SKIPNOT(";")           └──POS(37)
```

REM

Edit Pattern Description

Purpose: To match the remainder of the subject.

Prototype: REM

Action: Starting at the current cursor position, REM will match all the characters remaining in the subject. REM will never fail to match; if the cursor is already at the end of the subject, REM will match the null string.

Note that REM is the equivalent to the pattern element RTAB(0).

Examples: The resulting pattern match in the subject line after matching the pattern:

```
"Comments" REM

Comments on the case,  Regina versus Christensen,
└──────┘└──────────────────────────────────────────┘
   └──Comments          └──REM
```

The following SCAN command will assign the remainder of the subject line to the variable LAST:

```
SCAN * "Comments" REM $ LAST
```

RPAD

Edit Pattern Description

Purpose: To pad a string to a given length. The padding is done on the right end of the string.

Prototype: RPAD(string,len[,padch])

Parameters:

string	is the string to be padded.
len	is the string length desired.
padch	is the string to be used as the pad character. Only the first character will be used. If this parameter is omitted, the current editor pad character will be used (defaults to blank).

Action: The RPAD pattern element pads *string* on the right with the pad character to make it *len* characters long. If *string* is already more than *len* characters in length, it is left unchanged; in this case, RPAD still succeeds.

Examples: The RPAD pattern gives the following results:

```
RPAD("abc",5)   yields   "abc  "
RPAD("abc",2)   yields   "abc"
```

RPOS

Edit Pattern Description

Purpose: To test if the current cursor position in the subject is at a certain column with respect to the end (right-hand edge) of subject.

Prototype: RPOS(column)

Parameter:

column is a nonnegative integer value specifying the column position to be tested.

Action: The current cursor position relative to the end of the subject is compared with *column*. If they are equal, then RPOS succeeds. If they are not equal, then RPOS fails. RPOS never moves the current cursor position; when it succeeds it matches the null string.

RPOS(0) tests for the current cursor position immediately after the last character in the string. RPOS(n) tests for the cursor position immediately before the first character position in the subject, if the subject is "n" characters long.

RPOS is normally used to anchor the pattern match to a certain column position. This is useful when the pattern is being used to test for the occurrence of a character string starting at a certain column position.

Examples: The resulting pattern match in the subject line after matching the pattern:

```
RPOS(34) "matter"

The case is a matter of public record; we do not
              └─────┘
              │ └──matter
              └──RPOS(34)
```

The resulting pattern match in the subject line after matching the pattern:

```
SKIPNOT(";") RPOS(11)

The case is a matter of public record; we do not
└──────────────────────────────────────┘│
            └──SKIPNOT(";")           └──RPOS(11)
```

RTAB

Edit Pattern Description

Purpose: To match all characters in the subject string until the current cursor position is at a specific column with respect to the end of the string.

Prototype: RTAB(column)

Parameter:

column is a nonnegative integer value specifying the column position to be matched up to.

Action: The current cursor position with respect to the right-hand end of the subject is compared with *column*. If the current cursor position is less, RTAB fails; otherwise, RTAB matches the zero or more characters necessary to move the current cursor position to *column*.

RTAB(0) moves the current cursor position to immediately after the last character in the string. RTAB(n) moves the cursor to the first character position in the subject if the subject is "n" characters long.

Note that RTAB(0) is equivalent to the pattern element REM.

Examples: The resulting pattern match in the subject line after matching the pattern:

```
RTAB(2)

The case is a matter of public record; we do not
└──────────────────────────────────────────────┘
                └──RTAB(2)
```

The resulting pattern match in the subject line after matching the pattern:

```
SKIPNOT(";") RTAB(2)

The case is a matter of public record; we do not
└──────────────────────────────────┘└──────┘
             └──SKIPNOT(";")          └──RTAB(2)
```

SIZE

Edit Pattern Description

Purpose: To return the size (the number of characters) of a string.

Prototype: SIZE(string)

Parameter:

string is the string whose size is to be returned.

Action: The SIZE pattern returns the number of characters contained in the subject string.

The SIZE pattern may be used in conjunction with the DUPL pattern to replace a string with an equivalent number of blanks, e.g.,

```
ALTER * 'string' $ A = DUPL(' ',SIZE(A))
```

SKIP

Edit Pattern Description

Purpose: To match zero or more contiguous characters which are selected from a given set of characters.

Prototype: SKIP(string)

Parameter:

string is a string giving the set of characters to match (skip over).

Action: Starting at the current cursor position, SKIP will match all the characters in the subject up to a character which is not one of those given in *string*.

Note that SKIP will succeed if the character at the current cursor position is not one of those of *string* when it starts matching (i.e., it does not match any characters). Thus, SKIP is the same as SPAN except that SPAN must skip at least one character and SKIP does not have to. SKIP does not match the character it stops on.

Example: The resulting pattern match in the subject line after matching the pattern:

```
BREAK(" ") SKIP(" ") BREAK(" ")

Comments    the case,  Regina versus Christensen,
└───────┘└──┘└─┘
|         |   └──BREAK(" ")
|         └──SKIP(" ")
└──BREAK(" ")
```

SKIPNOT

Edit Pattern Description

Purpose: To match zero or more contiguous characters which are not part of a given set of characters.

Prototype: SKIPNOT(string)

Parameter:

string is a string giving the set of characters to stop matching on.

Action: Starting at the current cursor position, SKIPNOT will match all the characters in the subject up to a character which is one of those given in *string*.

Note that SKIPNOT, unlike BREAK, will succeed if it does not find a character in the subject which is also in *string* (i.e., SKIPNOT will match the entire line). SKIPNOT does not match the character it stops on.

Examples: The resulting pattern match in the subject line after matching the pattern:

```
SKIPNOT("/")

the work was performed under the ID's belonging,
└──────────────────────────────┘
                └──SKIPNOT("/")
```

The resulting pattern match in the subject line after matching the pattern:

```
SKIPNOT(" ,")
the work was performed under the ID's belonging,
└─┘
 └──SKIPNOT(" ,")
```

The resulting pattern match in the subject line after matching the pattern:

```
"was p" SKIPNOT(" ,")

the work was performed under the ID's belonging,
        └──┘ └──────┘
          └─was p └─SKIPNOT(" ,")
```

SPAN

Edit Pattern Description

Purpose: To match one or more contiguous characters in the subject which are selected from a given set of characters.

Prototype: SPAN(string)

Parameter:

string is a string giving the set of characters to match (skip over).

Action: Starting at the current cursor position, SPAN will match all the characters in the subject up to a character which is not one of those given in *string*.

Note that SPAN will fail if the character at the current cursor position is not one of those in *string* when it starts matching (i.e., it does not match any characters).

Examples: The resulting pattern match in the subject line after matching the pattern:

```
SPAN("Com")

Comments on the case,  Regina versus Christensen,
└──┘
 └──SPAN("Com")
```

The resulting pattern match in the subject line after matching the pattern:

```
BREAK(" ") SPAN(" ") BREAK(" ")

Comments    the case,  Regina versus Christensen,
└───────┘└──┘└─┘
  │       │   └──BREAK(" ")
  │       └──SPAN(" ")
  └──BREAK(" ")
```

SUBSTR

Edit Pattern Description

Purpose: To generate a substring.

Prototype: SUBSTR(string,beg[,end])

SUBSTR(string,beg | len)

Parameters:

string is a character string from which a substring is to be generated.

beg is an integer specifying the character position at which the substring is to begin. 1 means the first character position.

end is an integer specifying the character position at which the substring is to end.

len is an integer specifying the length of the substring from the beginning character position.

If neither *end* or *len* is specified, the remainder of *string* is taken as the substring.

Action: The SUBSTR pattern element generates a substring from the specified character string in *string*. For example, the following substrings are generated by the patterns given below:

```
SUBSTR("abcdefghi",2,4)  produces "bcd"
SUBSTR("abcdefghi",3|2)  produces "cd"
SUBSTR("123456789",6)    produces "6789"
```

Example: Assume a file with the following lines:

```
123456789012345
1234567890
123456769012
12345678901234
```

The following command will pad each line in the file with sufficient trailing blanks so that each line is exactly fifteen characters long.

```
ALTER@OPL /FILE COL=11 REM $ A = SUBSTR(A "     ",1|5)
```

Note: The characters assigned to the variable A by the REM pattern element are part of the subject string of the SUBSTR pattern element. Thus, for the third line of the file, the subject string to SUBSTR is “12 ” and the resulting substring is therefore “12 ”.

TAB

Edit Pattern Description

Purpose: To match all characters in the subject string until the current cursor position is at a specific column with respect to the beginning of the subject.

Prototype: TAB(column)

Parameter:

column is a nonnegative integer value specifying the column position to be matched up to.

Action: The current cursor position with respect to the left-hand edge of the string is compared with *column*. If the current cursor position is greater, TAB fails; otherwise, TAB matches the zero or more characters necessary to move the current cursor position to *column*.

TAB(0) moves the current cursor position to immediately before the first character position in the string. TAB(n) moves the cursor to immediately after the last character position in the subject, if the subject is "n" characters long.

Examples: The resulting pattern match in the subject line after matching the pattern:

```
TAB(8)

The case is a matter of public record; we do not
└──────┘
   └──TAB(8)
```

The resulting pattern match in the subject line after matching the pattern:

```
SKIPNOT(";") TAB(46)
The case is a matter of public record; we do not
└──────────────────────────────┘
             └──SKIPNOT(";")          └──TAB(46)
```

INDEX

Reader's Comment Form

The MTS File Editor

Volume 18

February 1988

Errors noted in publication:

Suggestions for improvement:

Your comments will be much appreciated. The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, UNYN, or NUBS.

Date: _____

Name: _____

Address: _____

_____

_____

Publications
Computing Center
University of Michigan
Ann Arbor, Michigan 48109
USA