PROGRAMMING GUIDE

PROGRAMMING GUIDE

This section covers the Command Mode in much greater detail. In particular it explains how the Command Mode may be used as a powerful text oriented programming language. It first introduces "*command macros*" which are the programs of VEDIT PLUS. It then covers the programming language topic by topic. Last, each command is described in detail.

Introduction to Programming

Even if you have never written computer programs before, you will find it easy and useful to write your own "programs" in VEDIT PLUS. As you will see, there is no real distinction between "commands" and "programs"; it can be said that a single command is just a very short program. If that doesn't satisfy your intuitive definition of a "program" you can follow the example in the Tutorial "Command Macros from Visual Mode". By all accounts, you have then written and used a real "program".

Programs are just sequences of commands. Some programs are quite short, such as the one in the Tutorial, while others can be quite long and sophisticated, such as the supplied COMPARE and SORT programs. Useful programs do not need to be long - many useful programs consist of less than ten commands and are just one line long!

Any program, no matter how long, can be entered directly from the keyboard while in Command Mode. Many short one line programs are in fact entered in response to the "COMMAND:" prompt - when you press **RETURN** the program is executed. However, it would be tedious and error prone to have to type in the entire program each time you wanted to execute it. Therefore, you can store programs in the text registers - one program per text register. The sequence of commands stored in any text register can then be executed, just as if you had typed them in by hand. Any sequence of commands executed in a text register is referred to as a "*command macro*" or just "*macro*" for short. Since all but the simplest programs are usually executed from text registers, we use the term "*command macro*" quite broadly to refer to all VEDIT PLUS programs.

Command Macros

Command macros are sequences of Command Mode commands stored in the text registers - it may be just a single command or hundreds of commands. Sequences of commands, i.e. "programs" which are used over and over again are usually saved as command macros. The entire program can then be executed with a single command. Command

macros can be saved on disk and re-used at a later time.

There is nothing special about the way command macros are stored in the text registers - there is no difference between text registers or even edit buffers that contain text and those that contain command macros. It is up to you to keep track of which text registers contain text and which contain commands. (You can accidentally execute normal text as a command macro, with unpredictable results.)

An existing command macro on disk is usually loaded into a text register with the "RL" command. A new macro is usually entered by editing the text register in Visual Mode. A newly entered macro can be saved on disk for later use by exiting the edit buffer, or with the "RS" (register save) command.

The commands in a text register are executed with the "M" command or from Visual Mode with the [MACRO] function.

```
M6                    Executes macro in register 6.
```

The 36 text registers allow complex macros to be broken down into several simpler ones, each stored in its own text register. This is analogous to the programming concept of "subroutines". Since a macro can itself contain the "M" command, it can execute other subroutine-like macros.

Command Macros in Visual Mode

Command macros let you create your own Visual Mode edit functions which are accessible via the [MACRO] function. The following short macros would be better handled as "keystroke macros" as described in the Tutorial. However, the following steps apply to those macros which are too large to be keystroke macros.

The following example is an edit function which duplicates the current text line on the next line. It also moves the cursor to the beginning of the new line. This saves you the time of typing a line of text which is identical or nearly identical to the previous line.

```
0L 1RC9 L RG9 -L      Macro to duplicate a line of text.
```

The second example is a cursor movement function which moves the cursor to the beginning of the next sentence.

```
@F/.¦S/ @F/¦F/ -C     Macro to move the cursor to the
                      beginning of the next sentence.
```

To access a command macro with **[MACRO]**, you must first place the commands in a text register you will not be using for other purposes. Notice that the first macro uses text register 9 for its operation; therefore, register 9 must also not be used for other purposes.

If you are repeatedly using a command macro (from Visual or Command Mode), it is easy to automatically set it up using the VEDIT.INI "auto-startup" file. For example, to set up the above two macros, create a VEDIT.INI file with the following commands in it (or add these commands to your existing VEDIT.INI file):

```
@RID;0L 1RC9 L RG9 -L;
@RIS;@F/.¦S/ @F/¦F/ -C;
```

After invoking VEDIT PLUS you can then immediately use the two macros from Visual Mode:

[MACRO]-D	To duplicate a line.
[MACRO]-S	To move cursor to next sentence.

Search and Replace in Multiple Files

This is an in-depth example of how to use command macros to automate the process of performing a large search and replace operation on several files.

This example assumes you have a long report written as ten separate files and that you have consistently misspelled 20 words. Correcting this could be a very time consuming editing operation, but it can be automated with two command macros. One macro contains the global search and replace for each of the 20 words. The second macro contains the commands to edit each of the ten files and, for each file, execute the search/replace macro. Once the two macros are entered and begin executing, you can take a coffee break while the 200 (10 times 20) operations are automatically performed. (ALWAYS make a backup copy of the files before performing complex macros. It is very easy for a small syntax error, a power failure or a disk error to destroy the files being automatically edited!)

For this example, the first macro will be created in the edit buffer "S". Use the **[WINDOW]-Switch** function or **"EE"** command to switch to edit buffer "S" and enter the following macro from Visual Mode: (*"word1"* is the first misspelled word and *"fix1"* is its correct spelling, etc.)

```
ES 8 1
ES 9 1
B#S/word1/fix1/
B#S/word2/fix2/
B#S/word3/fix3/
 .......
B#S/word20/fix20/
```

The first two commands specify that explicit delimiters are to be used and that search errors are to be suppressed. Since explicit delimiters are used, the <ESC> character is not needed anywhere. Search errors must be suppressed because, otherwise, if any word is not found the entire macro aborts.

The second macro reads in each of the ten files, executes the first macro, writes the file back to disk and continues with the next file. Switch to edit buffer "F" and enter the following macro:

```
EB file1.txt
MS
EY
EB file2.txt
MS
EY
 ........
EB file10.txt
MS
EY
```

With both macros entered you can now start the "program" with the command to execute text register "F":

```
MF                    Execute text register "F" as a macro.
```

It is often desirable to save macros on disk for future use. The commands to save these two macros are:

```
RSS macro1.vdm
RSF macro2.vdm
```

Similarly, the commands to retrieve them from disk are:

```
RLS macro1.vdm
RLF macro2.vdm
```

The commands to display them on the screen are:

```
RTS
RTF
```

ITERATION LOOPS

An iteration loop is a group of commands which repeats with or without user intervention as many times as desired. As its name implies, an iteration loop lets a sequence of commands loop over and over again.

An iteration loop's general construction is a group of commands enclosed by brackets "[" and "]" and preceded by an "*iteration count*", which specifies how many times the entire group of commands is to be executed. An iteration loop operates by executing the first command of the group through the last command and then starting over again with the first command.

The following example prints 10 pages with only 20 lines of text on each page:

```
10[20PR PE 20L]
```

The "20PR" prints 20 lines of text, the "PE" starts a new page and the "20L" moves the edit pointer over the text just printed. Due to the iteration loop, this sequence of commands is executed for a total of "10" times.

The "[" and "]" may also occur within each other for "*nested*" iteration loops. For example, the iteration "5[4T]" displays the same four lines over again five times. The iteration "3[5[4T] 4L]" displays the same four lines five times, then moves to the next four lines and displays them five times and last, moves to the next four lines and displays them five times.

Iteration Count

If no explicit "iteration count" is given, it defaults to "#" (65535) which signifies "forever" or "all". This is used when the iteration is to continue as long as possible. For clarity's sake, the "#" may also be explicitly specified. The following identical examples display (type) all lines which contain the word "teeth":

```
#[Fteeth$ 0T T]

 [Fteeth$ 0T T]      Short-hand for above.
```

It is normal to get the error message "CANNOT FIND ..." with iteration loops, such as the ones above, which search for all occurrences of a string, because they are literally searching for 65535 occurrences. Therefore, an iteration loop ends when its iteration count is exhausted or a search or other error occurs.

Search error handling is often suppressed in command macros. In this case, when a search is unsuccessful, no error is given, but the iteration loop is stopped and execution continues with the commands following the iteration loop. This may be an outer level iteration loop.

A similar situation occurs with an "L" command in iteration loops. If an "L" command attempts to go past either end of the edit buffer, it too stops the iteration loop. This is convenient for iteration loops which need to stop when the end of the text is reached. The supplied print macro "PRINT.VDM" is such an example.

Commands within Iteration Loops

Although a single "S" (substitute) command can be made into an iteration loop, this is not necessary because the "S" can itself be preceded by an iteration count. This also has the advantage of executing much faster. Therefore, the second command below is the preferable one:

Poor: **#[Steeths$teeth$]**

Faster: **#Steeths$teeth$$**

Note that the latter is not an iteration loop, but rather just a form of the "S" command.

Of course, the "S" command will commonly appear inside iteration loops which contain other commands too. The "T" command can be used to display the lines that are changed in an iteration loop without going into Visual Mode. For example, the command to change all occurrences of "teeths" to "teeth" and display those lines which changed is:

[Steeths$teeth$ 0TT]

("0T" displays from the beginning of the line up to the edit pointer and "T" displays from the edit pointer to the end of the line. Therefore, "0TT" is commonly used to display a line no matter where the edit pointer is on it.)

The "I" (insert) command, which does not take an iteration count, is often used singly in iteration loops. For example, the command to insert the text " enter me three times" is:

3[I enter me three times$]

It is very important to observe the placement of any necessary <ESC>'s to terminate strings and filenames in iteration loops. Filenames must always be followed by an <ESC> and all text strings

must end with an <ESC> unless you are using explicit delimiters. The following example shows a common error in which the string " enter me three times]" is entered into the text, which is not the intention (the "]" is erroneously part of the text string).

Wrong: **3[I enter me three times]**

Right: **3[I enter me three times$]**

If you are ever unsure, it is always safe to end a command with an <ESC> - any unneeded <ESC>'s are ignored.

The "I" command can be used to insert the same text into the edit buffer many times. For example, the following command creates a table of 60 lines, where each line begins with a <TAB> and ".....". The table can then be filled in Visual Mode.

60[I<TAB>.....<RETURN>$]

You could enter the above command as a macro by editing the corresponding text register. Since the **RETURN** starts a new line the "[" and "]" will appear on different lines - this works properly.

You could also enter the command directly in Command Mode following the "COMMAND:" prompt. Pressing **RETURN** starts a new line and gives you the "-" prompt as a reminder that it is waiting for the <ESC> delimiter. At this point the "I" command will also have been executed once. As soon as you type the "]" and another **RETURN**, the "I" command will execute another 59 times. It is allowable for an iteration loop to extend over several command lines. Just remember that the commands in the loop will be executed line by line as you enter them.

Iteration loops begin operation from the current edit pointer position. Therefore, be sure to place the edit pointer correctly before executing an iteration loop. As in the **"10[20PR PE 20L]"** example above, the edit pointer must often be explicitly moved within the iteration loop, commonly with the "L" command.

Using Visual Mode in Iteration Loops

Search and replace operations are often used in conjunction with the Visual Mode to edit the region, or to confirm that the replacement was done correctly. For example, the following command searches for all occurrences of the word "temporary" and lets those regions of the text be edited in Visual Mode.

```
[Ftemporary$V]
```

The following command could be used with a form letter to change "-name-" to the desired name, check that it was done correctly in Visual Mode and, if necessary, make any additional changes.

```
[S-name-$Mr. Jones$V]
```

The Visual Mode has two ways of exiting back to Command Mode in order to help in using iteration loops. **[VISUAL EXIT]** simply exits and lets any command iteration continue. **[VISUAL ESCAPE]** exits to Command Mode, but also aborts any iteration loop or command macro. The latter is used when you realize that the iteration loop is not doing what was intended and do not want to further foul things up. For example, to change all occurrences of the word "and" to "or", the following command may have been given:

Wrong: **[Sandorv]**

You might then see in Visual Mode that the word "sand" was changed to "sor", which was not the intention. Pressing **[VISUAL ESCAPE]** stops the iteration loop, and the following correct command can then be given:

Right: **[S and$ or$V]**

USEFUL COMMANDS IN MACROS

The following topics describe several commands which are primarily useful inside of command macros.

Displaying Text

The "YT" command displays (types) a text string on the screen. Its syntax is "YT*text*<ESC>" - '*text*' can be one or more lines long and must end in an <ESC> or explicit delimiter. "YT" can be used to display progress messages or debugging messages during the execution of command macros.

```
@YT/Part 1 is done/      Display a message on screen.
```

Other uses for the "YT" command include displaying user prompts and menus, such as in the supplied "MENU.VDM" macro.

The command "nYD" displays (dumps) a single character with decimal value 'n' followed by a <CR> <LF>. Alternatively, the command form "n:YD" dumps just the single character without the <CR> <LF>. It can be used to display special control and graphic characters since they are "dumped" to the screen and are not expanded.

```
1:YD 129:YD          Display two graphic chars (on IBM PC).
```

Displaying Input / Output Filenames

"ER" without a filename displays the input (read) filename and "EW" displays the output (write) filename on the screen. "EB" without a filename displays both the input and output filenames. The filenames are preceded by the messages "INPUT FILE:" and "OUTPUT FILE:" respectively and are followed by a <CR> <LF>. The preceding message and the following <CR> <LF> can be suppressed by preceding the command with a ":".

```
EW<ESC>        Display output filename and a <CR><LF>.

:EW<ESC>       Display output filename without <CR><LF>.
```

Re-routing Console Output

Any Command Mode console output, which normally goes to the screen, can be re-routed to either the printer or the edit buffer. Such re-routing is in effect until the next "COMMAND:" prompt or until re-routing is canceled.

The command "YP" re-routes console output to the printer. It is used in print formatting macros such as our supplied PRINT.VDM. The command "-YP" (or "0YP") cancels the re-routing and allows normal console output.

YP ED B:	Print the directory of drive B.
YP EW<ESC>	Print the output filename.

The "YP" command can be used in conjunction with the "YT" command to send page headers, carriage returns and form feeds to the printer. For example, the commands to send three blank lines to the printer are:

YP @YT/<RETURN><RETURN><RETURN>/

"YI" re-routes console output to the edit buffer. Each character is inserted at the edit pointer and the edit pointer incremented. "-YI" (or "0YI") cancels the re-routing. A simple example to try is:

YI EV	Insert version # into edit buffer.

This inserts the VEDIT PLUS version number into the edit buffer. Inserting text at the end of the edit buffer with the "YI" command operates very quickly. However, inserting text at the beginning of a large file may take as long as 1/2 second per character!

Extended "ED" Directory Display

The "ED" command first displays the current drive name and MS-DOS subdirectory. The filenames are then displayed in four (4) columns. The command form "-ED" displays the files one per line and without the drive and subdirectory line.

-ED	Display files one per line.

The "-ED" command is often used in conjunction with "YI" to insert the directory into the edit buffer. It is thereby possible for a command macro to determine what files are on disk and automatically edit those files. For example, the command sequence to insert all filenames with an extension of ".ASM" into the edit buffer is:

YI -ED*.ASM	Insert all ".ASM" filenames into the edit buffer, one per line.

The special form ":ED *file*" tests for the existence of the file '*file*' and sets the internal value ".rv" (described later) accordingly - "1" if the file is found and "0" if it is not found.

It does not display anything on the screen and is primarily used inside macros which must know if a particular file exists.

Suppress Error Handling

Normally when a "F" or "S" command is unsuccessful, it gives the error "CANNOT FIND *string*" and command execution stops. Similarly, if an "L" command attempts to go past either end of the edit buffer (file) it gives the error "END OF EDIT BUFFER REACHED" and stops execution.

These errors can be suppressed with the ":" command modifier - ":F", ":S" and ":L". When "suppressed", errors are still detected, but are handled differently. First, no error message is given; instead the ".er" error flag is set (described later under "Internal Values"). Second, command execution jumps out of any current iteration loop and continues with any following commands. If the command causing the error is not within an iteration loop, execution continues with the next command.

Search errors are often suppressed inside command macros. Macros often contain a sequence of "S" (substitute) commands which should not terminate the entire macro if some of the search strings are not found. Instead of specifying ":" for many commands, it is usually easier to make this command modifier the default by giving the command "ES 8 1" at the beginning of a command macro.

For additional flexibility, VEDIT PLUS has another option set with the command "ES 8 2". With this option, no error message is given and only the ".er" error flag is set. Command execution is not affected in any way. This option is available so that you can explicitly test the error flag ".er" and, if necessary, jump to another part of the macro with the command ".er JPlabel". (See "Jump on Search Error" for additional information on this option.)

In summary:

ES 8 0	"F", "S" and "L" command errors result in error message and command execution stops.
ES 8 1	"F", "S" and "L" command errors set the ".er" error flag and jump out of any current iteration loop.
ES 8 2	"F", "S" and "L" command errors only set the ".er" error flag and command execution continues with the next command. Must be used with care to prevent "infinite" loops.

Commenting Macros

"Commenting" is the useful practice of adding descriptive text (sentences and phrases) within a program (macro) to explain its operation. This helps other people understand how the program works, and will help you too, should you have to modify it sometime in the future.

There are two methods for adding comments to macros. One is to use the "**R***" command. Any text following the "**R***" to the end of the line is treated as a comment and is ignored during macro execution.

```
R* This text is a comment and is ignored by VEDIT PLUS
```

Alternatively, you can enclose comments between two "**!**" characters. As a convenience, any text on a line following a single "**!**" is also a comment. The "**!**" also serve as "*labels*", described later, but these two uses do not conflict.

```
!This is a comment, a "V" command follows! V

!This text to the end of the line is also a comment
```

Loading Macros into Text Registers

The command "**RIr***text*" can be used to load "*text*" directly into text register "r". "*Text*" is a text string which may be one or many lines long. The form "**RI+r***text*" appends to any text which is already in the register.

The main purpose for the "**RI**" command is for a command macro to insert text, usually another macro, into a text register. Each register of a multiple register macro could be loaded from a separate disk file, but this would be awkward and consume unneeded amounts of disk space. For example, instead of setting up eight registers from eight disk files, it is easier to just load one disk file and then set up the eight registers with eight "**RI**" commands. The later topic "Jumping To A Command Macro" describes this in more detail.

NUMERICAL CAPABILITY

VEDIT PLUS has extensive numerical capabilities. "Numbers" used in Command Mode can be "*constants*", "*variables*" and algebraic "*expressions*".

REMEMBER: The "on-line calculator" lets you evaluate any numeric constant, variable or expression by simply typing it at the "COMMAND:" prompt.

Numeric Constants

There are four types of numeric constants:

Type	Example	Description
Integer	345	A simple integer in the range 0 to 65535.
Signed Integer	-239 -59021	A signed integer in the range -65535 to 65535. (Integers are 17 bits wide.)
ASCII Constant	"A ^B	The value of an ASCII character or control character can be used as a numeric constant. The first example gives the value of the letter 'A', the second the value of **<CTRL-B>**.
#	#	This is a shorthand for the maximum integer 65535.

There is a distinction between "Integer" and "Signed Integer" because some commands take simple integers, while others take signed integers.

Examples - Numeric Constants

Many VEDIT PLUS commands may be preceded by a numeric constant. For example, the "T" command types (displays) lines of text:

12T	Type the next 12 lines of text.
-23T	Type the previous 23 lines of text.
#T	Type the rest of the text lines.

The following example shows one way to insert a <CTRL-S> into the text: (Type "^", then "S", then "EI")

^S EI	Insert a <CTRL-S> into the text using the "EI" command. (The space before "EI" has no effect on the command; it just improves the readability and can be left out.)

IMPORTANT - Numeric Notation

Normally, a numeric argument directly precedes a command, i.e "12T". However, a space can also appear between them to improve readability, i.e. "12 T". Many of the examples in this manual contain extra spaces.

12T	
12 T	Identical command to above. The extra space before "T" is optional and, in more complex commands, improves the readability.

Numeric Registers (Variables)

There are 100 numeric registers or "variables" named "0" thru "99". (CP/M versions have only 26 numeric registers.) Three commands access the numeric registers:

nXSr	sets numeric register 'r' to the value 'n'.
nXAr	adds the positive or negative value 'n' to register 'r'.
XTr	types the value of register 'r' in decimal, followed by <CR> <LF>. The command form ":XTr" suppresses the <CR> <LF>.

Additionally, the form "Qr" is used to access a numeric register. It can appear anywhere a numeric constant can appear.

Examples - Numeric Registers

12XS1 Q1 T	Types the next 12 lines of text. Note that the space before the "T" is optional!
12XS1 100XA1 Q1 T	Types the next 112 lines of text.

Q4 XS3	Copies the value of numeric register "4" to numeric register "3".
25XS1 45XS2 EG *file*[Q1,Q2]	Inserts lines 25 - 45 of '*file*' into the edit buffer. Shows that numeric variables (and expressions) can be used in the line range for the "EG" and "EL" commands.
Q4	The on-line calculator displays the value of register "4". In this context it is equivalent to the command "XT4". Note that "Q4" is not a command, but a numeric value.

Internal Values (Read Only)

Besides the 100 numeric register "variables", you can also access several internal "read-only" numeric values. You can use these internal values anywhere you can specify a numeric constant or a numeric register. The internal values are accessed by a "." followed by a one or two letter mnemonic. The internal values are:

.b Name of the edit buffer currently being edited: "0" - "9", "A" - "Z", or "@". The command ".bYD" displays which edit buffer is being editing.

.c ASCII value of the character at the edit pointer.

.ef End-of-file condition. Has value "1" if the end of the input file has been reached or has not been opened. Otherwise, has value "0" if input file is still open.

.er Value of the error flag. Cleared (value "0") before each command is executed. Set (value "1") by the "F", "S" and "EM" commands by a search/match error. Set by "L" when attempting to go past the end of the text. Note: since **.er** is cleared so quickly, we recommend using **.es** whenever possible.

.es Value of the search error flag. Similar to ".er", but is set/reset only by the "F", "S" and "EM" commands. Allows testing the results of a search many commands later.

.ew Write error flag set/reset by the last disk write operation. Has value "0" if there was no write error; has value "1" if there was a write error.

.f Number of free memory bytes. Same value as displayed by the "U" command.

.lm Value of the current left margin, i.e. the indent position in Visual Mode.

.m Name of text register containing the currently executing macro. In ASCII character form. Has value of 0 (zero) if no macro executing. (Register "0" is denoted by the ASCII code for "0" which is 48 decimal).

.n Number of characters matched by successful **F**, **S**, **EM** commands. Number of characters matched by **RM** command.

.of Output file open. Has value "1" if the output file is currently open. Otherwise has value "0".

.p The edit pointer's position (offset) in the current edit buffer. The position of the first character is "0". Note: this is not necessarily the position in the file, because part of the file may have been written to disk.

.rm Absolute value of the remainder from last division.

.rtR The "type" for register 'R': 0 = unused, 1 = text register, 2 = edit buffer.

.rv Return value from the commands **EM**, **EP**, **ES** and **RM**. Set by **":ED"** if the following filename is not found.

.t Position of the next tab based on the current value of **".x"**. Returns 255 if no more tab positions.

.ur Number of characters in text register '*r*'.

.v Numerical value of the expression at the edit pointer. The edit pointer is moved past the expression.

.vm The position (offset) in the current edit buffer for the "invisible" text marker set in Visual Mode corresponding to the "1 END" status line message.

.wa The screen attribute for text characters in the current window.

.wd The current display type:
0 = CRT terminal, 1 = Non-IBM memory mapped
2 = IBM Monochrome, 3 = IBM Color (CGA or EGA)

.we The screen attribute for erased (clear) portions in the current window.

.wh The horizontal size (in columns) of the current window.

.wn The name of the current window in ASCII.

.wt Total number of windows on the screen.

.wv The vertical size (in lines) of the current window.

.wx The horizontal ("x") position of the cursor in the window.

.wy The vertical ("y") position of the cursor in the window.

.x Horizontal column position for the character at the edit pointer. Same as displayed for "COLUMN" in Visual Mode.

.y Line number in the file for the line the edit pointer is on. Same as displayed for "LINE" in Visual Mode.

Do not be concerned if you do not understand all of these internal values yet. Several of them are return values from commands that are covered later.

Examples - Internal Values

Enter Visual Mode with any convenient file in the edit buffer and note the Line and Column numbers displayed on the status line. Exit back to Command Mode. By just typing the internal value designator, the on-line calculator will display its value.

```
.x          Displays same value as the Visual Mode column
            position.

.y          Displays same value as the Visual Mode line
            number.

.x XS2      Saves the current column position in numeric
XT2         register "2" and displays this value in decimal.

B           Positions the edit pointer 123 characters from
123C        the beginning of the edit buffer. ".p" then has
.p          the value of 123 assuming the edit buffer
            contains 123 characters. If it has less, the
            value of ".p" is the total number of characters
            in the edit buffer.

 f          Displays the number of free bytes, which is also
            the first number displayed by the "U" command.

 Z          Places the edit pointer at the end of the edit
.p          buffer. ".p" therefore has the same value as the
            second number displayed by the "U" command.
```

Expressions

You can use not only numeric constants and variables, but also expressions built from these constants and variables. Numerical expressions are similar to the types of operations you can perform on a simple calculator including addition, subtraction, multiplication and division. Besides numerical expressions, VEDIT PLUS also supports conditional expressions which evaluate to a truth value (TRUE or FALSE) for use with the Jump commands and decision making structures.

All expressions are a sequence of "*operands*" and "*operators*" that evaluate to a numeric value. "*Numerical expressions*" evaluate to a signed integer, while "*conditional expressions*" evaluate to just two numeric values - "1" which represents TRUE and "0" which represents FALSE. "*Operators*" are the functions such as addition, subtraction, multiplication and division. "*Operands*" can be numeric constants or variables (or internal values) or can themselves be expressions. Operands can, therefore, be of a numerical or conditional type.

Simple expressions can be combined with operators to create more complex expressions. The many operators in complex expressions are not evaluated left to right, but rather a rigid "*precedence*" determines the order in which operators are evaluated. This precedence can be overridden by using parentheses i.e. "(" and ")". Extra parentheses may be used to improve the readability of a complex expression.

Since all expressions evaluate to a numerical value, the difference between numerical and conditional expressions is the range of values they can evaluate to. This is determined by the types of operators in the expressions. There are three types of operators, and the following table lists the type of operands they take and the range of values to which they evaluate.

Numerical operators - take one or two numerical operands and evaluate to a signed integer.

Relational operators - take two numerical operands and evaluate to "1" meaning "TRUE" or "0" meaning "FALSE".

Logical operators - take one or two conditional operands and evaluate to TRUE or FALSE.

Examples of the three types of operators are:

Numeric:	12 / 3 + 7	The operators are "/" and "+" and the expression value is "11".
Relational:	35 > 14	The operator is ">" and the expression value is "1" or TRUE.
Logical:	(35 > 14) & (17 = 23)	The logical operator is "&" (AND) and the expression value is "0" or FALSE.

A complex expression may contain all three types of operators and the expression type is determined by the last operator evaluated. All operators can actually be used with any type of operand, but it is usually not meaningful to use conditional operands with numerical or relational operators. Similarly, it may not be meaningful to use numerical operands with logical operators.

Of primary importance in constructing expressions, is how the evaluated numeric value is used by the following command or flow control structure. If a command expects a numerical expression, but is given a conditional expression, the command will only see the two values "0" and "1". If a numerical expression is used where a conditional is expected, the results are generally predictable. The numeric values "1" and "0" will be interpreted as TRUE and FALSE. Generally, any other value will also be interpreted as TRUE.

Numerical Operators

The numerical operators are:

+	Addition
-	Subtraction (also performs unary minus function)
*	Multiplication
/	Division
%	Remainder of division
&	Bitwise AND
|	Bitwise OR
~	Bitwise complement (also called 1's complement)

Examples of these operators and the resulting values are:

Expression =	Value
12 + 19	31
54 - 36	18
-4 * 16	-64
14 / 4	3
14 % 4	2
14 & 7	6
14 ^ 7	15
25'	-26

In integer division "14 divided by 4" equals "3" with a remainder of "2". Following any division the remainder can be found in the internal value ".rm". If you are only interested in the remainder, use the remainder operator "%", which returns a signed remainder. This remainder has the same sign as the quotient.

The "-" operator performs subtraction when between two operands, or may precede an operand to change its sign. Note that no operator may immediately follow another. Thus "14/-4" is invalid and needs to be written as "14/(-4)".

All numerical values are 17 bits wide and therefore have a range of -65535 to 65535. Internally, addition and subtraction are computed with a 24 bit accuracy. Therefore:

50000 + 50000 + 50000 - 60000 - 60000 properly evaluates to 30000.

However, it is invalid for any final expression to have a value outside the allowed range and, except for such sequences of additions and subtractions, no intermediate value may fall outside the allowed range.

NOTE: If an invalid value occurs in an expression, division by zero occurs, or the expression is incorrectly written, the entire expression evaluates to zero (0).

Relational Operators

Relational operators are used in virtually every decision making function and every conditional expression must contain at least one relational operator. The relational operators are:

<	Less than
<=	Less than or equal to
=	Equal to
<>	Not equal to
>=	Greater than or equal to
>	Greater than

Examples of these operators and the resulting values are:

Expression	=	Value
4 < 12		1 or TRUE
-13 <= -4		1 or TRUE
Q1 = Q1+5		0 or FALSE
-9 <> 9		1 or TRUE
-5 >= 0		0 or FALSE
10 > 10		0 or FALSE

Logical Operators

The logical operators are:

&	AND	TRUE only if both operands are TRUE.
!	OR	TRUE if either operands is TRUE.
	NOT	Flips the truth value of the following operand.

The following examples show how the operators are used:

Expression	=	Value
1 & 1		1 or TRUE
1 & 0		0 or FALSE
0 & 0		0 or FALSE
1 ^ 1		1 or TRUE
1 ^ 0		1 or TRUE
0 ^ 0		0 or FALSE
~ 1		0 or FALSE
~ 0		1 or TRUE

Logical operators are used with conditional operands a
following examples. Numeric variables are used to m
examples more realistic. Note that numeric variables can
as conditional operands when they contain only the values
"0".

Assume: **Q1 = 12, Q2 = -7, Q4 = 0** (FALSE), **Q5 = 1** (TRUE)

Expression	=	Value
(Q1 > 10) & (Q2 <= -7)		1 or TRUE
(Q2 <> 0) & Q4		0 or FALSE (The right operand i
(Q1 = Q2) ^ Q5		1 or TRUE
(Q1 = Q2) ^ (~ Q5)		0 or FALSE
~ (Q4 & Q5)		1 or TRUE

You will notice that the logical operators for AND and OR
same symbols "&" and "^" as the numerical operator for bitw
and bitwise OR. "&" and "^" are really numerical op
However, when used with conditional operands, the bitwise
OR operators also perform the logical AND and OR functions.

Unlike the AND and OR operators, the numerical "Bitwise Complement" and logical "NOT" are not at all related:

0'	Evaluates to -1
~0	Evaluates to 1
25'	Evaluates to -26
~25	Evaluates to 0

Operator Precedence

In expressions involving two or more operators, the operators are not necessarily evaluated left to right, but rather in the order determined by a rigid precedence. You can override the precedence by using parentheses. Everything in the parentheses will be be evaluated before the entire parenthesized expression is itself used as an operand. Additional parentheses are often used to improve the readability of an expression since it is not always immediately clear what the precedence of operators will be. In the case of operators of the same precedence, the leftmost one will be evaluated first. The precedence of operators in VEDIT PLUS is the same as used by many programming languages and is:

Highest:	'	Complement
	* / %	Multiplication, Division, Remainder
	+ -	Addition, Subtraction
	< > = etc.	Relationals
	~	NOT
	&	AND (Both Bitwise and Logical)
Lowest:	^	OR (Both Bitwise and Logical)

ADDITIONAL NUMERIC FEATURES

Word Counting

The following command sequence counts the number of words in a file and displays the result. This is handy for a writer who needs to know the number of words in a manuscript.

```
0XS1                        Zero out register 1.
_B [_@:F/¦S¦A/ XA1]         Count words in entire file.
XT1                         Display the count.
```

Line Numbering

A numeric register can be used to count line numbers. For example, the following command sequence inserts 200 lines of the form "This is line number nnnn", where 'nnnn' increments for each line:

```
YI 1XS1 200[@I/This is line number / XT1 1XA1 ]
```

Some applications, such as BASIC programming, require line numbers at the beginning of each text line. It is usually easier to edit a file without line numbers and, when done editing, add the line numbers to the beginning of each line. The following command macro adds line numbers starting with "100" and with an increment of "10". Of course, you could choose any other starting number and increment. (Note: the following command lines are best entered into a text register and executed as a command macro.)

```
100XS1                      Set the starting line number.
B YI :XT1 @I/  /            Insert line number for first line.
[                           Start an iteration loop.
10XA1                       Add the increment.
:@F/<RETURN>                Find beginning of next line.
/ :XT1 @I/  /               Insert line number for this line.
]                           End of iteration loop.
-YI                         Stop inserting into text.
```

Additional On-Line Calculator Features

Besides evaluating just one expression, the on-line calculator can also evaluate multiple expressions per line. The expressions do not have to be typed in each time; they can be part of macros. In general, any expression which is not followed by a command is evaluated and displayed on the screen.

When evaluating multiple expressions, you can control whether they

are displayed one per screen line or all on the same screen line. The syntax is:

e1 $ e2 $ ... $ en	Each expression 'e1' through 'en' is evaluated and displayed on separate screen lines. '$' represents the <ESC> key, which forces each expression value to be displayed.
e1 :$ e2 :$ ... :$ en	Each expression 'e1' through 'en' is evaluated and displayed on the same screen line. The ':' character suppresses the <CR><LF>.

For example, you could display the values in four numerical registers on the same line with the command:

Q1 :$ Q2 :$ Q3 :$ Q4	Display the values in the numeric registers all on the same line.

"EP" and "ES" Parameter/Switch Values

The value of any "EP" parameter or "ES" switch can be accessed from within macros. An "EP" or "ES" command with just the first argument does nothing but save the current setting in the ".rv" internal value. When an edit parameter or edit switch is changed with "EP" or "ES", the old value is also saved in ".rv". The Command Mode menu macro uses this feature to display the current values of the parameters and switches.

EP	Display current values of all parameters.
EP 7	Set **.rv** to the current word wrap column.
EP 7 50	Save old word wrap value in **.rv**; set new value to 50.

For example, the command sequence to display the value of the word wrap column (EP 7 n) is:

```
@YT/Word Wrap Column is:/ EP 7<ESC> .rv
```

NOTE: Each edit buffer has its own set of **EP** and **ES** values.

Numeric Register as Text Register Name

The text register commands such as **"RCr"**, **"RGr"**, **"EEr"**, etc., operate on text register 'r'. You can also use the value of a numeric register to specify the desired text register. The form is: **"RC#r"**, **"RG#r"**, **"EE#r"**, etc., where 'r' is now the numeric register name - "0" through "25". The numeric register must hold the ASCII value of the desired text register. For example:

```
"5 XS3
RE#3
```

Empty text register 5. The ASCII for "5" is first placed in numeric register 3. "#3" then accesses the value of numeric register 3.

DIRECT EDIT BUFFER POSITIONING

You can save the position of the character at the current edit pointer in a numeric register so that this position may later be referenced. The commonly used commands "K", "T", "PR" and "RC" (Kill Lines, Type Lines, Print, Copy to Text Register), which normally operate on lines can also perform their operation between any two positions in the edit buffer. The "EJ" command can jump to any position in the edit buffer.

The position of the current edit pointer is accessed with the ".p" internal value. This position is an "offset" from the beginning of the edit buffer, with the first character having a position of zero "0". Using the on-line calculator you can easily determine the edit pointer's position by typing:

.p	Display edit pointer's current position as an "offset" from the beginning of the edit buffer.

You can save the edit pointer's position with the "**XSr**" command, where 'r' is the numeric register to save it in. For example:

.p XS4	Save edit pointer's position in register 4.

Note that this edit buffer position is not necessarily the same as the position in the file. If part of the file has been written to disk, position "0" will no longer refer to the first character in the file. Consequently you must be sure that no file writing takes place between the time you save any buffer positions and when you use them.

The command "**nEJ**" performs a "**jump**", moving the edit pointer to the 'n'th position in the edit buffer. Note that "**0EJ**" is equivalent to "B" and "**#EJ**" is equivalent to "Z". In practice, "EJ" is usually used to jump to the position saved in a numeric register:

Q4 EJ	Move (jump) edit pointer to position saved in numeric register 4.

Two Argument Commands

The commonly used "K", "T", "PR" and "RC" (Kill, Type, Print, Register Copy) commands can optionally operate between any two edit buffer positions with the command forms "*e1,e2*K", "*e1,e2*T", "*e1,e2*PR" and "*e1,e2*RC". The effective range is from the position specified by the numerical expression '*e1*' up to (but not including) the position specified by '*e2*'. These commands also move the edit pointer to the position '*e1*'.

4,9 K	Kill (delete) the 5th through 9th characters in the edit buffer. (Counting starts at zero.) Note that the space before "K" is optional.
123 XS1 234 XS2 Q1,Q2 RC7	Copy the 124th through 234th characters in the edit buffer into text register 7.
.p-24,.p T	Type out the previous 24 characters.

The counting scheme used with these commands may seem a little unnatural, but it actually works out well in practice, especially when used with the search commands, which is how the two edit buffer positions are usually set.

It may help to think of the "K", "T" "PR" and "RC" commands as performing a line oriented operation when they are preceded by a single expression and performing a character oriented operation when they are preceded by two expressions. Since a character oriented "K" command is more like a "D" command, the "D" command may also take two expressions, in which case it acts identical to the "K" command.

Setting the Edit Position by Searching

The search command "F" (Find) is often used to find some text, whose position is saved for further processing. A typical application is using one search to find the beginning of a block of text, saving the position, using a second search to find the end of the block, and then moving the block of text into a text register.

As an in-depth example, consider the task of extracting all mailing list entries with the name "Smith". Assume that the list is in the form used by the SORT macro. All such entries are to be appended to text register 4 and deleted from the current edit buffer.

First we need to find the word "Smith". This is a simple command:

@F/Smith/	Find the word "Smith" in the current edit buffer.

We do not want to save the current edit position, since our text would then not include the word "Smith". We need to backup to the beginning of the word. Since "Smith" should appear at the beginning of a line, we could backup with the command "0L". However, we want to examine the more general case. Since "Smith" is 5 characters long, we could save our edit position as follows:

```
.p-5 XS1
```
One way to save the beginning position of "Smith".

This would work fine, but for searches involving pattern matching, it may be unclear how many characters to backup. For the general case the internal value ".n" should be used. Its value is the number of characters matched by the last successful search.

```
.p-.n XS1
```
Preferred way to save beginning position of any search.

Next, we need to find the end of the entry, noting that entries are separated from each other by at least one blank line. Therefore, we can search for two Carriage Return - Line Feed pairs. The search command is:

```
@F/<CR><CR>/
```
Search for the end of a blank line.

We will save this position too with the following command:

```
.p XS2
```
Save position past end of desired text.

With the beginning and ending positions of the desired block of text saved, we are ready to append it to text register 4:

```
Q1,Q2 RC+4
```
Append the mailing list entry to register 4.

Since we also want to delete the entry from the edit buffer, we need the command:

```
Q1,Q2 K
```
Delete the mailing list entry just appended.

Finally, using an iteration loop to repeat this operation for all occurrences of "Smith", after first emptying text register 4, we have the command sequence:

```
RE4
[
@F/Smith/
.p-.n XS1
@F/<CR><CR>/
.p XS2
Q1,Q2 RC+4
Q1,Q2 K
]
```

MATCHING

Matching is the process of comparing the text at the edit pointer against either a "text string" or the contents of a text register. The comparison can be for "equality" or lexical "greater than" or "less than". In programming terminology, matching compares two "strings" - a "string variable" (at the edit pointer) against either a "string constant" or another "string variable" (in a text register/edit buffer). There are two match commands: "**EM**" and "**RM**".

"**nEM***string*" Compares '*string*' to the text at the edit pointer. The comparison is performed 'n' times. The comparison can be quite sophisticated because '*string*' may contain pattern match codes. "**EM**" has a form very similar to the "**F**" command.

The comparison is successful or "equal" when '*string*' completely matches the text at the edit pointer. (For "**nEM**" the '*string*' must match 'n' times in a row.) The text strings are considered "equal" even though the edit buffer can clearly be longer than '*string*'. If the strings do not completely match, the "**EM**" command sets the internal values "**.er**" and "**.es**" (error flags) to TRUE and computes a lexical "greater than" or "less than" for the two text strings.

The result of the comparison is saved in the internal value "**.rv**" (return value): "0" if successful; "1" if 'string' is "less than" text; "2" if 'string' is "greater than" text; "3" if the match failed on a pattern match code. The latter is needed because "greater than" or "less than" is meaningless with pattern matching.

Similar to the "**F**" command, "**EM**" moves the edit pointer past the characters which match '*string*' if (and only if) the match is successful. When successful, the internal value "**.n**" contains the number of text characters which were matched. If not successful, the edit pointer is not moved and "**.n**" is not changed.

'string'	Edit Buffer	Result	.rv	.n	.er
big	biggest...	Equal	0	3	0
biggest	bigger...	Greater	2		1
big dog	bigger...	Less	1	-	1
big¦A	biggest...	Equal	0	4	0
big¦A	big dog...	Not Equal	3	-	1

It may be helpful to consider how the "**EM**" and "**F**" commands are different.

1. **EM** does not search through the text trying again if the match at the current edit pointer is unsuccessful.
2. If the **EM** match is unsuccessful, this error is non-fatal and is only reported in the "**.er**" internal value.
3. The result of the match is also returned in "**.rv**". The "**F**" command does not change "**.rv**".

"**RMr**" Compares the contents of the text register (or edit buffer) 'r' to the text at the edit pointer. Performs direct character comparison without pattern matching; the Upper/Lower case switch (**ES 5**) is respected. The command "**_RMr**" performs auto-buffering, if needed, to complete the match, especially if 'r' is an edit buffer.

If 'r' is a simple text register, the comparison starts from the beginning of the register. If 'r' is an edit buffer, the comparison starts from 'r's edit pointer, and the edit pointer is moved past the last character matched. In either case, the rest of the text register must match for the comparison to be successful. The result of the comparison {0,1,2} is returned in "**.rv**" as with the "**EM**" command.

Unlike the "**EM**" command, "**RM**" moves the edit pointer past those characters which match the comparison register, regardless of whether the entire comparison is successful. The internal value "**.n**" is set to the number of characters which matched, regardless of success. "**.n**" is only set to "0" when the very first character does not match. It is not an error if the comparison is unsuccessful; therefore, the "**.er**" (error flag) is not affected.

The "**RM**" command is useful for moving the edit pointers in two edit buffers past all characters which match, even if the two edit buffers don't completely match. It is an important command in the "COMPARE.VDM" file comparison macro. The User section describes how to use the "**RM**" command to compare two files or blocks of text.

If you want to compare text against a text register, but want the characteristics of the "**EM**" command (i.e. pattern matching), use the "**|R**" pattern with the "**EM**" command.

FLOW CONTROL

The simplest type of macro consists of a sequence of one or more commands which are executed just once. For example:

@F/the/ OTT	This simple sequence of commands could be a macro. Notice that there is no flow control.

To execute a sequence of commands repeatedly, you must specify that the execution is to "loop" back to the commands which are to be repeated. Any such looping involves flow control - you are controlling the order (flow) in which commands are executed.

Independent of looping, you can also have decision making. The macro tests some condition - if it is TRUE the macro performs one function - if it is FALSE the macro performs a different function. For example, the description of a macro might be: "If the character is a lower case letter, change it to upper case; else leave the character unchanged". There is no looping involved in such a macro. However, decision making is often used together with looping. Consider the description: "Until the end of the file is reached, change all lower case letters to upper case, leaving all other characters unchanged". Any decision making also involves flow control.

Looping and decision making is performed primarily with "*Flow Control Structures*" and to a lesser extent with "*Jump*" commands. All flow control could be performed with Jump commands, but the use of many Jump commands tends to make a macro difficult to understand. The flow control structures make macros easier to write, easier to understand and easier to debug when necessary.

Flow Control Structures

"Iteration loops" are the basic structure from which all flow control structures are built. In review, an iteration loop is a structure of the form:

n[...]	The commands in an iteration loop are executed 'n' times. The iteration count 'n' may be any numerical expression.

In operation, all commands between the two outer brackets "[" and "]" are repeatedly executed 'n' times. This structure, therefore, performs looping, where the expression 'n' determines how many times the loop is performed. Often you will want to loop continuously until some special condition occurs. This can be done by just making 'n' a very large value. Remember that the default

value is 65535 if no 'n' is specified. Therefore, the form for a continuous loop is:

[...]	Loop "forever" or continuously until a special condition occurs.

The "special condition" might be the user pressing <CTRL-C> or, more likely, a condition such as not finding any more occurrences of a word. The latter condition does not need to be tested, since it is normally an "error" which automatically ends the iteration loop. Other conditions for ending the loop can be explicitly tested. In programming terminology, the operation of an iteration loop would be called a "REPEAT-UNTIL" looping structure.

If the iteration count has a value of "0" or FALSE, none of the commands between the brackets "[" and "]" are executed and execution immediately continues with any commands following the brackets. Since all condition testing (using conditional expressions) results in a value of "1" for TRUE and "0" for FALSE, the iteration loop can also be used as a decision making structure. It then has the form:

c[...]	Form of an **IF - THEN** decision structure. The condition 'c' may be any conditional expression.

If the condition is "1" or TRUE, the commands between the brackets "[" and "]" are executed once. If the condition is "0" or FALSE, the commands are not executed. Such a structure performs decision making where an operation is performed when the condition is TRUE. This is called "**IF-THEN**" decision making.

Very often in decision making you want to perform one alternative if the condition is TRUE and perform the other alternative if the condition is FALSE. This is called "**IF-THEN-ELSE**" decision making. A special form of the iteration loop performs IF-THEN-ELSE decision making. The form is:

c[...][...]	Form of an **IF-THEN-ELSE** decision structure. The condition 'c' may be any conditional expression.

If the condition is "1" or TRUE, the commands between the first set of brackets are executed once. If the condition is "0" or FALSE, the commands between the second set of brackets are executed once. In either case, execution then continues with any commands following the second set of brackets.

RETURN's can occur between commands and around the brackets "[" and "]" to improve their readability. However, the condition 'c' and the "[" bracket must be on the same line. Also, in an IF-THEN-ELSE

structure there must be no intervening characters between the "][". Therefore, an alternative form for the IF-THEN-ELSE structure is:

```
c[              Alternative form for an IF-THEN-ELSE
...             structure. Emphasizes that the brackets
][              "][" must appear together with no
...             intervening characters at all.
]
```

For more complex loops and decision making, these structures can occur within each other. This is called "*nesting*". Structures may be nested to a depth of 25.

The IF-THEN-ELSE structure allows two alternatives. Decision making often involves more than two alternatives. Consider the description: "If aaa then do vvv; else if bbb then do xxx; else if ccc do yyy; else do zzz". This can be implemented with a nested IF-THEN-ELSE structure, which is traditionally called a "**CASE**" structure. Its form is:

```
c1[              IF condition #1
][  c2[          ELSE IF condition #2
.                .
.                .
.                .
][  cn[          ELSE IF condition #n
][               ELSE
]]]              (n closing brackets)
```

Examples - Flow Control Structures

The following example of a **REPEAT-UNTIL** structure prints all text in the edit buffer with 50 lines printed per page. The "special condition" which ends the loop is the error caused by the "**L**" command reaching the end of the buffer.

```
                          REPEAT-UNTIL Example
[
50PR                      Print 50 lines
50L                       Advance by 50 lines in buffer
PE                        Start a new page
]
```

The following example of an **IF-THEN** structure displays the message "Letter", followed by the message "Thats All" if the character at the edit pointer is a letter. If the character is not a letter, it just displays the message "Thats All". Since **RETURN**'s are not allowed within expressions, the initial conditional expression is quite long.

IF-THEN Example

```
((.c >= "A) & (.c <= "Z)) ^ ((.c >= "a) & (.c <= "z)) [
                              Check if upper case letter
                              OR  lower case letter
@YT/Letter/                   Message if it's a letter
]                             End of IF-THEN
@YT/Thats All/                Final Message
```

The following example of an **IF-THEN-ELSE** structure displays the messages "Letter", followed by "Thats All" if the character at the edit pointer is a letter. If the character is not a letter, it display the messages "Not A Letter" followed by "Thats All".

IF-THEN-ELSE Example

```
((.c >= "A) & (.c <= "Z)) ^ ((.c >= "a) & (.c <= "z)) [
                              If upper case letter
                              OR  lower case letter THEN
@YT/Letter/                   Message if it's a letter
][                            End of THEN; beginning of ELSE
@YT/Not A Letter/             Message if not a letter
]                             End of ELSE
@YT/Thats All/                Final Message
```

The following example of a **CASE** structure displays the messages "Letter", followed by "Thats All" if the character at the edit pointer is a letter. If the character is not a letter, but is a numeric digit, it displays the messages "Digit", followed by "Thats ALL". If the character is not a letter and not a digit it displays the messages "Not Alphanumeric" followed by "Thats All".

CASE Example

```
((.c >= "A) & (.c <= "Z)) ^ ((.c >= "a) & (.c <= "z)) [
                              If upper case letter
                              OR  lower case letter THEN
@YT/Letter/                   Message if it's a letter
][                            End of THEN; beginning of ELSE
((.c >= "0) & (.c <= "9)) [   If a digit, THEN
@YT/Digit/                    Message if it's a digit
][                            End of THEN; beginning of ELSE
@YT/Not Alphanumeric/         Message if not letter or digit
]]                            End of first ELSE, second ELSE
@YT/Thats All/                Final Message
```

The following example of an **IF-THEN** structure within a **REPEAT-UNTIL** displays the line numbers of lines in the edit buffer which are more than 80 columns long (including TAB expansion). The macro uses the pattern match "¦>" to find the end of each line. The internal value ".x" is used to check the length of each line, and the value ".y" is used to indicate which line it is.

IF-THEN in **REPEAT-UNTIL** Example

```
B                       Start at begin of edit buffer
[                       Begin REPEAT-UNTIL
@F/¦>/                  Find end of current line
(.x > 81) [             Check if line is too long, THEN
@YT/Line #/ .y$         Display the long line number
]                       End of THEN
@F/¦L/                  Find begin of next line
]                       End of REPEAT-UNTIL
```

Jump (Branching) Commands

The order in which commands in a macro are executed can be controlled not only with Flow Control structures but also with "*Jump*" commands. The Jump command is analogous to the "GOTO" instruction in most programming languages.

The Jump commands are all conditional - the jump is only executed if the conditional expression preceding the Jump command is TRUE. However, a Jump command not preceded by an expression will unconditionally perform the jump. If the Jump command is preceded by a numerical expression, the jump is performed for any non-zero value.

The "**JP**" command performs a jump to a specified "*label*". The form for a label is:

!label! A label is any string of characters enclosed by exclamation marks "!". The entire label must appear on one line.

Labels may occur anywhere in a macro except within a text string. Since labels may exist without a corresponding Jump command, labels are often used for commenting a macro.

The five Jump commands are:

cJP*label*	If condition 'c' is TRUE, jump to "!*label*!"; otherwise continue processing any commands following '*label*'. '*label*' must end with an <ESC> or, alternatively, explicit delimiters may be used. You can jump out of a Flow Control structure, but you cannot jump into one.
cJL	If 'c' is TRUE, exit the current REPEAT-UNTIL structure. IF-THEN-ELSE clauses are recognized as such and are ignored when seeking the end of a REPEAT clause. However, brackets within text strings, if any, must be balanced.
cJN	If 'c' is TRUE, jump to the beginning of the current REPEAT-UNTIL structure and start the next iteration of it, so long as the count has not expired.
cJM	If 'c' is TRUE, exit the currently executing macro.
cJO	If 'c' is TRUE, cease executing any macro commands and return to the Command Mode prompt.

Jump on Search Error

Following the the command "**ES 8 2**", search and "**L**" command errors are suppressed where only the "**.er**" flag is set and command execution simply continues with the next command. This next command is often a "**.erJP***label*" to jump to another part of a macro. Alternatively, the command to jump out of an iteration loop on error is "**.er JL**". This common command has the abbreviation of ";". For example:

[@:F/help/ ;]	Jump out of iteration loop when "help" is no longer found. Equivalent to the commands **[@:F/help/ .erJL]**.

Note that the special ";" command is only needed when using the "**ES 8 2**" option. If you like, you can think of the "**ES 8 1**" option as placing a ";" after each "**F**", "**S**" and "**L**" command by default.

Notice also in the above example that if the ";" were inadvertently left out, the iteration would execute 65536 times regardless of the number of occurrences of "help". For this reason the "**ES 8 2**"

option must be used with care to prevent such nearly "infinite" loops. (You can break out by pressing <CTRL-C>.)

Jump when End-Of-File Reached

Many iteration loops will automatically end when the end of the file is reached due to an "L" command or search error. However, in other cases the End-Of-File condition has to be explicitly tested. When this condition is encountered, you can end the iteration loop with the appropriate Jump command.

The end of the edit buffer is marked with a <CTRL-Z>, and you can test if the edit pointer is at the end of the edit buffer with the expression:

(.c = ^Z)	This expression is TRUE if the edit pointer is at the end of the edit buffer. Otherwise it is FALSE.

Of course, the end of the edit buffer is not necessarily the end of the file. However, the internal value **.ef** is TRUE when the end of the edit buffer is also the end of the input file. Therefore, the expression to test if the edit pointer is at the end of the file is:

((.c = ^Z) & .ef)	This expression is TRUE if the edit pointer is at the end of the file. Otherwise it is FALSE.

Depending upon what you want to do when the End-Of-File is reached, you can use the above expression in front of the appropriate Jump command. Generally you will use the "**cJM**" command to exit the current macro. This common Jump command is:

((.c = ^Z) & .ef)JM	Exit the current macro when the edit pointer is at the End-Of-File.

As an elaborate example, we will build on the earlier CASE structure example which displayed "Letter", "Digit" or "Not Alphanumeric" depending upon the character at the edit pointer. The following example displays one of these messages for each character in the edit buffer. When the end of the edit buffer is reached, it displays the message "Thats All". Instead of a "**JM**" command it uses a "**JL**" command to jump out of the REPEAT-UNTIL loop in order to display the "Thats All" message.

CASE in REPEAT-UNTIL Example

```
B                                   Start at beginning of buffer
[                                   Begin continuous REPEAT-UNTIL

((.c >= "A) & (.c <= "Z)) ^ ((.c >= "a) & (.c <= "z)) [
                                    If upper case letter
                                    OR  lower case letter THEN
@YT/Letter/                         Message if it is a letter
][                                  End of THEN; beginning of ELSE
((.c >= "0) & (.c <= "9)) [         If a digit, THEN
@YT/Digit/                          Message if it is a digit
][                                  End of THEN; beginning of ELSE
@YT/Not Alphanumeric/               Message if not letter or digit
]]                                  End of first ELSE, second ELSE
C                                   Advance to next character
(.c = ^Z)JL                         Jump out if end of edit buffer
]                                   End of REPEAT-UNTIL
@YT/Thats All/                      Final Message
```

INTERACTIVE INPUT AND OUTPUT

Macros can be written to interact with a user. Messages, menus and prompts can be displayed on the screen, on the status line or in separate windows. Macros can accept input from the user in the form of single keystrokes, numbers, or entire lines of text. For fancier interaction, "forms entry" macros can be written.

Screen Control

Simple messages can be displayed on the screen with the "**YT***text*" command. Since the '*text*' may be several lines long, detailed "menus" can also be displayed. An "Input command" (described below) is then used to accept the user's selection. If desired, the window commands can be used to create a new window in which the menu is displayed and, after accepting the user input, the window can be deleted.

The Command Mode "cursor" can be positioned anywhere in the current window with the "**YEH**" and "**YEV**" commands.

"**nYEH**" Moves the cursor horizontally to column 'n'.

"**nYEV**" Moves the cursor vertically to line 'n'.

By first moving the cursor, new text can be displayed anywhere within the window, instead of just at the bottom of the window. This is useful for "forms entry" macros which paint the full screen and then move the cursor from one field to another.

For additional screen control, all or part of a window can be erased.

"**YEC**" Erases the entire window and moves the cursor to the upper left hand corner ("home" position).

"**YEL**" Erases from the cursor to the end of the window (screen) line.

"**YES**" Erases from the cursor to the end of the window (screen).

The erased parts of the screen use the "erase attribute" set with the "**YEA**" command - see the detailed "**YEA**" command description.

The command form "**+YT***text*" displays the text on the status line. In this case '*text*' must be a single line long and will normally be

displayed in reverse video.

"+@YT//"	This command can be used to clear the status line during macro execution. The status line will remain clear until the next "COMMAND:" prompt or until Visual Mode is entered.

Several "internal values" assist when manipulating windows inside of macros. ".wh" and ".wv" give the size of the current window. These values can be used if, for example, a macro is to split the current window in half. ".wa" and ".we" give the attributes (color) of the current window. You may want to copy these values to two numeric registers before changing colors so that you can restore the original colors later.

Input Commands

Macros can accept input in three forms using three commands. Similar to the BASIC language "INPUT" statement, each Input command includes a "*prompt*" to be displayed on the console for the user's benefit.

Command	Type of Input	Example of Input
@XKr/*prompt*/	Single Keystroke	Y
@XQr/*prompt*/	Number	-123
	Expression	(123 + 345) / 18 - 1
@RQr/*prompt*/	Text String	NEWFILE.TXT

Each command above is using the explicit delimiter option. This is not necessary, but otherwise there is no visual break between 'r' designating the register name and the '*prompt*'.

The command **@XKr**/*prompt*/ first prompts the user on a new console line with '*prompt*'. It then places the ASCII value of the next keystroke typed into numeric register 'r'.

The command **@XQr**/*prompt*/ first prompts the user on a new console line with '*prompt*'. It then evaluates the next expression, most likely a simple integer, and stores the result in numeric register 'r'. A **RETURN** or two **<ESC>**'s signal the end of the expression.

The command **@RQr**/*prompt*/ first prompts the user on a new console line with '*prompt*'. It then accepts an entire line of keyboard input, including the **RETURN** or two **<ESC>**'s signaling the end of the

line, and stores the line in text register 'r'. Remember that a **RETURN** is expanded into a **<CR><LF>** pair. The **<CR><LF>** pair or two **<ESC>**'s can be left off the stored line by using the command form:

:@RQr/*prompt*/	Store the input line in text register 'r', but without the **<CR><LF>** or two **<ESC>'s**.

Lines may be "appended" to the register with the command form:

@RQ+r/*prompt*/	Append the input line to text register 'r'.

The **":RQ"** command is frequently used to prompt the user for a filename. The filename in a text register can subsequently be used with the **"¦R"** pattern in place of '*filename*' with any command which takes a '*filename*'.

Using the special **"+"** command modifier on the **"XK"**, **"XQ"** and **"RQ"** commands causes the prompt to appear on the status line. In this case the '*prompt*' must be a single line long.

DEVELOPING COMPLEX MACROS

Writing Macros in Edit Buffers

When developing complex macros contained in multiple text registers, it is easiest to directly edit each text register; in the process making the text register into an edit buffer. Edit buffers may be used in the same way as text registers when executing command macros. However, two restrictions must be observed:

1. The currently active edit buffer may not be executed as a macro. It cannot be specified in the initial "**M**" command, nor from another executing macro. Doing so gives the error message "INVALID EDIT BUFFER OPERATION" and macro execution stops. To avoid this, simply switch to another edit buffer which does not contain macro commands (usually "@", the main edit buffer), before issuing the "**M**" command.

2. Although a macro may in general issue an "**EEr**" command to edit another edit buffer, it may not issue the command if edit buffer 'r' is itself an executing macro. Doing so stops execution and gives the error "MACRO ERROR IN r" where 'r' is the name of the text register containing the offending "**EE**" command. (Entering "??" at the command prompt shows the offending "**EE**" command in context).

Self-Modifying Command Macros

In general, VEDIT PLUS does not allow self-modifying macros; therefore, a macro may not modify the contents of its own text register.

A macro in one text register may itself contain "**M**" commands to "call" other macros (analogous to "subroutines") in other text registers. When this happens, the currently executing text register and the text registers which "called" it are both considered to be "executing". (Actually, one macro may call a second, which calls a third, etc., to a depth of 20.) More specifically, a macro cannot change the contents of any text register which is currently "executing". Otherwise, unpredictable results could occur. VEDIT PLUS checks for this possibility and, if it occurs, gives the error message: "MACRO ERROR IN r" where 'r' is the name of the text register containing the offending "**M**" command.

However, a macro can create or modify a macro in a text register which is not currently executing and, after modification, execute it. For example, it is common for a "main" macro to load other

macros from disk, or enter them into text registers with the "**RI**" command. After all the other macros are set up, they are then executed.

As a special exception, the command "**+REr**" empties text register 'r', even if it is executing. This can be used immediately following an "**RA**" (Auto-execute Register) command to empty the currently executing text register since it is no longer needed.

Jumping to a Command Macro

The "**Mr**" command performs a "call" to the "subroutine" macro in register 'r'. When the "subroutine" macro finishes, execution returns to any commands following the "**Mr**". As an alternative, the command "**RJr**" will "jump" to the macro in register 'r' without returning to the register containing the "**RJr**". The "**M**" and "**RJ**" commands are analogous to the normal programming language concepts of "CALL" and "GOTO", respectively.

The "**RJ**" command is used in the supplied "COMPARE.VDM" and "SORT.VDM" macros. These macros are first loaded into one text register, generally register "Z". (This can be done with auto-execution.) When executed, each macro first sets up all of the needed text registers using the "RI" command. Last, each macro uses the "RJ" command to "jump" to the register containing the main macro. Register "Z" is then emptied from the main macro.

This scheme has several advantages. It allows even the most complex macro to be saved in one disk file. You can save memory space by placing the majority of your comments before the actual commands. The "**RI**" commands can then be written to only load the actual commands and not the bulky comments. Following the "Jump" to the main macro, the initial macro is no longer executing and can be deleted by the main macro, saving a lot of memory space. In this way you will not pay a memory space penalty for the good habit of documenting your macros.

Auto-Execution of Macros

The Command Mode menu works by automatically executing the menu macro whenever the user attempts to enter Command Mode. This automatic execution or "locking in" of a text register is controlled by the command "**+RAr**" where 'r' is the text register to execute. Only register "0" may not be automatically executed, and the command "**RA0**" disables auto-execution. (Any 'r' which is not a valid register name, i.e. "**RA<ESC>**" also disables auto-execution.)

+RAY	Automatically execute register "Y" in place of the "COMMAND:" prompt.
RA0	Disable automatic register execution.

In effect, anytime VEDIT PLUS would normally present the "COMMAND:" prompt, it instead executes the specified text register. Use this feature carefully. Be sure to provide a method for the user to exit the macro to Command Mode, or to exit VEDIT PLUS. Otherwise the user will have to reboot the computer, losing any edit changes!

The "MENU.VDM" macro uses several tricks to get itself started. First, the file "MENU.INI" (which can be renamed to "VEDIT.INI") loads a few commands into register "Y" and then issues the command "**+RAY**" to "lock" into register "Y". Nothing further happens until Visual Mode is exited for the first time. Register "Y" then auto-executes and loads the menu macro into register "Z". It then issues the command "**+RAZ**" to "lock" into register "Z". Last, it issues the command "**+REY**" to clear itself out! Since there are then no further commands to execute, the menu macro starts executing in place of the "COMMAND:" prompt.

DEBUGGING MACROS

Hopefully you will have the opportunity to write your own command macros. Of course, if you write complex macros, the time will come when you have to debug one of your macros. The debugging of many macros is often obvious, involving just correcting typing errors or simple syntactic errors. This type of debugging just involves looking the macro over again.

Unfortunately, there will occasionally be macros, especially long and complex ones, whose flaws are not immediately visible. For those macros, VEDIT PLUS has several diagnostic features to help you in debugging them. The most useful feature is the ability to single-step through the macro execution, tracing the commands one by one. You know that a macro needs to be debugged when:

1. You receive an error message while a macro is executing.

2. There is no error message, but the results of the macro are not what you intended.

If you receive an error message, it may not be obvious which command caused the error. To display the most recently executed commands type "??" in response to the normal command prompt. You can also press <CTRL-C> to abort a macro and display the most recently executed commands.

?? Displays the most recent commands executed, ending with the last command executed. Be sure to enter "??" immediately following the command prompt. This helps you determine which command caused the error, and by what sequence of commands it got there. The displayed sequence of commands can be quite lengthy, so do not be dismayed if information scrolls off the screen. The message "(R*r*)" is also displayed, where '*r*' is the text register containing the most recent macro commands. If no macro was executing, "()" is displayed. Just the last few displayed commands are usually of interest, but you can often view the execution quite far back.

The Trace Mode

When the "??" command is not enough to determine the flaw in a macro, you can trace (or single step) through the macro command execution. The trace mode is enabled with the "?" command. You can place a "?" anywhere within a macro from where you want to begin tracing execution. In programming terminology, placing a "?" within the macro is called "*setting a breakpoint*". Alternatively, you can trace from the beginning of the macro with the command:

?Mr Begin tracing (single stepping) from the beginning of the macro in register 'r'.

When the "?" is encountered, trace mode is turned on. With trace mode on, one command at a time is executed. Before each command is executed, it is displayed on the screen, preceded by its evaluated numerical argument(s). For each command, VEDIT PLUS lets you control the tracing process with several single character choices:

RETURN Process the current command and remain in trace mode. This "single steps" through the commands.

SPACE Same as **RETURN** unless the current command is the macro command "**M**". in which case trace mode is disabled while the macro is executed; trace mode is resumed when the macro terminates. (Any "?"'s encountered in that macro will be ignored). This allows "single stepping" through an "**M**" command.

V This enters Visual Mode so that you can more readily see the current contents of the edit buffer. If desired, you can also perform any edit changes. Use **[VISUAL EXIT]** to return and continue tracing; use **[VISUAL ESCAPE]** to abort processing and return to Command Mode.

<ESC> Turns trace mode off and resumes normal execution with the current command.

<CTRL-C> Aborts processing and turns the trace mode off. Returns you to the Command Mode prompt.

- Displays more of the current command buffer: displays the next line of commands up to the next **RETURN** or the end of the command buffer. For each additional "?", an additional line of the command buffer is displayed.

-OTHER- Any other character is ignored.

Debugging Hints

While it is impossible to predict everything that could go wrong in writing command macros, here are a few hints to keep in mind when debugging.

* Check for missing/incorrect text delimiters. This could allow commands to be interpreted as text, or text to be interpreted as commands.

* Do not use the characters "[" and "]" in text strings or comments - the Jump commands and error handling can get confused. If necessary, place these characters into text registers and include them in the string with the "¦R" pattern.

* Remember that lines normally end in a Carriage Return and Line Feed - two characters.

* Do your search operations handle an unsuccessful search properly? Are search errors suppressed?

* Are files larger than available memory being handled correctly? Watch out when searching for text, part of which may be in memory and part on disk. Try adding the command "ES 11 1" to the beginning of the macro to see if it helps.

* Check how the end of the file (or edit buffer) is handled. Is it a special case? If it is, are you testing for the end of file properly?

* Check that the correct Jump commands are being used. Be careful not to mix up REPEAT-UNTIL with IF-THEN when using the "JL" and "JN" commands.

* Using the incorrect relation operator, such as "Greater Than" in place of "Greater Than or Equal" can lead to very subtle problems where the macro "works most of the time".

nA Append

Example: 100A 0A -0A

Description: This command appends 'n' lines from the input file to the end of the edit buffer. Fewer lines are appended if there is insufficient memory space for 'n' lines, or there are fewer than 'n' lines remaining in the input file. If 'n' is "0", an auto-read is performed, which reads all of the input file or until the edit buffer is almost full. The command can be issued (with 'n' not zero) after an auto-read to read in more of the file. The command is ignored if no input file is open. The input file can be opened with the **"EB"** and **"ER"** commands, or when VEDIT PLUS is invoked.

The special forms **"-nA"** and **"-0A"** read back 'n' lines from the output file to the beginning of the edit buffer. **"-0A"** reads all of the output file back or until the edit buffer is almost full. Nothing is read back if there is no output file or it is empty.

Notes: No indication is given if fewer than 'n' lines were appended. Use the **"U"** command to see if anything was appended. If the edit buffer is completely full, Visual Mode will not work well.

See Also: Commands: U, W, EB, EG, ER
Auto-buffering

Examples:

```
ER TEXT.DOC
```

0A	Opens the file "TEXT.DOC" and reads it all in, or until the edit buffer is almost full.
-0A	Reads back as much of the output file as will fit into the beginning of the edit buffer.

B Beginning

Example: B _B

Description: This command moves the edit pointer to the beginning of the edit buffer. The beginning of the edit buffer is not always the beginning of the file, especially when editing large files. Use the command "_B" to move back to the beginning of the file.

Notes:

See Also: Commands: EA, Z
Backward Disk Buffering

Examples: B 12T — Moves the edit pointer to the beginning of the edit buffer and types the first 12 lines.

mC Change

Example: 12C -4C

Description: This command moves the edit pointer by 'm' character positions forwards if 'm' is positive and backwards if 'm' is negative. The edit pointer cannot be moved beyond the beginning or the end of the edit buffer. Remember that every line normally ends in a <CR> <LF> (carriage return, line feed), which take up two character positions.

Notes:

See Also: Commands: D, L, EJ

Examples: Fhello$ -5C — Searches for the word "hello", and if it is found, positions the edit pointer at the beginning of the word.

mD Delete

Example: 12D -4D

Description: This command deletes 'm' characters from the edit buffer, starting at the current edit pointer. If 'm' is positive, the 'm' characters immediately at and following the edit pointer are deleted. If 'm' is negative, the 'm' characters preceding the edit pointer are deleted. Fewer than 'm' characters are deleted if either end of the edit buffer is reached.

Notes: Use the "K" command to delete entire lines of text.

See Also: Commands: C, K

Examples: 100[Fbikes$ -D] Deletes the 's' from up to 100 occurrences of the word 'bikes'.

E Extended Commands

Example: EX EV

Description: This is not a command by itself but just the first letter of the two letter commands beginning with "E", many of which have to do with file handling.

Notes: Other two letter commands begin with the letters "J", "O", "P", "R", "X" and "Y".

See Also:

Examples:

mF*string*<ESC> Find

Example: Fmispell$$ -10Fwords$$ F$$ @F/¦Sword¦S/

Description: This command searches the edit buffer, starting from the edit pointer, for the 'm'th occurrence of '*string*'. If 'm' is positive, the search is forwards and the edit pointer is positioned at the character following the 'm'th occurrence of '*string*'. If 'm' is negative, the search is backwards (toward the beginning of the file) and the edit pointer is positioned at the first character of the 'm'th occurrence.

If the 'm'th occurrence of '*string*' is not found, the error "CANNOT FIND *string*" is given (unless suppressed) and the edit pointer will be positioned at the last occurrence of '*string*' found, or be left at its original position if no occurrences were found. The command "**F$$**" searches for the previously specified '*string*'. The command switch "**ES 5**" ("Equate Upper/Lower case in search") determines if the search equates upper and lower case letters.

The form "**m_F***string*<ESC>" performs a global search to the end/beginning of the file instead of just to the end/beginning of the edit buffer. If the 'm'th occurrence is not found with a global search, it is possible that the previous occurrence is no longer in the edit buffer due to auto-buffering. In this case the edit pointer is positioned at the beginning of the edit buffer.

Notes: '*string*' may be up to 80 characters long and may contain "pattern matching codes". The internal value "**.n**" is set to the number of characters matched by '*string*'. The internal values "**.er**" and "**.es**" are set if the search is not successful. The "@" command modifier allows the use of explicit delimiters.

The command form "**#F***string*<ESC>" only gives an error if no occurrences of '*string*' are found. The error "NO OUTPUT FILE" occurs if no output file is open for performing the auto-buffering necessary for a global search.

See Also: Commands: S, EM, RM
Pattern Matching, Suppress Error Handling

Examples:

B Fhello$$	Searches for the word "hello" from the beginning of the edit buffer.
#[3Ffirst$ -5D Ithird$]	Changes every third occurrence of the word "first" to "third".
Z -100L Fend$$	Finds the word "end" if it occurs in the last 100 lines of the edit buffer.
#[@F/fix up/V]	Finds the next occurrence of the string "fix up" and enters Visual Mode. Any changes can be made in Visual mode. When **[VISUAL EXIT]** is pressed, the next occurrence of "fix up" is found and so on.
F$V	Searches for the next occurrence of the previously specified string and enters Visual Mode.

H Help

Example: H HES H¦

Description: This command performs interactive on-line help using the help file "VPHELP.HLP" which contains the Command Mode information in the "Quick Reference" section. The help file displays several menu screens of commands and topics for which help is available. The user enters the desired command or topic name and the selected help text is displayed. The help command then ends automatically. The user can also follow the "H" with the desired command or topic name and the menus will be skipped.

Notes: All three ".HLP" help files are user changeable and expandable. They have the capability to display menus and sub-menus. The internal structure of the help files is described under "Modifying On-line Help Files".

See Also: Commands: EH
Sample Edit Session, Modifying On-line Help Files

Examples:

H ES	Skips the help menu and directly displays the help text for the "ES" command, which consists of a summary of all the editing switches.
H¦	Skips the help menu and directly displays the help text for all of the pattern matching codes.

I*text*<ESC> Insert

Example: Ia word$$ I<CR>new line$$ -Ioverlay$$

Description: Inserts '*text*' into the edit buffer at the current edit pointer. The insertion is complete when the **<ESC>** is encountered; or explicit delimiters can be used. '*text*' may contain the **RETURN** key, which is expanded to **<CR> <LF>**. The edit pointer is moved just past the inserted text.

If insufficient memory space exists for '*text*', the error "*BREAK*" is given and only part of '*text*' will have been inserted.

The inserted text does not overwrite any existing text unless the command form "**-I***text***<ESC>**" form is used. The "**-I**" (and "**-EI**") command executes more quickly than the equivalent sequence of an "**I**" ("**EI**") command followed by a "**D**" command to delete the unwanted characters. However, the difference is only significant when performing hundreds or thousands of such operations in a large file.

Notes: Control characters can be inserted by preceding them with the literal character **<CTRL-Q>**. The "**@**" command modifier allows explicit delimiters to be used. Explicit delimiters must be used to insert an **<ESC>** character. The **<TAB>** key is not expanded with spaces as is optional in Visual Mode.

See Also: Commands: EI

Examples:

200[I<CR><TAB>$] Inserts 200 new lines, each beginning with a tab character.

Iunder<CTRL-Q><CTRL-H>_$$ Inserts the text "under", a BACKSPACE and the underline character. This will underline the "r" on some printers.

@I/a word/ Inserts the text "a word" into the edit buffer.

-Ioverwrite$ Overwrites the existing text with the text "overwrite", beginning at the edit pointer.

@I/EP 7 70<ESC><CR>/ Inserts the command line "EP 7 70<ESC>" into the edit buffer, including a **<CR><LF>**.

mK Kill

Example: 4_K -3K OK Q1,Q2 K

Description: This command kills (deletes) the specified number of lines. If 'm' is positive, all text from the current edit pointer up to and including the 'm'th **<LF>** is deleted. If 'm' is negative, all text preceding the edit pointer on the current line and the 'm' preceding lines are deleted. If 'm' is "0", all characters preceding the edit pointer on the current line are deleted. Fewer than 'm' lines are killed if either end of the edit buffer is reached. The global command "**m_K**" performs auto-buffering, if necessary, to delete the specified lines.

The command form "**p,qK**" deletes all characters from the 'p'th character in the edit buffer up to, but not including the 'q'th character. Counting starts with 0.

Notes: The command "**p,qK**" is equivalent to "**p,qD**".

See Also: Command: D, T
Two Argument Commands

Examples:

#[_Ftemp line$ OL K]	Kills all lines which contain the string "temp line".
-#_K	Kills all text before the edit pointer.
#RC6 #K	Saves all text in the edit buffer following the edit pointer in text register 6 and then deletes it from the edit buffer.

mL Lines

Example: 120L -14L 0L 1000_L

Description: This command moves the edit pointer by 'm' lines. If 'm' is positive, the edit pointer is moved to the beginning of the 'm'th following line. If 'm' is negative, the edit pointer is moved to the beginning of the 'm'th preceding line. If 'm' is "0", the edit pointer it moved to the beginning of the current line. The global command "**m_L**" moves by 'm' lines in the file, performing auto-buffering if necessary. Attempting to move past either end of the edit buffer (or file) leaves the edit pointer at the respective end and gives the error "END OF EDIT BUFFER REACHED" (unless suppressed).

Notes: The command form "**m:L**" suppresses the error message if either end of the edit buffer is reached. However, if this error occurs inside an iteration loop, it ends the current iteration level - execution continues with the commands (if any) following the iteration loop.

See Also: Commands: C, T
Suppress Error Handling

Examples:

2200_L	Moves down 2200 lines in the file performing auto-buffering, if necessary. If the edit pointer is at the beginning of the file, this moves to the beginning of line 2201.

Mr Macro

Example: M1

Description: This command executes the contents of text register 'r' as a command macro. Any legitimate command or sequence of commands may be executed as a macro. Macros are most easily created in an edit buffer and edited in Visual Mode. They may also be loaded from disk with the "**RL**" command or inserted with the "**RI**" command. A macro may invoke another macro, which in turn may invoke another, up to a nesting depth of 20. Macros are very convenient for holding long command sequences which are repeatedly used, saving the effort of retyping them each time.

Notes: **RETURN**'s may be used to separate commands in order to improve readability. The error "MACRO ERROR IN r" results if a macro attempts to change a text register which contains the executing command macro. An "**M**" without a following register name is interpreted as "**M0**". See Notes for "**RJ**" command.

See Also: Commands: RI, RJ, RL, RS
Command Macros, **[MACRO]** function

Examples: See the Tutorial "Command Macro" for an example.

nNss<ESC> Next

Example: Nbad line$$ 3@N/third/ N$$

Description: This command is a short hand for the global search command "**_F***ss*<ESC>".

Notes: The long form "**_F**" should be used in any macros you save on disk since the "**N**" command is likely to be changed in a future release.

See Also: Command: F

Examples: [3Nfirst$ -5D Ithird$] Changes every third occurrence of the word "first" to "third" in the rest of the file.

[3@N/first/ -5D @I/third/] Same as above, but using explicit delimiters.

nS*ss*<ESC>*text*<ESC> Substitute

Example: Stypo$type$$ #Sname$Mr. Smith$$ _Sold$new$$

Description: This command performs 'n' search and replace (substitute) operations. Each operation consists of searching for the next occurrence of '*ss*' in the edit buffer and changing it to '*text*'. An error is given if '*ss*' is not found. The edit pointer is positioned after '*text*', if '*ss*' is found, or else is left at its previous position if not found. For the command form "**#S***ss*<ESC>*text*<ESC>" an error is only given if no occurrences of '*ss*' are found. The form "**n_S***ss*<ESC>*text*<ESC>" performs a global search and replace, searching to the end of the file if necessary.

Notes: All Notes for the "**F**" command apply here too. A command like "**#Sfishes$fish$$**" executes much faster than the equivalent command "**#[Sfishes$fish$]**". There is no backward search and replace command. If there is insufficient memory space for inserting '*text*', '*ss*' will have been changed to as much of '*text*' as possible and the "*BREAK*" error is given.

See Also: Commands: F, I
Suppress Error Handling

Examples: #Stypo$type$$ Changes all occurrences of "typo" to "type".

#[Stypo$type$ OTT] Changes all occurrences of "typo" to "type" and types out every line that was changed.

ES 9 1
#[S/typo/type/ OTT] Alternate form of above command. "ES 9 1" allows explicit delimiters to be used without the "@" prefix.

#[Sname$smith$V] Changes the next occurrence of "name" to "smith" and enters Visual Mode. Any changes can be made in Visual Mode and when **[VISUAL EXIT]** is pressed the next occurrence of "name" is searched and so on.

#_Sgarbage$$ Deletes all occurrences of "garbage" from the rest of the file.

mT Type (Display) Lines

Example: 14T -6T 0T 66_T

Description: This command types out (displays) the specified lines. If 'm' is positive, all characters from the edit pointer up to and including the 'm'th **<LF>** are typed. If 'm' is negative, the previous 'm' lines and all characters up to the edit pointer are typed. If 'm' is "0", only the characters on the present line preceding the edit pointer are typed. Fewer than 'm' lines are typed if either end of the edit buffer is reached. This command does not move the edit pointer and is useful in iteration loops for displaying selected lines.

The global command "**m_T**" performs auto-buffering, if necessary, to display the specified number of lines. At the end of the command auto-buffering is again used, if necessary, to restore the edit pointer to its original position. The print formatting macro "PRINT.VDM" is a good example of using the "**_T**" command.

The command form "**p,qT**" types all characters from the 'p'th character in the edit buffer up to, but not including the 'q'th character. Counting starts with 0.

Notes: The commands "**0TT**" display the current line regardless of the position of the edit pointer on it.

The "**T**" command displays control characters in expanded format (i.e **<CTRL-A>** is displayed as "^A") and **<ESC>** is displayed as a "**$**".

See Also: Commands: PR, RT, YP

Examples: B #[Fmoney$ 0TT] Displays every line in the edit buffer with the string "money" in it.

U Usage (Memory)

Example: U

Description: This command displays the number of memory bytes free for use by the current edit buffer, followed by the number of characters in the edit buffer, followed by the combined number of characters in all the text registers which are NOT edit buffers. (Each character takes up one memory byte.)

Notes: The first two numbers will not always add up to the same total, since several other smaller buffers also use the same memory space. If the number of free bytes goes below 1028 (260 on 8080/Z80 versions), the FULL message is displayed during Visual Mode.

See Also: Commands: RU

Examples: 100A U An additional 100 lines from the input file are appended and the remaining number of free bytes displayed.

V Visual Mode

Example: v

Description: This command enters Visual Mode. The visual cursor position is set from the current edit pointer position. Visual Mode is exited with either the **[VISUAL EXIT]** or the **[VISUAL ESCAPE]** function. At that time the edit pointer is set from the cursor position.

The error "UNABLE TO ENTER VISUAL MODE" is given if the keyboard decode table is invalid. This is most likely due to loading the wrong file with the "YL" command. If this happens load a valid table or give the command "EXA" to save all files and exit VEDIT PLUS.

Notes: The text registers are preserved.

See Also: Visual Mode

Examples: Fhere$ V Finds "here" and enters Visual Mode.

nW Write

Example: 20W #W 0W -100W

Description: This command writes 'n' lines from the beginning of the edit buffer to the output file, which also deletes these lines from the edit buffer. If there are less than 'n' lines in the edit buffer, the entire edit buffer is written out. If 'n' is zero, the entire edit buffer up to the line the edit pointer is on, is written out. If no output file is open, the error "NO OUTPUT FILE" is given and no text is written. The output file can be opened with an "**EW**" or "**EB**" command or when VEDIT PLUS is invoked.

The special forms "**-nW**" and "**-0W**" write the end of the edit buffer to the temporary ".R" file. These commands are primarily used to make more memory space available for further edit operations. "**-nW**" writes the last 'n' lines in the edit buffer out to disk. "**-0W**" writes out the end of the edit buffer beginning with the line the cursor is on.

Notes: No indication is given if less than 'n' lines were written.

See Also: Commands: A, EB, EN, EW, EX

Examples:

```
EW part1.txt
24W
EF
EW part2.txt
EX
```

Writes the first 24 lines of the edit buffer to the file "PART1.TXT", writes the rest of the edit buffer to file "PART2".TXT", saves the files and exits VEDIT PLUS.

Z Zip

Example: Z _Z

Description: This command moves the edit pointer past the last character in the edit buffer. The command "_Z" moves the edit pointer past the last character in the file, performing forward disk buffering if necessary.

Notes:

See Also: Commands: B,
Disk Buffering

Example:

Z -100L	Positions the edit pointer to the 100th line before the end of the edit buffer.
Z -12T	Types out the last twelve lines in the edit buffer.
_Z -12T	Types out the last twelve lines in the file.

EA Edit Again

Example: EA

Description: This command saves the entire file being edited (including any un-read portion of the input file) to disk. It then begins a new edit session on the same file. Your original edit position in the file and any text markers are maintained. In effect, it saves your file on disk without disturbing anything else. The main purpose for this command is to save all of your edit changes as a safeguard against losing the file due to a user error, hardware, software or power failure.

All file backup and renaming is performed as with the "EY" or "EX" commands. This command is functionally similar to an "EY" followed by an "EB" on the file being edited. The contents of the text registers are not affected by the "EA" command.

Notes: The "EA" command starts a new edit session. Therefore, if you quit with an "EQ" sometime after an "EA", you will only abandon those changes made after the "EA" command. Those changes made before the "EA" command will have already been saved on disk.

The "EA" performs the identical operation as the [FILE]-Save function.

See Also: Commands: EX, EY
[FILE] function

EA _B V	Saves the current file on disk, moves the edit pointer to the beginning of the file and enters Visual Mode. Useful for periodic saving of ongoing work.

EB*file*<ESC> Edit Backup

Example: EB file.txt EB orgfile.txt newfile.txt

Description: When only one file is specified, it opens '*file*' for both input and output and then reads in all of the file, or until the edit buffer is nearly full. If two files are specified, it opens the first file for input and opens the second file for output.

"**EB** *file*" is similar to invoking VEDIT PLUS with the command:

VPLUS *file*

It is also similar to the sequence of commands:

ER *file*$ **EW** *file*$ **OA**

"**EB** *file1 file2*" is similar to invoking VEDIT PLUS with the command:

VPLUS *file1 file2*

The message "NEW FILE" is displayed if the file does not already exist and is therefore created. If an output file is still open, the error "CANNOT OPEN TWO" is given and the command has no other effect.

Notes: The term "backup" is used here to describe this command since the term is often used by other editors to perform a similar operation. Remember that VEDIT PLUS always renames an existing file to ".BAK" when it creates a new file by the same name.

See Also: Commands: W, ER, EW

Example:

EY EB newfile.txt	The current file being edited is saved on disk and the file "NEWFILE.TXT" is prepared for editing - it is opened for input and output and read in.
ER part1.txt$OA EB part2.txt	The file "PART1.TXT" is read into the edit buffer, the file "PART2.TXT" is then made the current input and output file and is appended to the end of the previous file "PART1.TXT".

EC Change Disks

Example: EC

Description: This command must be given before you attempt to change any logged-in disks in order to recovery from a disk write error, or to read files from another disk. It gives the error "CANNOT OPEN TWO" if the output file is open and the error "REV FILE OPEN" if the backwards buffering ".R" file is open - you cannot then change disks. Be sure that any input file has been completely read into memory before issuing the "EC" command because it also closes any input file. This command can also be used to switch to another disk before an "ER" or "EG" command.

If neither the output file nor ".R" file is open, the command gives the prompt "INSERT NEW DISK AND PRESS <RETURN>". At this time change the disk(s) and press **RETURN**.

Notes: The very careful user can change disks without using the "EC" command. For example, if you are editing on a hard disk, it is safe to read different disks in the floppy drive without issuing an "EC" command. Just be sure never to remove a disk which contains the output file (".$$$" file) or the ".R" file or any unread portion of the input file. The "EC" command must be used under CP/M if you intend to write to a disk which is not currently logged in - otherwise you will get a fatal BDOS error and loose your edit session!

See Also: Commands: ED, EF
Disk Write Error Recovery.

Example: EC Prompts you to enter the new disk(s) and press **RETURN**.

ED / ED*file* Disk Directory

Example: ED EDA: ED B:*.TXT 1ED

Description: This command displays the directory of files on any drive and/or MS-DOS subdirectory. It serves as a reminder of the names of files you wish to edit, merge or have already written out. Drive specifiers, the "?" and "*" wildcard characters and MS-DOS pathnames (or CP/M User Numbers) are allowed.

The command form "**nED**" displays the directory in 'n' columns instead of the normal 4 columns. The command form "**-nED**" also displays the directory in 'n' columns, but suppresses the normal header line which displays the current drive and subdirectory.

The command form ":**ED** *file*" displays nothing and only tests if the file '*file*' exists. If it exists it sets the internal value "**.rv**" to "1", otherwise it sets "**.rv**" to "0".

Notes: Use the "**EU**" command to change to a different drive or MS-DOS subdirectory (or CP/M User Number).

See Also: Commands: EU

Example:

ED B:*.TXT	Gives the directory of all files with extension ".TXT" on drive B.
ED B:/LETTERS/	Gives the directory of all files in subdirectory "LETTERS" of drive B.
YI -ED	Inserts the directory into the edit buffer one file per line.

EEr Switch Edit Buffer

Example: EE4 EE. -EE

Description: This command switches to edit buffer 'r'. If text register 'r' is not already an edit buffer it is converted into an edit buffer and will remain one. The commands "**EE@**", "**EE.**" or "**EE<RETURN>**" switch back to the main edit buffer.

If the new edit buffer is in a different 64K memory segment from the current edit buffer, the switch occurs without any file buffering. Otherwise, the "**EN**" command is internally called to make more memory free by buffering part of the current edit buffer back to disk. By default the current edit buffer is reduced in size to one file "page". The command form "**nEEr**" makes 'n' bytes free for the new edit buffer. The command form "**-EEr**" switches to edit buffer 'r' and suppresses any file buffering.

Notes: "**EE**" only switches to a different edit buffer, it does not itself switch to a different window although Visual Mode may use a different window for the new edit buffer. Use "**YWS**" to switch to a different window. The file buffering feature applies mostly to 8080/Z80 versions and when you have many edit buffers open on IBM PC/8088/8086 versions.

See Also: Commands: EN, YWS
Multiple File Editing, Memory Management

Example:

EE4	Switches to edit buffer 4.
-EE.	Switches back to the main edit buffer and suppresses any file buffering.

EF Finish (Close Output File)

Example: EF

Description: This command closes the output file, which only saves text that has ALREADY been written to disk. This command DOES NOT actually write any text to disk. Any existing file on disk with the same name as the output file is backed-up by renaming it with a file extension of ".BAK". The user is prompted for confirmation to close the file. The command "**EFY**" skips the confirmation. The error "NO OUTPUT FILE" is given if no output file is open.

WARNING

Use this command with care! You can easily erase the file you are editing! DO NOT USE THIS COMMAND TO QUIT YOUR EDITING - IT WILL ERASE YOUR FILE! In general, "**EF**" is only used to split large files into small ones and, for this, it is preceded by "**EW**" and "**W**" commands (See Tutorial "Splitting a File").

Notes: Technically, a file has to be "written" and "closed" in order to save it on disk. The commands "**EX**" and "**EY**" write text to disk AND close the file. In contrast, "**EF**" only closes the file; it writes nothing. Text can be written to disk by auto-buffering or with the "**W**" command. "**EF**" is primarily used when splitting a large file into smaller files.

Since the output file is initially opened with the file extension ".$$$", the ".$$$" file is first closed, then any existing file on disk with the same name as the output file is renamed to ".BAK" and, last, the ".$$$" file is renamed to the true output file name. (See EW command notes.)

See Also: Commands: EW, EX, EY

Example:
```
EW save.txt
100W EF
```
Writes the first 100 lines of the edit buffer to the file "SAVE.TXT" and closes it to save it.

EG*file*[*line range*] Get External File

Example: EGfile.txt[1,100] EG file.txt

Description: This command inserts a specified line number range of the file '*file*' into the edit buffer at the edit pointer. If no line range is specified, the entire file is inserted. If insufficient memory exists to insert the entire file segment, as much as possible is inserted and the *BREAK* error is given.

Notes: The line numbers of a file can be displayed using the "**EL**" command. A space may be used instead of a comma in the "line range". As with all file names in VEDIT PLUS, an optional drive and MS-DOS pathname (or CP/M user number) may be specified. More memory can be freed in the edit buffer with the "**EN**" command.

Be sure not to leave a space between the filename and the "[". If you accidentally do, it will insert the entire file and treat the line range as a harmless iteration loop to be repeated 65536 times. You can break out of this almost "infinite loop" by pressing **<CTRL-C>**.

See Also: Commands: A, EL, ER

Example: EG library.asm[34 65]
Lines 34 through 65 of the file "LIBRARY.ASM" are inserted into the edit buffer at the edit pointer.

EH Extended Help

Example: EH EH EXIT

Description: This command performs interactive on-line help using the help file "VPEHELP.HLP" which contains help information for common Command Mode tasks. Except for using a different help file, this command is identical to the "**H**" command; thus "**EH**" may be immediately followed by the desired topic name.

Notes: Once familiar with VEDIT PLUS, you may not need the information contained in "VPEHELP.HLP" any further. In this case you may want to create a help file for another program, such as the V-PRINT print formatter, and make this file accessible by the "**EH**" command. This procedure is described under "Modifying On-line Help Files" in the User Guide.

See Also: Commands: H
Modifying On-line Help Files

Examples: EH EXIT Skips the help menu and directly displays the help text for the topic on how to "exit" VEDIT PLUS.

nEI Insert Single (Control) Character

Example: 12EI 0EI -45EI

Description: Inserts the character whose decimal value is 'n' into the edit buffer at the edit pointer. This is useful for inserting special control characters, especially characters which cannot be generated from the keyboard, such as the IBM PC graphics characters with a decimal value between 128 and 255. Only the "End of File" marking character with a value of 26 cannot be inserted. The command form "**-nEI**" overwrites the existing character.

Notes: The non-ASCII character with value 255 should not be used. This character is internally used as a "Null" and will be deleted by the "S" command.

The command "**0EI**" is the only way to insert the character with value "0" from the IBM PC keyboard.

See Also: Commands: I
[NEXT CHAR LITERAL] function

Example:

8EI	Inserts a backspace character into the edit buffer at the edit pointer.
132EI	Inserts a graphics character into the text with the EI command, because it cannot be generated from the keyboard.

nEJ Jump (To Position)

Example: 100EJ Q1 EJ

Description: This command moves (jumps) the edit pointer to position 'n' in the edit buffer. 'n' is usually the value of a numeric register containing a computed position. Position "0" is the first character in the edit buffer.

Notes: The command "**0EJ**" is equivalent to "B" and "**#EJ**" is equivalent to "**Z**".

See Also: Commands: C

Example: Q1 EJ — Jumps to the position in the edit buffer determined by numeric register 1.

EK*file*<ESC> Kill (Delete) File

Example: EKfile.txt EK *.bak -EK file.bak

Description: This command erases (kills) the file '*file*' from the disk. Wildcard characters may be used to erase more than one file. The command first displays a directory of the files to be erased and asks for confirmation before erasing them. Alternatively, the command form "**-EK**" skips the directory display and confirmation prompt. Using "**EK**" is the easiest way of recovering from a disk write error by making more disk space free.

Notes: Never erase any ".$$$" or ".R" files from within VEDIT PLUS! These are the temporary files VEDIT PLUS is using. Do not delete the input file until all of it has been read into memory.

See Also: Commands: EC, ED
Disk Write Error Recovery

Example: EK oldfile.txt — Erases the file "OLDFILE.TXT" from the disk.

EK *.bak — Erases all files with a filename extension of ".BAK" from the current drive.

EL*file*[*line range*] Look at External File

Example: EL file.txt EL b:file.txt[200,400]

Description: This command allows you to view a range of lines from another file. If no line range is specified, the entire file is displayed. Line numbers are displayed ahead of each line. The displayed line numbers may subsequently be used with the "**EG**" command to extract a portion of the file.

Notes: Press **<CTRL-C>** to stop the file display.

See Also: Commands: EG

Example: EL library.asm Displays the file "LIBRARY.ASM" with line numbers.

nEM*string*<ESC> Match String

Example: EMalign<ESC> @EM/¦<begin/ #@EM/¦A/

Description: This command compares (matches) '*string*' to the text at the edit pointer. '*string*' may contain pattern matching codes. If the entire '*string*' matches the comparison is successful - internal value "**.rv**" is set to "0", "**.n**" is set to the number of matching characters and the edit pointer is moved past the matching characters. If the match is not successful "**.er**" is set to "1", "**.n**" is not changed and "**.rv**" is set to "1" if '*string*' is lexically "less than" the text or "2" if '*string*' is "greater than" or "3" if '*string*' contains pattern matching codes. The edit pointer is not moved if the match is unsuccessful.

Notes: "**EM**" is different from "**F**" in that "**F**" will search for the string, while "**EM**" does not search if it does not completely match the text right at the edit pointer. Also, no error is given if "**EM**" is unsuccessful. Use the command form "**EM¦Rr**" when the '*string*' is contained in text register 'r'.

See Also: Commands: F, RM

Example: EM¦<begin$ Checks if the edit pointer is at the beginning of a line beginning with the word "begin". Sets the internal values accordingly.

nEN Need Memory

Example: 10000EN 0EN EN

Description: This command makes 'n' bytes of memory free in the edit buffer, if possible, by buffering some of the file back to disk. If 'n' is "0" it tries to make free the amount specified as "Size of Memory after Auto-Read" during Installation (Task 9.4). If 'n' is not specified it buffers the file out to disk until only one "page" (Task 9.5) remains in memory - this is typically 6 - 12K.

"**EN**" does not buffer out any text which is within 2000 bytes of the edit pointer. If the desired amount cannot be written to disk with just forward disk buffering it will also use backward disk buffering. If 'n' is too large, it will make as much memory free as possible. Use the "**U**" command to confirm how much is free.

Notes: This command is typically used to make a large block of memory free before inserting a large text register or inserting a large file segment with the "**EG**" command.

See Also: Commands: U, EE

Example:

```
10000EN
RG5
```

Make 100000 bytes free in the edit buffer in preparation for inserting the contents of text register 5.

EP n k Edit Parameters

Example: EP 1 2 EP 7 70

Description: This command changes the value of parameter 'n' to 'k'. The command "**EP**" with no arguments displays the current value of all parameters. The command form "**EP n**" without the second argument only stores the value of parameter 'n' in internal value ".**rv**". The default values for these parameters are determined during Installation. The parameters are:

1	Cursor type	(0 - 4)
2	Cursor blink rate	(5 - 255)
3	Indent Increment	(1 - 20)
4	Lower/upper case conversion	(0, 1, 2 or 3)
5	Conditional conversion character	(32 - 126)
6	Delay for Command Mode scrolling	(0 - 255)
7	Right margin for Word Wrap	(0 - 255)
8	High-Bit (8th bit) processing	(0 - 7)
9	Cursor positioning mode	(0 - 2)
10	Horizontal scroll margin	(40 - 255)
11	Horizontal scroll increment	(1 - 78)

Parameter (1) determines the type of cursor displayed in Visual Mode for memory mapped versions. The CRT terminal versions use the terminal's cursor instead. The cursor types are: 0=Underline, 1=Blinking Reverse Video Block, 2=Solid Reverse Video Block, 3=Attribute, 4=System cursor (IBM PC only). Type 1 is generally best on the IBM PC unless you are using Borland's Turbo Lightning for which you need type 4.

Parameter (2) determines the cursor's blink rate for cursor types 0 and 1 above.

Parameter (3) determines how much further the editor indents each time **[INDENT]** is pressed. The indent position after pressing **[INDENT]** four times is therefore the "Indent Increment" multiplied by four.

Parameter (4) determines whether lower case characters are converted to upper case. For value (0), no conversion takes place; for (1), all lower case are converted to upper case. For (2), lower case letters are converted to upper case, unless the cursor is past a "special" character on the text line. This "special" character is set by parameter (5). Mode (3) is similar to (2) except that characters are reversed instead of being forced to upper case. All of this is primarily applicable to assembly language programming, where it is desirable to have the Label, Opcode and Operand in upper case and the comment in upper and lower case.

Parameter (5) sets the conditional upper/lower case convert character used for parameter (4) above.

Parameter (6) determines how much delay is added to each line displayed in Command Mode. Without any delay, memory mapped systems, such as the IBM PC, will display the "**ED**", "**T**" and other commands too quickly to read.

Parameter (7) is the right margin for Word Wrap. It is also the right margin used for formatting paragraphs. A value of 0 disables both Word Wrap and formatting. It should be disabled when editing programs!

Parameter (8) governs the processing of high-bit characters and is a 3 bit parameter. Setting Bit 1 allows high-bit input characters. Bit 2 allows high-bit characters on output (otherwise, the 8th bit is stripped and the character displayed in reverse video if possible). Bit 3 allows unused function/control keys to be inserted into the edit buffer. Users with an IBM PC, NEC APC or other machines with graphics characters will want to use a value of "3". CRT terminals work best with a value of "1" - allow 8 bits on input, display high-bit characters in reverse video.

Parameter (9) determines the cursor positioning mode. The modes are 0 - cursor only at real text; 1 - cursor allowed past end of lines; 2 - pad with spaces when past end of line.

Parameter (10) is the horizontal scroll margin which sets the maximum right margin for scrolling. Text lines longer than this "scroll margin" are wrapped to the next screen line.

Parameter (11) is the horizontal scroll increment. It determines how much the screen scrolls right or left when **[SCROLL RIGHT]** and **[SCROLL LEFT]** are pressed or VEDIT PLUS scrolls automatically.

Notes: The numbers are specified in decimal and are separated by spaces or commas. The last number should be followed by an **<ESC>** or a **RETURN** to prevent ambiguity.

Each edit buffer has its own set of "**EP**", "**ES**" and "**ET**" values; when an edit buffer is created, they are initially set to their installed values.

See Also: Commands: ES, H (to help you remember them all)
Installation

Examples:

EP 3 6	Sets the "Indent Increment" to six.
EP 7 70	Sets the Word Wrap margin to 70.

EQ Quit (Abandon) Editing

Example: EQ EQY EQA

Description: This command quits (abandons) the edit session without saving any edit changes. It also leaves the current edit buffer and, if this is the last edit buffer, it exits VEDIT PLUS. Except for the main edit buffer, the current edit buffer is converted back to an empty text register. The user is prompted for confirmation to abandon the edit session. The command "**EQY**" skips the confirmation. The command "**EQA**" quits all edit buffers (abandons all files) and exits VEDIT PLUS.

"**EQ**" is often used after examining a file which you don't want to change. In this case it is safer to quit, rather than exit with "**EX**", in case you accidentally did change something.

Notes: You can also quit with the **[FILE]-Q**uit function.

Any existing backup (".BAK") file with the same filename as the output file will have been deleted if any characters were written to the (now abandoned) output file. Any files deleted with the "**EK**" command will remain deleted. With these exceptions, the files will exist on disk just as they did before you started the edit session.

If you quit with an "**EQ**" sometime after an "**EA**" command, you will only abandon those changes made after the "**EA**" command. Those changes made before the "**EA**" command will have already been saved on disk.

See Also: Commands: EA, EZ

Example:

VPLUS oldfile.txt	You only want to examine the file without changing anything.
EQ	When done you quit to leave the file unchanged and leave VEDIT PLUS.

ER*file*<ESC> Edit Read (Open Input File)

Example: ER newfile.txt ER

Description: This command opens the file '*file*' for input (reading). However, nothing is actually read into the edit buffer. The "A" command or auto-buffering is used to actually read the input file. If the same file was already open for input, the file is "rewound", so that the file can again be read from the beginning. The error "FILE NOT FOUND" is given if '*file*' does not exist.

The command "ER" without a filename displays the name of the current input file if it is still open. The command ":ER" also displays the filename, but suppresses the following <CR><LF>. The command "+ER" displays the filename and includes its drive and/or path.

Files can also be read from disks which are not currently logged in by using the "EC" command. Issue the "EC" command, insert the new disk into a drive which is not being used for any output files and open a file for reading with the "ER" command. This may be necessary in case a file has been split into two parts during a disk write error recovery.

Notes: Filenames may be preceded with spaces to improve readability.

As soon as VEDIT PLUS reads the entire input file it closes the input file. This allows the file to be accessed by other users on a multi-user or network system. However, since the input file is no longer open, the "ER" command (without a filename) displays nothing - it only displays the filename when an input file is open and has not already been completely read.

See Also: Commands: A, EC, EB, EW

Example: ER parts.inv
20A The file "PARTS.INV" is opened for input and twenty lines from it are appended to the end of the edit buffer.

ES n k Edit Switches

Example: ES 1 0 ES 3 1

Description: This command changes the value of switch 'n' to 'k'. The command **"ES"** with no arguments displays the current value of all switches. The command form **"ES n"** without the second argument only stores the value of switch 'n' in internal value **".rv"**. The default values for these switches are determined during Installation. The switches are:

1	Expand Tab with spaces	(0=NO 1=YES)
2	Auto-buffering in Visual Mode	(0=NO 1=YES 2=BACK)
3	Auto-Indent Mode	(0=NO 1=YES)
4	Point past text register insert	(0=NO 1=YES)
5	Equate Upper/Lower case in search	(0=NO 1=YES)
6	MS-DOS End-of-file padding	(0=NO 1=YES)
7	Reverse all upper and Lower case keys	(0=NO 1=YES)
8	Suppress error handling	(0=NO 1=YES)
9	Use explicit text delimiters	(0=NO 1=YES)
10	Global file operations	(0=NO 1=YES)
11	Justify paragraphs	(0=NO 1=YES 2=UNJUSTIFY)

Switch (1) determines whether the **[TAB CHARACTER]** function is expanded with spaces to the next tab position. If not, a tab character is inserted into the edit buffer. Except for special applications, **[TAB CHARACTER]** should not normally be expanded with spaces.

Switch (2) determines whether auto-buffering is enabled in Visual Mode. "0" disables auto-buffering, "1" enables only forward disk buffering, and "2" enables both forward and backward disk buffering. A value of "2" is recommended for use with hard disks and a value of "1" for use with floppy disks. Use "0" when you are giving explicit Read/Write commands. This prevents unexpected disk read and write from occurring while editing in Visual Mode.

Switch (3) enables/disables "Auto-Indent" mode. When enabled, the indent position for a new line of text is initially the same as for the previous line of text. This is convenient for programming in 'C', Pascal, PL/I, etc. The indent position can always be changed with **[INDENT]** and **[UNDENT]**.

Switch (4) determines the edit pointer's position (or cursor's in Visual Mode) following insertion of a text register. If the switch is "0", the edit pointer is not moved, and is left at the beginning of the newly inserted text. If the switch is "1", the edit pointer is moved just past the newly inserted text.

Switch (5) determines whether upper and lower case letters are equated when searching using the "F", "S" and "EM" commands and **[FIND]** and **[REPLACE]** functions. Typically they are equated so that the string "why" will match "Why", "WHY" and "why".

Switch (6) determines whether MS-DOS files are written in their exact file length or are written with a **<CTRL-Z>** at the end and are padded to make the file length a multiple of 128. A value of "0" for exact length files works best for most applications.

Switch (7) determines whether all letters typed on the keyboard will be reversed with respect to upper and lower case. It should normally be OFF, but does allow a user with an upper case only keyboard to enter lower case letters. Setting the switch to "1" reverses all keyboard letters in both Command and Visual Mode.

Switch (8) determines whether the "suppress error handling" command modifier ":" is set by default for all applicable commands. If not suppressed, a search or "L" command error causes an error message and the command to be aborted. Search errors are usually only suppressed for command macros.

Switch (9) determines whether the "explicit text delimiter" command modifier "@" is set by default for all applicable commands. This is a matter of personal preference, but is useful with command macros.

Switch (10) determines whether the "global" command modifier "_" is set by default for all applicable commands. We suggest only enabling this switch in command macros. Otherwise you may find unnecessary file buffering occurring. Enabling this switch also sets the "global" modifier for the **[FIND]** and **[REPLACE]** functions.

Switch (11) determines whether the **[FORMAT PARAGRAPH]** function and **"YF"** command will also justify the formatted paragraph. Switch value "0" disables justification. Switch value "1" enables justification. Switch value "2" will "unjustify" the paragraph, removing extra spaces.

Notes: The numbers are specified in decimal and are separated by spaces or commas. The last number should be followed by an **<ESC>** or a **RETURN** to prevent ambiguity.

See Also: Customization, Visual Mode

Example: ES 1 1 Enables the **<TAB>** key in Visual Mode to be expanded with spaces.

ET Edit Tab

Example: ET 20 40 60 80 100 120 ET 8 ET

Description: This command changes the tab positions used for displaying tab characters and, when the **"ES 1 1"** ("Expand Tab") switch is set, for expanding the **[TAB CHARACTER]** key. Up to 33 tab positions are allowed and they must be in the range 1 - 254. The default positions are set during Installation. If only one number 'n' is given, the tab positions will be set to every 'n' columns. The command **"ET"** with no arguments displays the tab positions.

Counting starts at 1 (not at zero). Therefore the normal tab positions at every 8 columns are:

9 17 25 33 41 49 57 65 73 81 89 97 105 113 121 ...

Notes: For word processing the tabs can be set to the same positions as are specified for the print formatter program (V-PRINT) in order to preview how they will look when printed

Each edit buffer has its own tab positions. **"ET"** changes the tab positions for the current edit buffer (but not other existing edit buffers) and sets the initial values for subsequently created edit buffers. The command **"-ET"** changes the tab positions only for the current edit buffer.

If you set the tab positions to anything other than every 8, you may find that other programs will not display your text properly because many programs have fixed tabs at every 8 columns.

If you send files containing tab characters to mainframe computers, you may find that the tabs are lost in the transfer. (Many mainframes do not have tab characters internally.) These two cases are good candidates for expanding the **[TAB CHARACTER]** key with spaces to the next tab position.

See Also: Installation, Visual Mode, Indent and Undent Functions

Example:

EU *drive:/pathname* Drive/Path Used

Example: EU B: EU C:\LETTERS

Description: This command changes the "current" (logged in) drive and/or MS-DOS subdirectory (or CP/M user number) to the specified one. This allows files to be accessed without having to specify the drive and/or pathname each time. The command "**EU**" without any arguments displays the current drive and subdirectory.

Notes:

See Also: MS-DOS Pathnames, Changing Current Drive / Directory

Example: EU B: Changes to drive B.

EU C:\LETTERS Changes to subdirectory "LETTERS" on drive C.

EV Editor Version

Example: EV

Description: This command displays the VEDIT PLUS version number. This number should be used in any correspondence you have with us concerning this product. This command can also be used inside iteration loops to give some indication of the progress being made in long executing macros.

Notes:

See Also:

Example:

EW*file*<ESC> Edit Write (Open Output File)

Example: EW newdat.inv EW

Description: This command opens the file '*file*' for output and subsequent writing. No text is actually written by this command. An output file must be opened in order to save any text on disk. A file can also be opened with the **"EB"** and **"EA"** commands, with the **[FILE]-New** function and when VEDIT PLUS is first invoked. If a file is already open for output, the error "CANNOT OPEN TWO" is given and the command cancelled.

The command **"EW"** without a filename displays the name of the current output file if it is open. The command **":EW"** also displays the filename, but suppresses the following **<CR><LF>**. The command **"+EW"** displays the filename and includes its drive and/or path.

Note: The file opened is actually a temporary file with the same filename, but with an extension of ".$$$". The file is not made permanent and given its true name until it is "closed" with the **"EA"**, **"EF"**, **"EX"** or **"EY"** commands or by the **[FILE]-Exit** function. At that time, any existing file on disk with the same name as the output file is backed up by renaming it with an extension of ".BAK". Any existing backup file with the same name is deleted when the first text is written to the output file.

See Also: Commands: W, EA, EF, EX, EY

Example:

```
EW part1.txt
24W
EF
EW part2.txt
EX
```

The first 24 lines of the edit buffer are written out to file "PART1.TXT" and the rest of the edit buffer is written out to file "PART2.TXT" and edit session is completed.

```
ER a:bigfile.asm
EW b:bigfile.asm
OA V
```

Typical procedure for editing a file which is too big for both it and its backup to fit on the same disk. In this case, it is read from drive A: and written to drive B:. Just be sure that disk B: is nearly empty.

EX Save And Exit

Example: EX

Description: This is the normal way to save the file being edited on disk and exit the edit buffer. It saves the entire file being edited (including any un-read portion of the input file) on disk. It also exits the current edit buffer and, if this is the last edit buffer, it exits VEDIT PLUS. Except for the main edit buffer, other edit buffers are converted back to empty text registers. All file backup and renaming is done as with the "**EF**" command. The error "NO OUTPUT FILE" results if no output file is open. The error "NO DISK SPACE" results if there is insufficient disk space to save the entire file.

Notes: "**EX**" performs the same operation as **[FILE]**-Exit. In case of a "NO DISK SPACE" or "NO DIR SPACE" error, see the heading "Disk Write Error Recovery" in the User Guide for the procedure to save your file.

See Also: Commands: EA, EB, EF, EQ, EW, EY
[FILE] function

Example:
```
VPLUS FILE.TXT
V
EX
```
The editor is invoked in the normal way to edit a file in Visual Mode. The new file is then saved on disk.

EY Save And Remain

Example: EY

Description: This command saves the entire file being edited on disk, in preparation for editing another file. Like the "**EX**" command it saves the entire file being edited (including any un-read portion of the input file) on disk. However, it stays in the same edit buffer. It is usually followed by an "**EB**" command to edit another file. The error "NO OUTPUT FILE" is displayed if no output file is open.

Notes: In case of a "NO DISK SPACE" or "NO DIR SPACE" error, see the heading "Disk Write Error Recovery" in the User Guide for the procedure to save your file.

The **[FILE]-New** function is a combination of the "**EY**" and "**EB**" commands.

See Also: Commands: EX, EF

Example:

```
EY
EB newfile.txt      The current file is saved on disk,
                    and the file "NEWFILE.TXT" opened
                    for editing.
```

EZ Abandon Edit Session

Example: EZ EZY

Description: This command quits (abandons) the edit session, like the "**EQ**" command, without saving any edit changes. However, it stays in the same edit buffer and is often followed by an "**EB**" command to edit a different file. The user is prompted for confirmation to abandon the edit session. The command "**EZY**" skips the confirmation.

"**EZ**" is often used right after you invoke VEDIT PLUS and you realize that you loaded the wrong file. "**EB**" is then used to edit the desired file.

Notes: The notes for the "**EQ**" command apply.

See Also: Commands: EQ, EY

Example:

#K	Shoot!! Meant -#K
EZ	Since a bad mistake was made in the above command, it is best to quit this edit session and start over. All edit changes are lost, but not your original file!

cJL/cJM/cJN/cJO Jump Commands

cJP*label*<ESC>

Example: .erJM Q1 JO JPloop$

Description: These commands perform a conditional jump within command macros. If the conditional expression 'c' evaluates to "0" the jump is not taken; if it evaluates to "1" (or any other value) the jump is taken. If any command is not preceded by an expression it defaults to "1" and, therefore, unconditionally performs the jump.

"**JL**" jumps out of the current REPEAT-UNTIL (iteration) loop and execution continues with any commands following the "]" at the end of the loop.

"**JM**" exits (jumps out of) the currently executing macro. Execution continues with a higher level macro, if there is one, otherwise it returns to the "COMMAND:" prompt.

"**JN**" starts the next iteration of current REPEAT-UNTIL loop. If the iteration count is exhausted the loop ends.

"**JO**" aborts the command macro execution and returns immediately to the "COMMAND:" prompt.

"**JP***label***<ESC>**" jumps to '**!***label***!**', which must appear somewhere in the current macro. You can jump out of a flow control structure, but you cannot jump into one. The command must end with an **<ESC>** or, alternatively, explicit delimiters may be used.

Notes: "**!***label***!**" can be used as a label for the "**JP**" command or as a comment. Unused labels serve no programming purpose and are therefore a convenient way of placing comments within a macro.

See Also: Commands: EJ
Flow Control, Jump (Branching) Commands, Commenting Macros

Example: See the heading "Flow Control" in the Programming Guide for examples.

OC*command* DOS Command

Example: OC DIR @OC/V-PRINT CHAPTER1/

Description: This command executes a single MS-DOS command and returns to VEDIT PLUS. The command may run another program such as a compiler, V-PRINT or V-SPELL.

'*Command*' must be followed by **<ESC>** or **RETURN**, or explicit delimiters may be used. '*Command*' may contain a "¦**Rr**" to use the contents of register 'r' as all or part of the DOS command.

As with all commands which can take explicit delimiters, if you have set "**ES 9 1**", then you MUST use explicit delimiters with the "**OC**" command.

Although the "**ED**" command is easier to use, you can display the directory with the command "**OC DIR**" which also displays the size of each file.

Notes: MS-DOS has to be able to find its COMMAND.COM file on the same drive as when the computer was first booted or it will give the error "Insert COMMAND.COM disk in drive A:". This generally only occurs if you boot from a floppy disk and then remove the system disk.

If you run another program from within VEDIT PLUS it may not have enough memory to run properly due to the memory used by VEDIT PLUS. VEDIT PLUS will "grab" 128K of memory, if available, and another 64K for each additional edit buffer. You can free memory for use by other programs by exiting all unneeded edit buffers which converts them back to text registers and frees their 64K of memory. Alternatively, you can invoke VEDIT PLUS with the "-S" option (see ERRATA.DOC file), in which case it only "grabs" 64K for all edit buffers.

See Also: Commands: OS
Memory Management

Example: OC VPRINT CHAPTER1
Runs the program V-PRINT to format and print the file CHAPTER1.VPR from within VEDIT PLUS.

OC VSPELL ¦R9 Runs the program V-SPELL on the file whose name is in register "9".

OS DOS Operating System

Example: OS

Description: This command enters the MS-DOS operating system without leaving VEDIT PLUS. The normal DOS prompt, i.e. "A>" or "C>" will be displayed. Any number of DOS commands and programs can be executed. Give the DOS command "**EXIT**" to return to VEDIT PLUS.

Notes: All notes for the "**OC**" command apply here too.

See Also: Commands: OC

Example:

PE Page Eject

Example: PE

Description: This commands advances the printer to the start of a new page. In printer terminology this is often called a "Form Feed". It is used in print macros to force the following text to start on a new page. It can be used from the "COMMAND:" prompt to start a new page or to send a blank page through the printer. The command "**-PE**" resets the internal line counter without ejecting a page on the printer. This is useful inside macros which change the "**PP**" parameters.

Notes: "**PE**" is equivalent to the **[PAGE]**-Eject function.

"**PE**" will start a new page by sending either multiple "Line-Feed" characters to the printer or by sending a single "Form-Feed" character. This is determined by the setting of the "**PP 4**" print parameter. Form-feeds are preferable and should normally be used except for those rare printers which do not support the Form-Feed character.

See Also: Commands: PP, PR
[PRINT] function

Example: B [40PR 40L PE] This macro prints the entire edit buffer, but with only 40 lines of text per page.

PP n k<ESC> Print Parameters

Example: PP 2 50 PP 4 1

Description: This command changes the value of print parameter 'n' to 'k'. The command "PP" with no arguments displays the current value of all switches. The command form "**PP n**" without the second argument only stores the value of parameter 'n' in internal value "**.rv**". The default values of these parameters are determined during Installation. The parameters are:

1	Physical lines per page	(5 - 100)
2	Printed lines per page	(1 - 100)
3	Left margin for printing	(0 - 100)
4	Use Form-Feed for new page	(0 = NO, 1 = YES)

Parameter (1) must be set to the length of a page in lines. Typical paper is 11 inches long and is printed 6 lines per inch, giving a value of 66. Set to the length of your page or, when printing labels, to the number of lines between labels.

Parameter (2) sets the number of lines printed per page before a new page is automatically started. The lines will be centered top to bottom. For example, with 66 physical lines per page and 60 printed lines, there will be a 3 line (1/2 inch) margin at the top and bottom of each page. Picking a smaller number gives larger top and bottom margins. Setting this value to the same as parameter (1) allows pages to be printed without top/bottom margins.

Parameter (3) determines by how many columns printed text is offset from the very left edge of the paper - i.e the size of the left margin. The actual size of the left margin also depends upon the paper's alignment in the printer and on the number of columns printed per inch (10, 12 or 15). The normal value of 12 gives a margin of roughly one inch. When printing computer programs you may want to set this parameter to "0" - no left margin.

Parameter (4) determines whether new pages are started by sending out the correct number of Line-feeds (blank lines) or a single Form-feed character. Most printers respond properly to a Form-feed character and this option should then be used. (If your printer automatically wraps long lines to the next line, new pages won't start at the right place unless you have enabled Form-feeds).

Notes: See Notes for "EP" command.

See Also: Commands: PE, PR
[PRINT] function, Installation Task 4.

Example: PP 2 50 Set to print only 50 lines per page.

mPR Print Text

Example: 40PR -20PR 0PR 60_PR

Description: This command prints the specified lines. The line range and syntax are identical to the "**T**" command. If there are fewer than 'm' lines to print, as many as possible are printed and no error is given.

The command form "**m_PR**" performs auto-buffering, if necessary, to print the specified lines. The command form "**p,qPR**" prints all characters from the 'p'th character in the edit buffer up to, but not including, the 'q'th character.

Notes: The printing can be stopped by pressing **<CTRL-C>**.

Each page is printed with the number of lines specified by the "**PP 2**" parameter. After a "full" page is printed, a new page is automatically started.

Since the "**PR**" command does not move the edit pointer, it is often followed inside macros by an "**L**" command to advance the edit pointer to the next block of text to print. In case of auto-buffering, the edit pointer is restored to its original position following the printing.

Tab characters are printed by sending the correct number of spaces to the printer. All other control characters are sent verbatim - without expansion. The command "**YP mT**" will also print the text, but will expand control characters and print **<ESC>** as "$".

See Also: Commands: L, T, RP, YP
[PRINT] function

Example: _B #_PR Moves the edit pointer to the beginning of the file and prints the entire file.

R* Command Macro Comment

Example: R* This is a comment in a command macro

Description: This command allows a comment to be placed within command macros. All text through the next <CR><LF> is ignored.

Notes: The "R*" and following comment can appear anywhere in a command macro except in the middle of a text string or filename.

See Also: Commenting Macros, Labels

Example:

RAr Auto-execute Register

Example: +RAY RA0

Description: This command causes the macro in text register 'r' to be executed in place of the normal "COMMAND:" prompt. It is primarily used to replace Command Mode with a main menu of operations, as in the "MENU.VDM" macro. 'r' may be any register except "0". Due to the potential danger of this command (it can cause infinite loops), its syntax is checked carefully - leaving off the "+", specifying register "0" or no register at all will disable auto-execution.

Notes: Use this command with care! Since pressing <CTRL-C> normally returns to the "COMMAND:" prompt, it re-executes the register instead. When writing a main menu macro debug it fully before adding the "RA" command. Be sure to allow a way for the menu user to exit VEDIT PLUS (it can be done with the [FILE] function).

See Also: Commands: M, RE, RJ
Auto-Execution of Macros
Examine the "MENU.VDM" and "MENU.INI" files.

Example: +RAY Sets up to execute register "Y" in place of the normal command prompt.

mRCr Copy To Register

Example: 40RC1 -20RC+2 Q1,Q2 RCZ

Description: This command copies the specified lines 'm' to text register 'r'. The previous contents of the text register are destroyed, unless 'r' is preceded with a "+" indicating that the text is to be appended. The range of lines copied is the same as for the "K" or "T" commands. The text in the edit buffer is unchanged. If there is insufficient memory space for the text copy, the text register is only emptied, nothing is copied to it and the "*BREAK*" error is given.

The command form "p,qRCr" copies all characters from the 'p'th character in the edit buffer up to, but not including, the 'q'th character to text register 'r'. Counting starts with 0.

Notes: The error "MACRO ERROR IN r" results if a macro attempts to change a text register which contains an executing command macro. The error "INVALID EDIT BUFFER OPERATION" results if 'r' is an edit buffer.

On systems with less than 192K of memory, the text registers will share memory space with the edit buffers - therefore, saving text in the text registers decreases the amount of memory available to the edit buffer. Thus the "REr" command should be given to empty a register when it is no longer needed.

See Also: Commands: K, T, RG
[BLOCK] function

Examples:

120RC1 120K	Saves 120 lines in text register 1 and then deletes them from the edit buffer.
-23T -23RC6	Displays text lines for verification before saving them in the text register.

RDr Register Dump

Example: RD3 +RD9

Description: This command displays (dumps) the contents of text register 'r' on the console. Control and tab characters are not expanded. The command is useful for sending initialization sequences to a CRT terminal, such as sequences to set up programmable function keys. The **"RT"** command should be used to display the registers, since control characters are then expanded.

VEDIT PLUS's window handler sends a **<CR><LF>** to a CRT terminal if the **"RD"** command attempts to exceed the right window (or screen) margin. If this interferes with terminal initialization, use the command form **"+RDr"** which allows the dumped text to exceed the window margin.

Notes: Press **<CTRL-C>** to stop the **"RD"** command.

See Also: Commands: RT
Auto-Startup

Example: RD5 The contents of text register 5 are dumped (displayed) on the console.

REr Empty register

Example: RE4 +REM

Description: This command empties text register 'r'. An edit buffer cannot be emptied and leads to the error "INVALID EDIT BUFFER OPERATION". Normally 'r' also cannot be emptied if it contains a currently executing command macro unless the command form **"+REr"** is used. **"+REr"** must be used with care, but is useful for saving the memory space of a command macro which VEDIT PLUS thinks is still executing, but is no longer really needed.

Notes: It is a good habit to empty unused text registers.

See Also: Commands: RC
Command Macros

Example: RE4 Empty text register 4.

RGr Get (Insert) Register

Example: RG4

Description: This command inserts a copy of text register 'r' at the position of the edit pointer. If the text register is empty, nothing is inserted. Depending upon the setting of the "**ES 4**" switch, the edit pointer will either remain unchanged or move just past the inserted text. The contents of the text register are not affected. If there is insufficient memory space for the entire copy, nothing is inserted and the "*BREAK*" error is given.

Notes: 'r' can be an edit buffer, in which case the current contents of the edit buffer are inserted.

See Also: Commands: RC
[BLOCK] function

Examples:

B RG9	Inserts the contents of text register 9 at the beginning of the edit buffer.
12[RG2]	Inserts the contents of text register 2 twelve times at the edit pointer position.
132RC3 132K _B 10L RG3	Moves 132 lines of text, by saving it in text register 3, killing the original lines and inserting the text after the tenth line of the file.

RIr*text*<ESC> Insert Into Register

Example: @RI3/B #PR/

Description: This command places '*text*' into text register 'r'. If 'r' is preceded by a "+" the '*text*' is appended to any existing contents in the register. The '*text*' may contain the **RETURN** key, which is expanded to **<CR> <LF>**. If insufficient memory space exists, the error "*BREAK*" is given and only part of the '*text*' will be inserted. This command is useful for setting up a text register from within a command macro.

Notes: If a register needs to be set up from the keyboard, it is often easier to change the register into an edit buffer and enter the text directly from Visual Mode.

See Also: Programming Guide: "Loading Macros into Text Registers"

Example: @RI3/B #PR/ The command sequence "B #PR" is placed into text register 3.

RJr Jump To Macro

Example: RJY

Description: This command "jumps" to execute text register 'r' as a command macro. It differs from the "**M**" command in two important ways. First, there should be no commands following the "**RJ**" because command execution does not return to the text register issuing the "**RJ**". Second, since the text register which issued the "**RJ**" is no longer "executing", it can be modified without getting the "MACRO ERROR IN r" error.

Notes: The "**M**" command is analogous to a subroutine "CALL" with execution returning to the "calling" program when the subroutine is done. (VEDIT PLUS maintains an internal stack of these "return addresses".) In contrast, the "**RJ**" is a "JUMP" since execution doesn't return to the "program" which issued the "**RJ**" command. (Since a return address for the issuing macro is not on the internal stack, it is no longer considered to be "executing".) In programming terminology "**RJ**" is similar to "chaining" in BASIC.

See Also: Commands: M, RE
Jumping to a Command Macro

Example: RJY "Jump" to execute text register "Y" as a command macro.

RLr *file*<ESC> Load Register

Example: RL4 macro1.vdm

Description: This command loads the entire file '*file*' into text register 'r'. The file itself is not affected. If there is insufficient memory space to load the entire file, as much as possible is loaded and the error "*BREAK*" is given. This command is often used to load command macros which have been saved on disk. The command form "**+RL**" performs an extended search for the file.

Notes: Do not use this command to edit another file. Instead use the "**EE**" and "**EB**" commands to simultaneously edit another file in an edit buffer/text register.

See Also: Commands: RS, EG

Example: RL4 macro.vdm Loads the file "MACRO.VDM" into text register 4.

RMr Match Register

Example: RM8 _RMC

Description: This command compares (matches) the contents of text register (or edit buffer) 'r' to the text at the edit pointer. It performs a direct character comparison without pattern matching; only the "Equate Upper/Lower case" switch is respected. If 'r' is a text register the comparison is with the entire text register; if 'r' is an edit buffer the comparison starts with 'r's edit pointer and its edit pointer is moved past the last character matched.

In either case, if the rest of the text register matches, the internal value "**.rv**" is set to "0", otherwise "**.rv**" is set to "1" if 'r' is lexically "less than" the text or "2" if 'r' is "greater than" the text. Regardless of whether the entire text register matches, the edit pointer is moved past as many characters which do match and "**.n**" is set to the number of characters which matched. The error flag "**.er**" is not affected.

Notes: Two major differences between "**EM**" and "**RM**" are that "**EM**" uses pattern matching and does not move the edit pointer unless the entire match is successful. "**RM**" matches as much as possible; it is not an error if the entire text register doesn't match. "**RM**" is useful for moving the edit pointer in two edit buffers past all characters which match, regardless of whether the two edit buffers match completely.

See Also: Commands: EM

Example: _RMY Matches the text at the edit pointer with the text at the edit pointer in edit buffer "Y" and moves both edit pointer past all characters that match. Auto-buffering is performed, if necessary, in the current edit buffer.

RPr Print Register

Example: RP3

Description: This command prints the contents of text register 'r'. It allows a hardcopy of the text or command macro in the text register to be made. Also allows a disk file to be printed after first loading it into a text register.

Notes: Be sure your printer is "On Line". Press **<CTRL-C>** to stop the printing.

Tab characters are printed by sending the correct number of spaces to the printer. All other control characters are sent verbatim - without expansion. The command "**YP RTr**" will also print the text, but will expand control characters and print **<ESC>** as "$".

See Also: Commands: RD, RT

Example: RP5 The contents of text register 5 are printed.

RQr*text*<ESC> Register Query

Example: @RQF/Enter File Name:/

Description: This command queries (prompts) on the screen with '*text*' and then reads the user's response line into text register 'r'. The user ends the input line by pressing **RETURN** or two **<ESC>**'s which are also stored in the text register. The command form ":**RQr***text***<ESC>**" does not store the ending **RETURN** or two **<ESC>**'s in the text register. The 'r' may be preceded with "+" to append the response line to the existing contents of the text register. The command form "**+RQr***text***<ESC>**" prompts on the status line - '*text*' must then be a single line prompt.

Notes: The query always starts on a new screen line. The response line may be edited in the same way as a command line may be edited. In programming terminology "**RQ**" performs a "string input", similar to an "INPUT" statement in BASIC, or a combination of "puts(*text*)" and "gets(r)" in C.

See Also: Commands: XK, XQ, YT

Example: @RQF/Enter File Name:/
EB¦RF Prompts for a filename and saves the filename in register "F". Then opens the file which the user specified for editing.

RSr *file*<ESC> Save Register

Example: RS4 macro1.vdm

Description: This command saves the contents of text register 'r' in the created file '*file*'. The register contents are not affected. This command is commonly used to save a section of text in its own disk file, or to save a command macro for later use. If there is insufficient disk space for the entire file, as much as possible is saved and the error "NO DISK SPACE" is given. The error "NO DIR SPACE" is given if there is insufficient directory space on the disk.

Notes: If an existing '*file*' already exists, the user is prompted for confirmation to overwrite it. If there is insufficient disk space to save the register, try deleting some files or insert another disk and give the command again.

See Also: Commands: RL

Example: RS4 macro.vdm Saves the contents of text register 4 in the file "MACRO.VDM".

RTr Type (Display) Register

Example: RT3

Description: This command displays (types) the contents of text register 'r'. This is commonly used to remind oneself what is in a particular register. Press **<CTRL-S>** to pause the display or **<CTRL-C>** to abort the command. Any embedded **<CTRL-S>**'s will also pause the display.

Notes: Control characters are expanded and **<ESC>** is represented as a "$". Use the "**RD**" command to dump out a text register without expanding control characters.

See Also: Commands: RD

Example: RT5 The contents of text register 5 are displayed on the console.

RU Register Usage

Example: RU

Description: This command displays the number of characters held in each text register/edit buffer. A "*" is displayed next to those text registers which are also edit buffers. It is commonly used to see which registers are being used, how many characters they hold and which registers are currently edit buffers.

Notes: The third number displayed by the "U" command is the total number of bytes held by all text registers which ARE NOT also edit buffers. If this value is not zero the status line TEXT message is displayed during Visual Mode.

See Also: Commands: U

Example:

RU	Displays the sizes of the text registers.
40RC3 RU	Saves 40 lines of text in register 3 and then displays how many bytes are now in the text registers.

RXr *file*<ESC> Register Execute

Example: RXP print.vdm

Description: Loads *file* into text register 'r' and executes it immediately as a macro. The command is a short hand for the equivalent commands "**+RLr** *file*" and "**Mr**".

Notes: The "X" in "**RX**" was purposely chosen to coincide with the "**-X**" invocation switch (think of the word eXecute). The "**RX**" command allows a macro to be loaded and executed from within VEDIT PLUS, and also lets you choose which register it will be loaded into.

See Also: Commands: M, RL
Auto-execution

Example:

RLP PRINT.VDM	Loads the print formatter macro into register 'P' and executes it.

mXAr Add to Numeric Register

Example: XA2 12XA22

Description: This command adds the positive or negative value 'm' to numeric register 'r'. (If 'm' begins with a '-' the value is subtracted from the numeric register.) The arithmetic is performed as 17 bit signed integers.

Notes: The numeric registers are pre-initialized to "0" when VEDIT PLUS is invoked.

See Also: Commands: XS, XT

Example:

XA3	This increments (adds "1" to) numeric register 3.
60XA22	This adds 60 to numeric register 22.
-2XA4	Subtracts 2 from numeric register 4.

XKr*text*<ESC> Single Key Query

Example: @XK19/Press "Y" or "N":/

Description: This command queries (prompts) on the screen with '*text*' and then reads the value of the user's next keyboard character into numeric register 'r'. The command form ":**XKQr***text***<ESC>**" allows even a **<CTRL-C>** to be read from the keyboard (value = 3); otherwise, **<CTRL-C>** performs its normal function of breaking out of any command macro. The command form "**+XKr***text***<ESC>**" prompts on the status line - '*text*' must then be a single line prompt.

Notes: The query always starts on a new screen line. Since the very next keyboard character is read, there is no line editing. In programming terminology **"XK"** performs a "character input", similar to an unbuffered "getchar(r)" in C.

See Also: Commands: RQ, XQ, YT

Example:

```
@XK19/Press "Y" or "N":/
(Q19 = "Y) @JP/DOIT/
```

Prompts for confirmation. If the user presses "Y" jumps to the label "!DOIT!" in the command macro.

XQr*text*<ESC> Numeric Value Query

Example: @XQ15/Enter Line Number:/

Description: This command queries (prompts) on the screen with '*text*' and then reads the user's response of a decimal number into numeric register 'r'. The user's response may also be an algebraic expression which will be evaluated. The user normally ends the number (expression) with a **RETURN**, but any invalid character also ends the number. Pressing an immediate **RETURN** sets 'r' to "0". The command form "**+XQr***text***<ESC>**" prompts on the status line - *text* must then be a single line prompt.

Notes: The query always starts on a new screen line. The response line may be edited in the same way as a command line may be edited.

See Also: Commands: RQ, XK, YT

Example: @XQ15/Enter Line Number:/
_B Q15_L
Prompts for the desired line number. Then goes to that line in the file.

nXSr Set Numeric Register

Example: 0XS2 1200XS3

Description: This command sets numeric register 'r' to 'n'. 'n' is treated as a 17 bit signed integer.

Notes: The numeric registers are pre-initialized to "0" when VEDIT PLUS is invoked.

See Also: Commands: XA, XT

Example: 0XS9 This clears (sets to 00) numeric register 9.

12XS0 Numeric register 0 is set to 12.

XTr Type Numeric Register

Example: XT4 :XT4

Description: This command types (displays) numeric register 'r' in decimal followed by a **<CR><LF>**. The command form "**:XTr**" suppresses the final **<CR><LF>**. The number is displayed right justified. The command form "**-XTr**" displays the number left justified.

Notes: The numeric registers are pre-initialized to "0" when VEDIT PLUS is invoked.

The value of a numeric register can be printed in conjunction with the "**YP**" command or inserted into the edit buffer with the "**YI**" command.

See Also: Commands: XA, XS

Example: 10XS2 12XA2

XT2	Types out numeric register 2 displaying the value 22.
YP XT9 -YP	The number in numeric register 9 is printed out.

nYD Dump Character

Example: Q1YD Q1:YD .bYD

Description: This command dumps (displays) the ASCII character with value 'n' followed by a **<CR> <LF>**. The command form "**n:YD**" suppresses the **<CR> <LF>**. Primarily used inside command macros, with 'n' being a numeric register. It can also be used to display graphics characters on the IBM PC, where 'n' has a value between 128 and 255.

Notes: It is called a "dump" because control characters are not expanded.

Example:

Q1YD	Interprets the contents of numeric register "1" as a character and dumps it to the screen.
.bYD	Displays the name of the current edit buffer.

n1,n2YEA Change Window Attributes

Example: 6YEA 6,240YEA

Description: This command changes the screen attribute(s) for the current window. "**nYEA**" changes both the text character and "erase screen" attributes to 'n'. "**n1,n2YEA**" changes the text character attribute to 'n1' and the "erase screen" attribute to 'n2'. The "erase screen attribute" is the attribute used for the erased (or clear) portions of the window. Both attributes are usually set to the same value.

Notes: The attribute values are hardware dependent. On a CRT terminal only "0" for normal video and "1" for reverse video are supported. On an IBM PC all allowable screen attribute values are supported. Both foreground and background colors can be set on IBM CGA and EGA systems. The heading "Changing Window/Screen Color" in the User Guide gives values for most color combinations.

The default attribute values are set in Installation Tasks 10.6 and 10.7. The "**YWI**" command restores the attributes to their installed values.

Before changing the attributes inside a macro, you may want to save the current values so that they can be restored later. The current attribute values are available in the internal values "**.wa**" and "**.we**".

See Also: Installation Task 10.6 and 10.7

Example: 5YEA Changes to magenta colored characters on an IBM PC.

YEC / YEL / YES Erase Window

Example: YEC YEL YES

Description: These commands erase part or all of the current window. "**YEC**" erases (clears) the entire window and homes the cursor. "**YEL**" erases from the cursor position to the end of the window line. "**YES**" erases from the cursor position to the end of the window (screen). For each command, the erased portion of the window is set to the "erase screen" attribute initially set during Installation or changed with the "**YEA**" command.

These commands are primarily used in command macros which are creating a "menu" or "form" on the screen.

Notes: These commands perform the common CRT emulation functions of "Clear screen", "Erase to end of line" and "Erase to end of screen".

See Also: Commands: YEA, YEH, YEV

Example:

```
5YEV 15YEH
@YT/Name:/
YEL
```

Position the cursor to row 5, column 15 in the window, display the text "Name:" and clear the rest of the line.

nYEH / nYEV Set Cursor Position

Example: 5YEV 15YEH 1YEH 1YEV

Description: These commands position the cursor within the window (screen). "**nYEH**" positions the cursor horizontally to column 'n'. "**nYEV**" positions the cursor vertically to row 'n'. This cursor position determines where the next console character will be displayed. The upper left hand corner (home) has position "1,1".

These commands are primarily used in command macros which are creating a "menu" or "form" on the screen.

Notes: This cursor positioning has absolutely nothing to do with the cursor positioning in Visual Mode; it only performs a "CRT emulation" function for command macros.

See Also: Commands: YEC, YEL, YES

Example:

5YEV	Positions the cursor to row 5, leaving it in its current column.
1YEV 1YEH	Positions the cursor to "home" - the upper left hand corner of the window.

nYF Format Paragraph

Example: YF 1YF 10YF

Description: This command formats the paragraph of text that the edit pointer is in. **"YF"** is similar to the Visual Mode **[FORMAT PARAGRAPH]** function. The left margin is set by the Visual Mode indent position and the right margin is the Word Wrap column. Alternatively, **"nYF"** uses a left margin of 'n'. Use the command **"1YF"** to use a left margin of one (1), independent of the indent position. After formatting, the edit pointer is positioned to the beginning of the next paragraph. If word wrap is off, the command is ignored.

Notes: The paragraph can optionally be justified with the command switch **"ES 11 1"**. Similarly, the command switch **"ES 11 2"** allows a paragraph to be "unjustified". Note that you must use an iteration loop to justify multiple paragraphs.

See Also: Formatting Paragraphs

Examples:

5YF	Formats the paragraph with the left margin starting at column 5.
8[YF]	Formats eight (8) paragraphs, using a left margin set by the Visual Mode indent position.

YK *file*<ESC> Save Keyboard Layout

Example: YK SPECIAL.LAY

Description: This command saves the current keyboard decode table, including any keystroke macros, in the file '*file*'. This is the only way of saving keystroke macros to disk so that they can later be loaded back with the "**YL**" command. The command form "**+YL**" performs an extended search for the file.

Notes: The entire keyboard layout set up during Installation and any keystroke macros created with the **[DEFINE]** function are saved to disk.

See Also: Commands: YL
Keystroke macros

Example: YK SPECIAL.LAY Saves the current keyboard layout and keystroke macros to disk.

YI Insert Into Edit Buffer

Example: YI

Description: This command re-routes (inserts) any Command Mode console output into the buffer at the edit pointer. For example, this can be used in conjunction with the "**XT**" command to insert numeric information, such as a line or page number, into the text. "**-YI**" stops the re-routing and resumes normal console output. The "COMMAND:" prompt also stops any re-routing.

Notes: The "**YI**" command will operate very quickly near the end of the edit buffer. However, inserting text at the beginning of a large file may take as much as 1/2 second per character.

See Also: Commands: YP

Example: YI XT3 Inserts the number in numeric register 3 into the edit buffer.

YI -ED Inserts the disk directory into the edit buffer, one file name per line.

YL *file*<ESC> Load Keyboard Layout

Example: YL SPECIAL.LAY

Description: This command loads a new keyboard decode table from the file '*file*'. A keyboard decode table is saved to disk with the "**YK**" command and includes any keystroke macros which were set up when the "**YK**" was issued.

Notes: Loading a new keyboard table overwrites any existing keystroke macros. When entering Visual Mode, the "V" command checks that the keyboard table is valid. For example, if you erroneously load a text file as a keyboard table, the "**V**" command will give the error "UNABLE TO ENTER VISUAL MODE".

See Also: Commands: V, YK
Keystroke macros

Example: YL SPECIAL.LAY The file "SPECIAL.LAY" replaces the current keyboard decode table including any keystroke macros.

YM Find Matching Parentheses

Example: YM

Description: This command searches (forwards) for the next "parenthesis" type character - "(", ")", "{", "}", "[", "]", "<", ">". If the edit pointer is already at one of these characters it searches (forwards or backwards) for its "matching" character. "**YM**" properly handles nested "parentheses". The command is primarily useful for checking the syntax of structured programming languages such as "C" and VEDIT PLUS macros.

Notes: "**YM**" is similar in operation to the **[MISC]-Match** function.

Example:

YP Route to Printer

Example: YP

Description: This command re-routes any Command Mode console output to the printer. The command "**-YP**" stops the re-routing and resumes normal console output. The "COMMAND:" prompt also stops re-routing. This command can be used in conjunction with the commands **RT**, **T**, **EL** **ED**, **EV**, **XT**, etc., to print text.

Notes: There is a subtle difference between, for example, "**YP T**" and "**PR**". Most print commands, such as "**PR**", do not expand control characters sent to the printer so that special printer features can be accessed. However, "**YP**" allows text to print exactly as it is displayed on the screen where control characters are usually expanded. Therefore, "**YP T**" will expand control characters - for example, instead of sending a **<CTRL-H>** to the printer, it will send the two characters "^**H**".

See Also: Commands: YI, T, RT, XT

Examples:

YP ED	Prints the directory on the printer.
YP RT6	Prints the contents of register 6 on the printer. (Similar to "RP6".)

mYS Strip High Bit

Example: 12YS #_YS

Description: This command strips the high bit (bit 8) from all characters in the specified line range. (Line range is same as for "**T**" command.) "**YS**" is predominately used to convert Wordstar and similar word processing files into a format easier to use with VEDIT PLUS.

Notes: Be careful not to use this command on IBM PC "graphics" characters which also have their high bit set.

See Also: Convert WordStar Files

Example:

B #_YS	Strips the high bit from all characters in the file.

YT*text*<ESC> Type Text String

Example: @YT/Part 1 is done/

Description: This command displays '*text*' on the console. '*text*' may be several lines long with a **RETURN** at the end of each line. The command form "**+YT***text*" displays the text on the status line.

"**YT**" is often used in command macros to display messages, menus, forms and prompts.

Notes: The "**YEH**" and "**YEV**" cursor positions commands can be used to display the text at any position in the window (screen).

The commands "**YP YT***text***<ESC>**" can be used to print a header or footer line on the line printer.

See Also: Commands: RQ, XK, XQ, YP

Example: YP @YT/Chapter 1<CR>/
The header line "Chapter 1" is printed. (The **<CR>** starts a new line on the printer.)

YWD / YWI Delete / Initialize Window

Example: YWI

Description: "**YWD**" deletes the current window and "**YWDw**" deletes window 'w'. The remaining window(s) will expand in size to fill the freed screen area. Only the default "@" window cannot be deleted. The new active window is the "parent" of the deleted window. This command does not change the active edit buffer.

"**YWI**" deletes all windows and initializes the default "@" window to the same condition as when VEDIT PLUS was invoked - it fills the entire screen (except the status line), clears the window and sets the text and "clear screen" attributes to to their Installation values.

Notes: You may need a "**YWS**" command to switch to the desired window following a window deletion.

See Also: **[WINDOW]** function

Example: YWD4 YWS1 Delete window "4" and switch to window "1".

YWB / YWL / YWR / YWT Create Window

Example: YWR 1 40 YWB $ 5

Description: These commands create a new window by splitting the current window into two windows. "**YWBw n**" creates a new window with name 'w' and size of 'n' lines in the bottom of the current window. Similarly, "**YWTw n**" creates window 'w' with 'n' lines at the top. "**YWRw n**" creates window 'w' with 'n' columns in the right of the current window. "**YWLw n**" creates window 'w' with 'n' columns at the left. 'w' may be any single character name. If an edit buffer is to be displayed in the window, the name of the edit buffer must be picked. The window with name "$" will be used by the Command Mode.

Notes: The current window is reduced in size (including the border line needed for the n but cannot be reduced smaller than one columns. These commands do not switch t window; this is done with the "**YWSw**" comman

The screen attributes for the new win changed with the "**YEA**" command. On an IB allows windows to be displayed in different

See Also: Commands: YEA, YWD, YWI, YWS
[WINDOW] function

Example: YWR1 40 Create window "1" with 40 the right part of t window. It can be used edit buffer "1".

YWB$ 5 Create window "$" with the bottom of the curre It will automatically be Command Mode.

YWSw Switch Window

Example: YWS1 YWS$

Description: This command switches to the window 'w'. If window 'w' does not exist, the command is ignored - no error is given. "**YWS**" only switches to a different window; it does not also switch edit buffers.

Notes: If "**YWS**" switches to a window which was last used for Visual Mode the text is scrolled up one line and the cursor positioned at the bottom of the window; otherwise the cursor is positioned to its previous position in the window.

Whenever you exit Visual Mode to Command Mode, VEDIT PLUS will automatically switch to the "$" window if it exists. Otherwise it will use the previous Command Mode window.

See Also: Commands: RQ, XK, XQ, YP
[WINDOW] function

Example: YWS1 Switch to window "1".

YWZ Zoom Window

Example: YWZ

Description: This command "zooms" the current window so that it fills the entire screen (except for the status line). It is often easier to edit in the larger window.

Notes: Anytime VEDIT PLUS switches windows and one window was "zoomed", all windows are redrawn on the screen. Therefore, entering Visual Mode will usually cause the windows to be redrawn. You can "zoom" the Visual Mode window with the **[WINDOW]**-Zoom function.

When redrawing the screen, VEDIT PLUS restores the contents of all Visual Mode windows, but it cannot restore the contents of Command Mode windows.

See Also: Commands: YWI, YWS

Example: YWZ Zoom the current window to fill the screen.

This Page Reserved For Your Notes

This Page Reserved For Your Notes