VEDIT 6.0

Programmable
Universal File
Editor / Translator

Macro Language Reference Manual

Greenview Data

VEDIT

Programmable, Universal File Editor and Translator
For Text, Program, Database, Binary
And Mainframe File Editing
Version 6.03

Manual Written By:
Theodore Green & Timothy McLean

Programmed By:
Theodore Green & Thomas Burt

Greenview Data, Inc.
2773 Holyoke Lane
Ann Arbor, MI 48103
Telephone: (734) 996-1300
Fax: (734) 996-1308
E-Mail: support@vedit.com
Visit our Web site: www.vedit.com

DISCLAIMER

Greenview Data, Inc. and the authors make no claims or warranties with respect to the contents or accuracy of this publication, or the product it describes, including any warranties of fitness or merchantability for a particular purpose. Any stated or expressed warranties are in lieu of all obligations or liability for any damages, whether special, indirect, or consequential, arising out of or in connection with the use of this publication or the product it describes. Furthermore, the right is reserved to make any changes to this publication without obligation to notify any person of such changes.

Last Manual Revision: May. 10, 2002

ACKNOWLEDGEMENTS

We would like to thank the following people for their assistance.

Christian Ziemski for the exceptionally thorough beta-testing of new versions and assistance in setting up the Discussion Conference on our Web site.

Scott Lambert for the numerous suggestions, for supporting other users in the Discussion Conference, and for running his own "VEDIT macro" Web site.

Maxim Glukhov for writing the new BOX-DRAW.VDM and ASCII2.VDM macros supplied with VEDIT.

Wayne Barrett, for donating many hours to editing and enhancing this manual and for his helpful feedback over many years.

Peter Freed of Data Base Management Systems, Inc. for writing the CMD-CONV.VDM, DBASE.VDM and WS6.VDM (enhanced WordStar emulation) macros supplied with VEDIT.

This manual was created using Corel Ventura in conjunction with VEDIT. V-SPELL was used for spelling correction.

TRADEMARKS

TABLE OF CONTENTS

Chapter 1
Introduction

Welcome to VEDIT

Purpose of Program

VEDIT is an editor/translator designed not only for text preparation and program development, but also for editing large database, mainframe and binary files. It can edit in ASCII, EBCDIC, Hexadecimal, Octal or any combination of modes. It supports variable length and fixed length database records.

The powerful macro language makes VEDIT an ideal alternative to conventional programming languages such as Basic, C or Pascal when writing file translators, converters and filters. A single-line "macro" can often perform the equivalent of a 100+ line C program. Translation between ASCII and EBCDIC, and sorting are built in.

Since VEDIT can efficiently edit any type of file, including binary/data files and huge files up to 2 Gigabytes (2000 Megabytes) in size, it is ideal for editing and translating files downloaded from Mainframe computers and CD-ROM data files.

VEDIT can process entire groups of files automatically. For example, the same edit changes can be applied to all files in a directory, or even in all subdirectories.

Powerful, Easy to Use Macro Language

Besides the normal "Visual Mode" editing, VEDIT additionally has an interactive "Command Mode" which gives instant access to over 400 powerful macro language commands using a C-like programming syntax. (In computerese: The VEDIT macro language is interpreted and not compiled.)

This "Command Mode" makes the VEDIT macro language exceptionally easy to use. (It is much, much easier to learn and use than C.) Any desired sequence of commands can be entered at the "COMMAND:" prompt. The command(s) execute immediately when you press **<Enter>**; the main window lets you observe the effect the command(s) had on your file. Yes, there is undo.

The macro language is ideal for automating any repetitive editing operations. Macros can be saved as files for future use, can be assigned to "hot-keys" to add custom editing features and can be added to the **{USER}** menu. Macros can easily be modified as needed.

VEDIT can be used as an application programming language. Most macros developed with VEDIT only require VEDIT to run. With many applications, users would need to know little or even nothing about VEDIT itself.

This manual builds on the VEDIT manual and describes the VEDIT macro language in detail.

Macro Language Features

- Instant access to over 400 powerful commands. Any desired sequence of commands can be entered at the "COMMAND:" prompt. Allows "off-the-cuff" macros; there is no compiling.
- Command Mode window. Visual and Command modes can operate in separate windows — you can observe the effect of Command Mode commands in the Visual Mode window. This is useful for learning the macro language and for debugging macros.
- Extensive on-line help describes each command and its options. Many short examples are given.
- Extensive set of file handling commands to open, close and save files, manage buffers and windows. Files can be copied, moved and renamed. File attributes can be checked or changed.
- Command oriented configuration. All aspects of the configuration can be examined and modified with the macro language.
- Command macros (programs) can be saved and loaded as files, can be permanently assigned to a "hot-key" (keystroke macro) or can be added to the **{USER}** menu.
- Commands permit editing by character, line, block or file. Character, columnar and line oriented blocks are supported.
- Complex macros can be simplified and modularized by splitting them into "subroutines". Other macros can be accessed by a "Call" or "Chain" command.
- Command macros can be automatically executed when VEDIT is invoked.
- "Bomb proof" macros can be written that return to a main menu (or perform other operations) when unusual conditions occur.
- Event macros. Special macros can be executed for each file opened and closed. Permits automatic file translations or checking a file's integrity before it is saved to disk. Event macros are used to implement language-specific "template editing".
- On-line calculator. Any algebraic expression entered at the "COMMAND:" prompt is immediately calculated and displayed in either decimal or hexadecimal.
- Complete numerical capability with 32 bit resolution (+/- 2,147,483,613). Complete set of "C-like" numeric, relational and logical expressions. Over 250 numeric variables.

- Over 100 text registers can be used as "cut and paste" buffers, as string variables or to hold subroutine macros. Permits the use of variable filenames, search/replace strings and much more.
- "C-like" program flow control with **While**, **Do-while**, **For**, **If-then**, **If-then-else**, **Goto**, **Break**, **Continue** and **Return** statements. Includes additional flow control statements not found in C.
- Powerful search and replace can use pattern matching or regular expressions. Flexible options can select character or columnar block search/replace, selective or global replace, search every 'n'th occurrence and much more.
- Flexible matching and compare commands for numeric, character and string testing.
- Interactive input and output. Messages and prompts can be displayed under macro control. Input can be in the form of single characters, character strings, numbers, or numerical expressions.
- Keyboard layout can be dynamically changed. Individual key assignments can be added or deleted, keystroke macros can be defined, and a complete new layout can be loaded from disk.
- Command macros can also access any basic editing functions or menu items.
- Flexible window control. The creation, deletion, size, position and color of windows is fully programmable. Address, write and erase any window. Permits writing application programs with windows, menu prompts and forms entry.
- Macro language can shell out to the operating system to execute other programs and OS commands.
- (DOS only) Direct hardware access. Memory and I/O ports can be examined and modified. DOS functions, BIOS functions and interrupts can be directly called. Blocks of memory can be read and written. Assembly language routines can be stored in VEDIT's memory spaced and called from the macro language.
- When necessary, macros can easily be debugged using breakpoints, tracing and a debugging window.

Using this Manual

This manual assumes that you have a working knowledge of the "Visual Mode" of VEDIT as described in the VEDIT User's Manual.

Chapters 2 and 3 are intended to be read sequentially; new topics usually build on previous topics. However, once Chapters 2 and 3 are understood, the experienced user will primarily only use Chapter 4, which is the detailed command reference.

HINTS: The best way to learn VEDIT is with the tutorial items in the **{TUTOR}** menu. If the **{TUTOR}** menu is not displayed, select **{MISC, Load Tutorial menu}**. The tutorial covers VEDIT's features in great depth — it is well worth the one or two hours it takes to carefully work through it.

With VEDIT's extensive on-line help, we hope that you won't need this manual very much.

This manual is organized into the following chapters:

Chapter 1 - Introduction

Introduces VEDIT.

Chapter 2 - Command Mode Guide

Introduces command macros which are the "programs" of VEDIT. Step-by-step examples show how to run a simple command macro from the "COMMAND:" prompt, as a keystroke macro, from the **{USER}** menu and from a file. It also introduces basic commands for exiting and quitting the editor, for editing new and additional files and other commands that are often used individually.

Chapter 3 - Programming Guide

Covers the main topics of writing command macros, including flow control, numeric expressions, interactive input and output, block operations, and more technical topics.

Chapter 4 - Command Reference

Covers each macro language command in complete detail.

Appendices

The following topics are included in the appendices.

A summary of the command language syntax including numeric expressions and string arguments.

A summary of the all macro language commands.

A summary of text register usage, including a description of all the reserved registers.

Chapter 2

Command Macro Guide

This manual covers the VEDIT macro language in complete detail. Even if you have never written a computer program, you will find the VEDIT macro language easy to use. This is true because some useful macros consist of just a single short command! Combining just a handful of commands lets you easily perform complex tasks or add custom editing features to VEDIT.

This chapter begins with simple step-by-step examples of the four main ways that the macro language can be used. You will then know exactly how to add custom features to VEDIT.

This chapter then explains the unique VEDIT "Command Mode". In this mode you simply type the desired macro language commands and press **<Enter>** — the commands are immediately executed.

The rest of this chapter introduces basic commands that you are likely to use to display status information, get on-line help, create windows, edit new files and exit VEDIT.

The next chapter (Programming Guide) covers how commands are combined to create more powerful command macros.

Definition of "Command Macro"

We use the term *command macro* to refer to any sequence of macro language commands. You can think of a command macro as a "program".

We are careful to distinguish command macros from *keystroke macros*. A keystroke macro is a sequence of recorded keystrokes, such as a hot-key to a menu item; this has nothing to do with command macros. Similarly, a command macro that is stored as a file has nothing to do with keystroke macros.

However, as you will soon see, keystroke macros can also contain command macros. This lets you assign command macros, such as custom editing functions, to a hot-key.

Easy as 1 - 2 - 3 - 4

There are four ways to run VEDIT macros. This topic gives step-by-step directions for creating and running a simple command macro in each of the four ways. The four ways are:

- Enter Command Mode and simply type in the macro at the "COMMAND:" prompt.
- Add the macro to the keyboard layout, making it a keystroke macro that is activated with a hot-key. This is preferred for commonly used, simple macros.
- Add the macro to the **{USER}** menu.
- Create the macro as a file (typically with the .VDM extension) and execute it from **{MISC, Load/execute macro}** or when you invoke VEDIT. It can also be added to the **{USER}** menu.

"COMMAND:" Prompt Example

VEDIT has a special "Command Mode" in which you simply type one or more macro language commands that are immediately executed.

It is often easier to use the Command Mode by first creating a special Command Mode window. As each command line is executed, you can then immediately see its effect on your file. A handy five line window can be created by selecting **{ESCAPE, Command mode window}**.

You can enter Command Mode by selecting **{ESCAPE, Command Mode}**. (Other ways are described later.) VEDIT then prompts you with "COMMAND:" in the Command Mode window.

Our example command macro is just a single command that switches the case of all letters from the cursor to the end of the line. We will show you all four ways in which the same command macro can be run. The command is:

```
Case_Switch_Block(Cur_Pos,EOL_Pos)
```

(A single-command macro may seem overly simplistic, but the same principles will apply to macros of any size.)

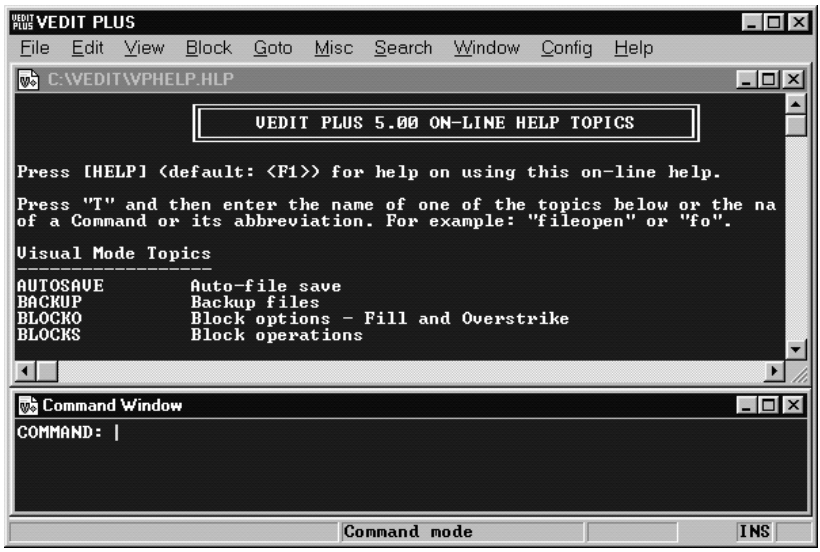
Here we go, step by step, assuming you haven't started VEDIT yet:

➤ **"COMMAND:" prompt example:**

1. Start up VEDIT, select **{FILE, Open}** (default: **<Ctrl-O>**) and open up a text file that you can safely modify.

   Position the cursor in the middle of a line that has upper and lower case letters.

2. Select **{ESCAPE, Command mode window}** (default: **<Alt-/>**) to create the special Command Mode window. You should now see:



3. For the sake of experimentation, enter the command "**line**" (without quotes) followed by **<Enter>** and notice that the cursor in the top window has moved to the next line. Enter the command "**bof**" to move the cursor back to the beginning of the file.

4. Now enter the command to switch the case of all letters from the cursor to the end of the line; it can all be entered in lower case.

   Case_Switch_Block(Cur_Pos,EOL_Pos)

   You should be able to see the effect of the command on the file displayed in the upper window as soon as you press **<Enter>**.

5. Enter the command "**visual**" or its abbreviation "**v**" to return to normal Visual Mode editing.

-OR-

Press **<Alt-/>** again to remove the Command Mode window and return to normal Visual Mode editing.

Keystroke Macro Example

Commonly used command macros can be set up as keystroke macros and assigned to hot-keys. The topic "Keystroke Macros - Adding Keystroke Macros from KEY-MAC.LIB" in Chapter 4 of the VEDIT User's manual also describes this procedure.

Although keystroke macros that use the macro language could be set up with **{CONFIG, Keyboard layout, Add macro}**, we suggest using **{CONFIG, Keyboard layout, Edit layout}** instead. Since you are then editing your VEDIT.KEY file (which is automatically loaded when you start VEDIT), you can easily copy and paste the macro commands from KEY-MAC.LIB or another source directly into the VEDIT.KEY file without retyping.

For this example, we will use the same command macro that switches the case of all letters from the cursor to the end of the line.

Here we go, step by step, assuming you are in VEDIT:

➤ **Keystroke macro example:**

1. Select **{CONFIG, Keyboard layout, Edit layout}**. Your current keyboard layout is displayed as a normal text file that you can directly edit.

   Go to the end of the file.

2. Select **{CONFIG, Keyboard layout, Unused keys}** to view a list of unassigned keys. You can assign the keystroke macro to any of these keys. We'll assume here that **<Shift-F12>** is available.

3. Enter the following line at the end of the keyboard layout:

   Shft-F12 [VISUAL EXIT] Case_Switch_Block(Cur_Pos,EOL_Pos)

   Be sure to hit **<Enter>** at the end of the line. (Your cursor should be in column 1 of the following line; there must not be any additional blank lines.)

4. Press **[VISUAL EXIT]** (default: **<Ctrl-E>**) when finished editing. You are prompted with:

   [I]gnore changes, [T]emporary, [S]ave into VEDIT.KEY

   If you only want the keystroke macro to work until you exit VEDIT, type "**T**"; to make the keystroke macro permanent, type "**S**".

5. Open a file for editing, if you haven't already. Press the hot-key, e.g. **<Shift-F12>** and you should be able to see its effect.

This technique can be used to easily copy and paste any desired keystroke macro from the supplied KEY-MAC.LIB file into your own keyboard layout. KEY-MAC.LIB contains many useful functions that can be added as keystroke macros.

{USER} Menu Example

You can easily modify the supplied **{USER}** menu, adding and deleting items as desired.

For this example, we will implement the same macro above, which switches the case of letters, as an item in the **{USER}** menu instead of as a keystroke macro.

Here we go, step by step, assuming you are in VEDIT:

➤ **{USER} menu example:**

1. Open the file "user.mnu" for editing. (It is in the VEDIT Home Directory.)

   Go to the end of the file.

2. Enter the following three lines at the end of the file:

9
Switch case to EOL
Case_Switch_Block(Cur_Pos,EOL_Pos)

Be sure to press <**Enter**> after each of the three lines.

For detailed information about the layout of the **{USER}** menu, refer to the on-line help topic "{USER} and {TOOL} Menus" (DOS: "USER").

3. Save the new "user.mnu" file.
4. Exit VEDIT and start it up again. The new **{USER}** should be loaded and you can try out the new item "Switch case to EOL".

Creating a Macro as a .VDM file

More complex command macros are best created as files, typically with a ".vdm" filename extension. VEDIT is supplied with dozens of ".vdm" macros. Some are loaded and run when you select menu items. For example, **`wildfile.vdm`** is run when you select **{MISC, WILDFILE macro}**; **`sallbuff.vdm`** is used by **{SEARCH, Search all buffer}**.

Any ".vdm" macro file can be loaded and run with **{MISC, Load/Execute macro}**.

There is nothing special about ".vdm" files; they are simple text files that contain the macro language commands exactly as you see them in this manual. (Remember, VEDIT macros are not compiled.)

Once again, we will implement the previous switch case macro as a ".vdm" file. You will then have implemented the same macro as a keystroke macro, as an item in the **{USER}** menu and as a ".vdm" file.

In practice, you would rarely create a single command ".vdm" file, but it is possible.

Here we go, step by step, assuming you are in VEDIT:

➤ **Creating a Macro as a .VDM file:**

1. Select **{FILE, Open}** (default: <**Ctrl-O**>) and enter the filename for the macro. Let's use "swcase.vdm" in the *VEDIT User Macro Directory*, typically **c:\vedit\user-mac** or **c:\program files\vedit\user-mac**.

   You should now have an empty editing window and "New file" should temporarily be displayed on the status line.

2. Type in the same macro as before:

   Case_Switch_Block(Cur_Pos,EOL_Pos)

3. Select either **{FILE, Close}** (default: <**Ctrl-W**>) or **{WINDOW, Close}** (default: <**Ctrl-F4**>) to close the file and save it to disk.

4. Open a file for editing, if you haven't already.

   Select **{MISC, Load/execute user macro}**. Select or enter the filename "swcase.vdm". (The default register number "100" is fine.) As soon as you

close the dialog box, the macro will run and convert the case of letters on the current line in your file.

5. Since the macro remains loaded, you can select **{MISC, Execute macro}** and select the default register number of "100" to run the macro again.

This obviously isn't as convenient as running a command macro from a hot-key, but demonstrates how a less often used macro can be run. Command macros in ".vdm" files can also be attached to hot-keys or to an item in the **{USER}** menu; this is described next.

Running Bigger Macros

(You may want to skip this topic for now if you are new to VEDIT.)

Bigger command macros are usually saved to disk as ".vdm" files. Although there is no absolute limit on how large a command macro you can build into a keystroke macro or the **{USER}** menu, we suggest saving any macro larger than a few hundred bytes as a ".vdm" file.

It is easy to attach a command macro in a ".vdm" file to a hot-key or to an item in the **{USER}** menu by using the **Call_File( )** command. It is equivalent to **{MISC, Load/Execute macro}**; it loads and runs the specified ".vdm" file.

To see an example of this, you may want to examine the supplied "user.mnu" file. The item "Color scheme" is implemented with the following command:

```
#103=Reg_Free Call_File(#103,"color.vdm") Reg_Empty(#103)
```

The command **Call_File(#103,"color.vdm")** loads and runs the macro in the file "color.vdm". (Technical: the remaining commands find a free text register in which to load the macro, and empty the register when done.)

Similarly, the **Call_File( )** command can be used in a keystroke macro to load and run a command macro saved as a ".vdm" file.

Please use the On-Line Help!

Please use the on-line help for more information about a particular command. It is constantly updated and describes every command and every command option, including any recent additions that are not in this manual.

It is organized differently from this manual and includes additional topics and examples.

Since the on-line help file "vphelp.hlp" is just a text file, you can easily open it in VEDIT and print any desired portions. ("vphelp.hlp" is used by the non-Windows versions of VEDIT.)

Command Mode

The VEDIT Command Mode lets you enter macro language commands at the "COMMAND:" prompt. They are immediately executed when you press **<Enter>**. This is a very convenient way to enter any desired "off-the-cuff" macros and learn the macro language.

You may want to run the Command Mode in its own special window. As each command line is executed, you can then immediately see its effect on your file. A handy 5 line window can be created by selecting **{ESCAPE, Command mode window}**.

Entering Command Mode

Assuming you are in the normal Visual Mode of VEDIT, you can enter Command Mode by selecting "Command Mode" from the **{ESCAPE}** menu, or by pressing the key for **[VISUAL EXIT]** or **[VISUAL ESCAPE]**.

[VISUAL EXIT]	**<Ctrl-E>** **<Ctrl-F10>**	Exit Visual Mode and enter Command Mode. If any command macro is currently running, it will continue running.
[VISUAL ESCAPE]	**<Alt-F10>** **<Ctrl-Shift-E>**	Exit Visual Mode and enter Command Mode. If any command macro is currently running, it is stopped. Same as **{ESCAPE, Command mode}**.

Normally there is no difference between these two functions and you will get the "COMMAND:" prompt.

However, if a macro is already running, **[VISUAL EXIT]** will let it continue running, whereas **[VISUAL ESCAPE]** will stop it. With the exception of "locked-in" macros, **[VISUAL ESCAPE]** will always give you the "COMMAND:" prompt.

If you already created a Command Mode window, e.g. with **{ESCAPE, Command mode window}**, the "COMMAND:" prompt will appear in this window. Otherwise your current window will scroll up and the prompt will appear at the bottom of the window.

The first few times the "COMMAND:" prompt is displayed, it is preceded with a help line to remind you of a few frequently used commands.

See Also:

The on-line help topic "Command Mode basics" (DOS: "CMD").

Return to Visual Mode

To exit the Command Mode and return to the Visual Mode, enter the **Visual( )** command; it can be abbreviated as just **V**. All commands end with **<Enter>**.

Visual( ) Enter Visual Mode. You will remain in Visual Mode until you select **[VISUAL EXIT]** (**<Ctrl-E>**) or **[VISUAL ESCAPE]** (**<Ctrl+Shift-E>** or **<Alt-F10>**).

V Use this convenient abbreviation for **Visual( )**. Many common commands have a short abbreviation.

EXIT Exit VEDIT. You will be prompted to save or abandon each modified file. This command is identical to **{FILE, Exit}**.

Command Lines

In Command Mode you are prompted for "*command lines*" by the "COMMAND:" prompt. Each command line you enter consists of a single command or a sequence of multiple commands. Each command line is ended by pressing **<Enter>**, at which time the command line is executed.

Since no commands are executed until you press **<Enter>**, the line can be edited as you enter it. Previous command lines can also be reused by pressing **[CURSOR UP]**. Once execution begins, it can usually be aborted by pressing **[CANCEL]** (default: **<Ctrl-\>**), **<Ctrl-C>** or **<Ctrl-Break>**. This results in the "*BREAK*" message and a new "COMMAND:" prompt. VEDIT checks for **[CANCEL]** before any new command is executed, at the end of each line displayed on the screen and during commands which repeatedly access the disk.

The maximum command line length is 256 characters. Longer commands are more easily handled as command macros — the macro is copied to a text register or loaded from a file.

Occasionally you may see a "<<" displayed at the end of a line preceding the "COMMAND:" prompt. This only indicates that the previous line did not end in a "newline". It can occur when you use the "NOCR" option on selected commands to suppress the newline.

Command Line Editing

The command line can be edited in the same way as a string in a dialog box.

Characters typed in the middle of a command line will either be inserted or overstrike existing characters, depending upon the Dialog insert mode. The Dialog insert mode is independent of the Visual insert mode.

[INSERT TOGGLE] **<Ins>** Toggle the Dialog Insert Mode between Insert and Overstrike modes.

After editing the command line press **<Enter>** — it is NOT necessary to first move the cursor to the end of the line.

To enter control characters on the command line, first press **[ENTER CTRL]** (default: **<Ctrl-Q>)** and then the control character, e.g. **<Ctrl-X>** or **<Enter>**. **[ENTER CTRL]** causes the following character to be taken literally, and not to be interpreted as a line editing character or any other special character.

Reusing Previous Command Lines

The previous ten or more command lines can easily be reused.

[CURSOR UP] Accesses the previous command line.

[CURSOR DOWN] Accesses the next command line.

A previous command line can be edited before pressing **<Enter>**.

Using the Scratchpad

You can easily transfer command lines to and from the scratchpad (text register 0). For example, you can copy a line of macro language commands from a file to the scratchpad: press **[T-REG COPY]** (default: **<Ctrl-F11>**), select register "0", enter Command Mode, and press **[T-REG INSERT]** (default: **<F11>**).

[T-REG COPY]	**<Ctrl-F11>** **<Numpad+>**	Copy the entire command line to the scratchpad.
[T-REG MOVE]	**<Alt-F11>** **<Numpad->**	Move the entire command line to the scratchpad.
[T-REG INSERT]	**<F11>** **<Numpad*>**	Insert the scratchpad into the command line.

When using these **[T-Reg...]** functions at the "COMMAND:" prompt, you are not prompted for the register number; it automatically uses register 0.

Command Syntax

VEDIT's command syntax loosely follows the syntax of the "C" programming language. The format of commands is:

Command(arguments)

Command names can be entered in any combination of upper and lower case letters. To improve readability, we usually capitalize the first letter of each command word, e.g. **Type_Space( )**.

The "_" character is optional and is only intended to improve readability.

Most commands have a short abbreviation. It is often, *but not always*, the first letter of each command word.

Therefore, the following commands are identical:

Type_Space( ) typespace( ) TS( ) ts( )

Many commands take one or more arguments which must be enclosed in parentheses "**(...)**". With a few exceptions, commands that take no arguments can have the empty "**( )**" left off.

NOTES: The macros we supply on disk and the examples in this manual usually use the full command name and include empty "**( )**" for commands that perform an operation and leave the "**( )**" off for the commands (called "*internal values*") that only return a value. However, we usually abbreviate the command **Visual( )** as just **V**.

Commands that take a single numeric argument, e.g. **Type_Space( )**, will use the default argument of "1" if no argument is specified.

Therefore, the following commands are identical:

Type_Space(1) Type_Space( ) Type_Space TS

Multiple commands may be typed one after another on a command line. They are always executed left to right. Their effect is the same as if each command had been typed on its own command line. For clarity's sake, you should leave a space between the commands. For example, the three command lines, each with a single command:

```
Begin_Of_File
Print(ALL)
V
```

are equivalent in operation to the single command line with three commands:

Begin_Of_File Print(ALL) V

You will often want to use a sequence of commands over and over again. We refer to any sequence of commands as a "*command macro*". As described earlier in this chapter, command macros can be saved to disk as ".vdm" and/or assigned to keystroke macros, or added to the **{USER}** menu for easy access.

Chapter 3 (Programming Guide) covers command macros in more detail.

Command Arguments

VEDIT commands take two types of arguments - numeric and string. Arguments must be enclosed in **(...)** following the command name. When there are two or more arguments, they must be separated from each other with commas.

Each numeric argument can be a numeric expression consisting of numeric constants (e.g. **12345**), numeric variables (e.g. **#10**), reserved words (e.g. **ALL**) or the return value from a command (e.g. **File_Size**). Chapter 3 (Programming Guide) covers numeric expressions in detail.

Numeric arguments have the range of +/- 2,147,483,647. When a large number is needed, for example to specify an "infinite" repeat count, the reserved word "**ALL**" can be used; its value is greater than one billion.

For example, the command **Print(*n*)** prints '*n*' lines of text. The command **Print(1000000)** could be used to print the entire file (up to 1 million lines anyway). However, the following is preferable:

Print(ALL) Print the entire file, starting at the current edit (cursor) position.

String Arguments

Each string argument can either be a string constant (e.g. **"hello"**) a string variable stored in a text register (e.g. **@20**), or one of the predefined string values, (e.g. **CUR_DIR**).

A string constant is enclosed in "*delimiters*" that cannot occur in the string. The allowable delimiters are:

` ' " % & * , . : ; / ~ ^ =

As a convention, we use double-quotes as the string delimiters whenever possible. When the string contains double-quotes, we use single-quotes or the forward-slash.

```
Ins_Text(/The name is "TOM"/)
```

Since the normal double-quote character is part of the string constant, "/" is used as the string delimiter.

String arguments are often accessed from text registers which can be used as string "variables". The syntax ***@r*** uses text register '*r*' as the entire string argument. The register can contain any characters, even including the string delimiter.

```
Reg_Set(20, /The name is "TOM"/)
Ins_Text(@20)
```

Text register 20 is used as a string variable in the **Ins_Text( )** command. It performs the same insertion as the previous example.

Although not available for all string arguments, all filename arguments plus the **Search( )**, **Replace( )**, **Match( )**, **Statline_Message( )**, **System( )** and **Goto** commands can use a text register as a *portion* of the string argument. The syntax **|@(*r*)** uses text register '*r*' as a portion of the string argument.

The following equivalent commands display a directory of all "chapter1.*" files in the current directory.

```
Directory("chapter1.*")

Reg_Set(10,"chapter1.*")
Directory(@10)

Reg_Set(10,"chapter1")
Directory("|@(10).*")
```

The predefined string values can be used anywhere that a string constant can be used.

Reg_Set(10,CUR_DIR) Set text register 10 to the pathname of the current directory.

Appendix D gives a description of all predefined string values.

Command Options

Many commands have an optional argument referred to as "*options*". Technically it is a normal numeric argument, but we highly recommend that any options be specified using the appropriate reserved words. When two or more options are needed, the corresponding reserved words are ORed together with "|" or added together with "+". (Note: "|" is <Shift-\>.)

NOTE: It is better to "OR" the options together with "**|**" in case you inadvertently specify the same option twice. For example, the options "CASE | ... | CASE" works properly, while "CASE + ... + CASE" gives unpredictable results. However, we use "**+**" in this manual because it is more readable.

For example, the command **Search("text", REVERSE+NOERR)** performs a search in the reverse direction and suppresses the error message if the string is not found.

The command option "COUNT" is followed by an additional numeric argument, often indicating the number of times the command is to be repeated.

Search("text",COUNT,3)

The command option "COLSET" is followed by two additional numeric arguments that set the explicit columns for a columnar block operation.

Search_Block("text",Block_Begin,Block_End,COLSET,10,40)

When "COUNT" and "COLSET" are both used, the argument for "COUNT" comes first. Combining the two examples above gives:

Search_Block("text",BB,BE,COUNT+COLSET,3,10,40)

NOTE: Although all reserved words are numeric constants, your macros should always use the reserved word because their specific value may change in future versions.

Multiple Line String Arguments

The string argument in commands such as **Search("*ss*")**, **Ins_Text("*text*")** and **Message("*mtext*")** can be several lines long. If you press **<Enter>** before the second delimiter, the "newline" becomes part of the string argument and the command waits for the rest of its string — the command prompt then changes to "**-**".

Prompt: "-" VEDIT is waiting for the second string delimiter.

For example, the following command will search for two lines:

Search("This is line 1<Enter> Note: Prompt changes to "-" here.
This is line 2")<Enter>

The screen display just before pressing the second <**Enter**> is:

```
COMMAND: Search("This is line 1
-This is line 2")■
```

The "**-**" command prompt is normal for string arguments that contain <**Enter**>. However, if you have made a mistake and unexpectedly receive the "**-**" prompt, press **[CANCEL]** (<**Ctrl-**\>) or <**Ctrl-C**> to abort the command.

The **Message( )** command can take a multiple line string argument or the sequence "**\n**" can be used be specify a "newline". Therefore the following two commands are equivalent.

```
Message("Line 1
Line 2
Line 3")

Message("Line 1\nLine 2\nLine 3")
```

Command Return Value On-Line Calculator

Each command executed returns a numeric value. For some commands, such as **Search( )**, this value indicates whether the command succeeded. The only purpose for some commands, such as **File_Size**, is to return status information. Most commands simply return "1".

The "COMMAND:" prompt has a handy on-line calculator feature that evaluates any numeric expression that you enter. It can also display the return value from any command.

12000/25+123	Evaluate the numeric expression and display the answer in decimal.
$ 12000/25+123	Evaluate the numeric expression and display the answer in hexadecimal.
. File_Size	Display the current file's size in decimal.

The special operator "." forces VEDIT to display the following numeric expression in decimal; if it is a command, it displays the command's return value. For commands that do nothing but return a value, such as **File_Size**, the "." is not necessary, but in general it is.

The special operator "$" forces VEDIT to display the following numeric expression in hexadecimal.

Command Mode Window

The best way to experiment in Command Mode is to create a dedicated Command Mode window named "**$**". A major advantage of this is that you

can observe in the Visual Mode window(s) the effect of the Command Mode commands. The cursor displayed in the Visual Mode window will correspond to the Command Mode "edit position".

As described earlier, **{ESCAPE, Command mode window}** creates a handy five line for this purpose. Alternatively, you can use **{WINDOW, Create}** or the **Win_Reserved( )** command to create a window of any size.

Win_Reserved($,10,BOTTOM) Create the special Command Mode window "$" on the bottom with 10 lines.

Update Re-display the current file in Visual Mode and immediately return to Command Mode.

This illustrates an application of the **Update** command which only updates the Visual Mode window and immediately returns to Command Mode. Since the "**$**" window now exists, it also switches to this window.

All edit changes performed in Command Mode can now be immediately observed in the Visual Mode window.

Controlling Screen Display

Some commands can display dozens or even thousands of lines of text. When any commands attempt to display more than one window of text, VEDIT prompts on the status line with:

Display more text [More] [Non-stop] [Cancel]

Select **[More]** (the default) to permit the display of up to another window full of text. Select **[Non-stop]** to temporarily turn off this feature so that all the text can quickly be displayed. Select **[Cancel]** to abort the commands that are running. This feature is reset each time VEDIT waits for keyboard input.

This feature can be disabled by setting **{CONFIG, Display options, Enable -MORE- operation}** to "No".

Screen output can also be temporarily stopped by pressing **<Ctrl-S>**. Pressing any other key, but typically another **<Ctrl-S>**, resumes the screen output.

During lengthy screen output, you can press **[CANCEL]** (**<Ctrl-\>**), **<Ctrl-C>** or **<Ctrl-Break>** to abort the command macro that is running. This typically returns you to the "COMMAND:" prompt.

Basic Commands

Help Command

On-line help is available in the Command Mode via the **Help( )** command.

Help( )	Start up the on-line help for the Command Mode. Starts with the topic "Command".
H	Abbreviation for **Help( )**.

From Command Mode, the on-line help starts with the topic "Commands" which is a complete list of commands (including abbreviations). The detailed command descriptions are organized into topics containing related commands.

Windows version: Simply click on the name of any command to link to the detailed description of that command.

DOS version: Select "[T]opic" and enter the name of any command to go to the detailed description of that command.

You can also go directly to the on-line help for a specific command.

H("file_open")	Go to the on-line help detailed description of the **File_Open( )** command.

You can also access the Command Mode help from Visual Mode by indexing to the topic "Commands" or to a particular command. For example:

➤ **To access on-line help for Replace( ) from Visual Mode:**

1. Press **[HELP]** (**<F1>**).
2. Windows version: Select "Search".

   DOS version: Type "T" to enter a new help topic.
3. At the help topic prompt enter "Replace". You should now see the on-line help for the command.

Exiting VEDIT

The **Exit** command is identical to **{FILE, Exit}** and exits VEDIT after prompting whether each modified buffer (file) is to be saved or abandoned.

Exit	Prompt whether each buffer is to be saved or abandoned. Then exit VEDIT.

To save all files and exit quickly without any prompts, use the **Xall( )** command. It is equivalent to **{FILE, Exit}** followed by selecting **[Save all]**.

Xall	Save all files without prompting and exit.

Note that **Xall( )** does not save modified buffers that have no assigned filename, whereas **Exit( )** will prompt for the filename. However, as a precaution,

Xall( ) will prompt for a filename if the main buffer "1" has no assigned filename.

If you want to quit (abandon) all files and exit, use the **Qall( )** command. It is equivalent to **{FILE, Exit}** followed by selecting **[Quit all]**".

Qall	Quit (abandon) all files and exit. It prompts for confirmation.
Qally	Quit (abandon) all files and exit without confirmation.

Think of the "Y" in **Qally** as answering "Yes" to the confirmation prompt. Use it with caution!

Display Status Information

Several commands simply display status information. This is similar to the information displayed by **{HELP, Status display}**.

Version or **Ver**	Display VEDIT's version number.
Name_Dir or **ND**	Display the current drive and directory.
Name_Read or **NR**	Display the name of the input (read) file.
Name_Write or **NW**	Display the name of the output (write) file.
Name_File or **NF**	Display the input and output file names.
Mem_Status or **MSTAT**	Display the number of bytes free in the current edit buffer, the number of bytes used, and the total number of bytes in all text registers.
Date	Display the current system date.
Time	Display the current system time.
Reg_Status or **RSTAT**	Display the total number of bytes in all text registers and the number of bytes in each text register.

VEDIT has many other commands that return a numeric value for useful status information. For example, **File_Size( )** returns the size of the current file; **Cur_Line( )** returns the line number of the edit position (cursor). These commands are fully described in Chapters 3 and 4. As described earlier, the on-line calculator at the "COMMAND:" prompt will display their value.

.File_Size or **.FSIZE**	Display the size of the current file.
.Disk_Free or **.DKF**	Display the amount of free disk space.
.Buf_Total or **.BT**	Display the number of edit buffers (files) currently open.

The Config( ) Commands

The **Config( )** command lets you display or change any of VEDIT's configuration parameters including many not included in the **{CONFIG}** menu.

You can display either all 180+ configuration parameters or just one category.

Config or **CF**	Display the current values of all configuration parameters. It is about 180 lines long. Assuming the "-MORE-" function is enabled, the display will pause after each screen full.

The display from the **Config( )** is very similar to the contents of the VEDIT.CFG file described in the VEDIT User's Manual. The name of each configuration parameter begins with a one or two letter category name. (The on-line help lists all categories.) For example, all screen-display oriented parameter names begin with "S_" and all printing oriented parameter names begin with "P_". You can display just the configuration parameters in one category as in the following examples:

Config(s)	Display just the configuration parameters in the screen-display category, i.e. those with a name beginning with "S_".
Config(pg)	Display just the programming oriented configuration parameters, i.e. those with a name beginning with "PG_".

To change the value of a configuration parameter, you must specify its name (upper/lower case doesn't matter and "_" is optional) and its new value.

Config(W_RT_MARG,70)	Set the right margin to column 70.

If you examine a VEDIT.CFG file (created with **{CONFIG, Save config}**), you will notice that the parameter name, e.g. **W_RT_MARG**, can optionally be followed with a descriptive name in quotes, e.g. "**Right margin (*) (0=Window, 16 - 255)**". VEDIT ignores the text between the quotes — it is only intended to improve readability.

Similarly, the **Config_String( )** command lets you display all configuration strings or change one.

Config_String or **CFS**	Display the current values of all configuration strings.
Config_String(PR_DEF,"LPT2")	Change the "Default" printer to be the 2nd parallel port.

The **Config_Tab( )** command displays the current tab stops or lets you set new ones. It is very similar to **{CONFIG, Tab stops}**. You can specify either uniform tab stops or up to 33 explicit tab stops.

Config_Tab or **CFT**	Display the current tab stops.
CFT(10)	Set uniform tab stop at every 10 columns.
CFT(7 15 24 64)	Set tab stop at columns 7, 15, 24 and 64.

The tab stops can be separated from each other with commas or a space. It is the only command that allows numeric arguments to be separated by just a space.

Edit Buffer Dependent Configuration Parameters

For maximum flexibility when editing multiple files, VEDIT maintains a separate set of Tab stops and selected configuration parameters for ***each edit buffer***. This lets you, for example, have word wrap enabled for one file being edited, but not for another.

The edit-buffer dependent configuration parameters are identified with a "(*)" in their name. Many of the programming, word processing and file handling parameters are included.

{CONFIG, Config all buffers} controls whether **{CONFIG}** menu changes apply only to the current edit-buffer or all buffers. The **Config** command is even more flexible; you can change up to three different copies of each edit-buffer dependent parameter.

- The "global" copy of the configuration parameter. This copy is used as the initial value for all newly opened edit buffers. It is also the copy saved to disk with **{CONFIG, Save to disk}** or **{CONFIG, Save into VEDIT}**.
- The "local" copy of the configuration parameter in the current edit buffer.
- The additional copies of the configuration parameter in any additional edit buffers that are currently open.

The **Config( )** command without options, changes both the "local" value and the "global" value. This is equivalent to the **{CONFIG}** menu with **{CONFIG, Config all buffers}** disabled.

Config(W_RT_MARG,70) Set the right margin in the current edit buffer and set the "global" value. Other existing edit buffers are not affected.

Use of the "ALL" option changes all copies of the configuration parameter. This is equivalent to the **{CONFIG}** menu with **{CONFIG, Config all buffers}** enabled.

Config(W_RT_MARG,70,ALL) Set the right margin in all edit buffers and set the "global" value.

Use of the "LOCAL" option changes only the copy in the current edit buffer. The "global" copy is not changed. This cannot be done with the **{CONFIG}** menu.

Config(W_RT_MARG,70,LOCAL) Set the right margin only in the current edit buffer. Other buffers and the "global" value are not affected.

Moving the Edit Position

Many of the commands operate on the text at the "*edit position*". The edit position corresponds to the cursor position in Visual Mode. Commands exist to move the edit position by character or by line. The number of lines or characters the edit position moves is determined by the numeric argument for the command. Negative numbers mean backward movement, towards the beginning of the edit buffer.

Begin_Of_File or **BOF** Move the edit position to the beginning of the file. Equivalent to **{GOTO, Beginning of file}**.

End_Of_File or **EOF** Move the edit position to the end of the file. Equivalent to **{GOTO, End of file}**.

Begin_Of_Line or **BOL** Move the edit position to the beginning of the current line. Similar to **[LINE BEGIN]**.

End_Of_Line or **EOL** Move the edit position to the end of the current line. Similar to **[LINE END]**.

Goto_Line(*n*) Move the edit position to the beginning of line '*n*' in the file.

Goto_Pos(*n*) Move the edit position to the *n*'th character in the file. Counting starts at 0 (zero); therefore, **Goto_Pos(0)** moves to the beginning of the file.

Goto_Pos(500000) Move the edit position to the 500,001 'st character in the file. If the file is not this large, moves to the end of the file.

Line(*m*) Move the edit position forward or backward by '*m*' lines.

Line(6) or **L(6)** Move the edit position forward by 6 lines.

Line(-5) or **L(-5)** Move the edit position backward by 5 lines.

Line(0) or **L(0)** Move the edit position to the beginning of the current line. Equivalent to **BOL( )**.

Char(*m*) Move the edit position forward or backward by '*m*' characters. Remember that with DOS/Windows files, the "newline" is usually two characters - **<CR>** and **<LF>**.

Char(90) or **C(90)** Move the edit position forward by 90 characters.

Char(-2) or **C(-2)** Move the edit position backward by 2 characters.

The commands that alter the text all operate from the current edit position. The search and replace commands normally start their search at the edit position. You can display the lines in the region of the edit position with the **Type( )** command:

Type(*m*)	Display (type) the following or previous '*m*' lines of the file.
Type(10) or **T(10)**	Display the following 10 lines of the file.
Type(-4) or **T(-4)**	Display the previous 4 lines of the file.
T(0) T	Display the current line regardless of where the edit position is on it.

Alter Commands

The basic alter commands let you delete a specified number of characters or lines. Variations of these commands, described in Chapter 3 (Programming Guide) let you delete blocks of text.

Del_Line(*m*)	Delete the following or previous '*m*' lines of text.
Del_Line(6) or **DL(6)**	Delete from the edit position up to and including the 6th "newline".
Del_Line(-4) or **DL(-4)**	Delete all characters (if any) on the current line up to the edit position and the previous 4 lines.
DL(0) DL	Delete the current line regardless of where the edit position is on it.
Del_Char(*m*)	Delete the following or previous '*m*' characters. Remember that with DOS/Windows files, the "newline" is usually two characters.
Del_Char(100) or **DC(100)**	Delete the following 100 characters.
Del_Char(-2) or **DC(-2)**	Delete the previous 2 characters.

The basic insertion commands are **Ins_Text( )** and **Ins_Char( )**:

Ins_Text("*text*")	Insert the text '*text*' at the edit position and advance the edit position.
Ins_Text("Help ")	Insert the word "Help " at the edit position.

The "COUNT" option lets you insert multiple copies.

Ins_Text(".",COUNT,80)	Insert eighty periods at the edit position.
Ins_Char(*n*)	Insert the single character with ASCII value '*n*' at the edit position and advance the edit position.
Ins_Char(10)	Insert a line-feed (<**LF**>) at the edit position.
Ins_Char(205)	Insert a graphics character at the edit position.

The "OVERWRITE" option overstrikes the current character with a new one:

Ins_Char('A',OVERWRITE)	Overstrike (change) the current character to "A".

Search and Replace

This topic describes the search and replace commands **Search( )** and **Replace( )** which correspond to the functions **{SEARCH, Search}** and **{SEARCH, Replace}**.

Searching and Search Options

The text to search for is specified with a "*search string*". Usually the search string consists of the exact characters you want to locate. For example:

Search("today") Search for the next occurrence of "today".

Like any string argument, the *search string* must be enclosed in *delimiters* that are not part of the search string.

If the search is successful, the "*edit position*" is placed at the first character of the matched text, e.g. at the "t" of "today". If not, the command gives the error message "CANNOT FIND "*string*". (Search errors can also be suppressed.)

Sometimes it is preferable to have the edit position placed past the matched text, e.g. immediately after the "y" of "today". For example, this is convenient when a following block operation needs to include the matched text. This is easily done with the "ADVANCE" option.

Search("today",ADVANCE) Search for the next occurrence of "today"; if found, place the edit position past the matched text.

You can directly search for the '*n*'th occurrence with the command form **Search("*string*", COUNT, *n*)**. For example:

Search("today",COUNT,7) Search for the 7th occurrence of "today".

All searches are normally forwards, toward the end of the file. However, you can also search backwards toward the beginning of the file by using the command option "REVERSE". For example:

Search("house",REVERSE) Search backward for the nearest occurrence of "house".

When searching, VEDIT normally equates upper and lower case letters. However, when needed, the command option "CASE" can be used to distinguish between upper and lower case letters.

Search("house",CASE) Make a distinction between upper and lower case letters. This search will not match "House".

Sometimes you want to search for a distinct "word" that is separated from other characters with spaces or other separators. For example, you might want to search for the word "and", but not match "sand", "Anderson" or other words that contain "and". For this use the "WORD" option:

Search("and",WORD) Restrict the search to distinct words. This search will not match "band".

The search string can include any desired pattern matching codes, just like **[SEARCH]**. For example: (The "|" is the keyboard character above "\".)

Search("|D|D") Search for two consecutive digits using Pattern matching.

Regular Expressions can also be used when the "REGEXP" option is specified:

Search("[A-Z][a-z]*",REGEXP) Search for a capitalized word using Regular expressions.

NOTE: The configuration parameters **{CONFIG, Search, Default case-sensitive option}** and **{CONFIG, Search, Default search mode}** have no effect on the **Search( )** and **Replace( )** commands.

The **[SEARCH]** function saves the current search string so that it can be reused by **[SEARCH AGAIN]**. You can do the same thing in Command Mode by using the command option "SET":

Search("house",SET) Perform the search and save the search string as the default search string.

Following the example above, **[SEARCH AGAIN]** would search for the string "house". There is an equivalent to **[SEARCH AGAIN]** in Command Mode:

Search( ) Search again for the next occurrence of the default search string.

One difference between a "search" and a "search again" is that the "search" begins at the current edit position, while the "search again" begins with the character following the current edit position. Otherwise, a "search again" would tend to match the same text over and over again.

Replacing

The **Replace( )** command is used to search for text and replace it with new text.

Replace("today","not today")

Replace the next occurrence of "today" with "not today".

All of the options of the **Search( )** command apply. For example:

Replace("today","not today",COUNT,4)

Replace the next four occurrences of "today" with "not today"

Replace("today","not today",REVERSE)

Search backwards for "today" and replace it with "not today"

A common use of **Replace( )** is to replace all occurrences of a word (perhaps a misspelled one) with another word(s). You could use the command option "COUNT" with a large number, but it is preferable to use the command option "ALL":

Replace("today","not today",WORD+ALL)

Replace all occurrences of the word "today" with "not today".

Another use for **Replace( )** is to delete all occurrences of some particular text. For example, the command to find and delete all occurrences of the word "junk" is:

Replace("junk","",WORD+ALL)

Search and delete all occurrences of "junk", i.e. "junk" is being replaced with nothing.

Unsuccessful Search and Replace

An unsuccessful search normally gives the error message CANNOT FIND "*string*". When the "ALL" option is used with the **Search( )** and **Replace( )** commands, the error is only given if no occurrences were found.

The error message can be suppressed with the "NOERR" and "ERRBREAK" options. This is fully described in Chapter 3 (Programming Guide). The "NOERR" option must be used with care to avoid "infinite" loops inside command macros that use looping. (You can break out of an infinite loop by pressing **[CANCEL]**, **<Ctrl-C>** or **<Ctrl-Break>**.)

When a search is unsuccessful, **Config(SR_RES_POS, *n*)** controls where the edit position will be placed. This is identical to **{CONFIG, Search, Restore edit position on error}**.

0 The edit position is only restored if it is still in memory. No file buffering is performed.

1 The edit position is always restored.

2 The edit position is left at the end of the file. In case of a backwards (REVERSE) search, it is left at the beginning of the file.

When searching large files, it is not always desirable to restore the edit position following an unsuccessful search because it can take a significant amount of time. Using the option "NORESTORE" with **Search( )** or **Replace( )** overrides the setting of **Config(SR_RES_POS, *n*)** and does not restore the edit position; it is left at the end (beginning) of the file.

Search("house",NORESTORE) If the search is unsuccessful, don't restore the edit position. This saves time when searching in very large files.

File Editing Commands

Opening Files for Editing

One or more files can be opened at a time with the **File_Open( )** command. If you specify a filespec with the wildcard characters "*" and "?", **File_Open( )** will open all of the files (when possible).

File_Open("myfile.txt") Open the file "myfile.txt" for editing.

File_Open("c:\jobs\acme*.c") Open all files in the "c:\jobs\acme" directory that have a filename extension of ".c".

Long Filenames

Long filenames with embedded spaces or commas must be enclosed in double-quotes; if multiple files are specified, or the "**-a**", "**-l**" and "**-t**" options are used, the entire string must be enclosed in other delimiters, typically single-quotes.

File_Open('"a long filename.txt","file2.abc"')

Open the two files "a long filename.txt" and "file2.abc" for editing. Notice how single and double-quotes are used to support long filenames with spaces.

File_Open('"*.c", "*.h"') Open all files in the current directory that have a filename extension of ".c" or ".h".

File_Open( ) also supports the "**-a**" and "**-l**" options available when you invoke VEDIT.

The "**-a**" option lets you specify separate input and output files. For example, if you want to edit the text in the file "infile.txt" and in the process create a new file with the name "outfile.txt", you could use the following command:

File_Open('"infile.txt" -a "outfile.txt"')

This is equivalent to the command sequence **File_Open("infile.txt") File_Save_As("outfile.txt").**

The "**-l**" option lets editing begin at the specified line number.

File_Open('"myfile.txt" -l125') Open "myfile.txt" and begin editing at line number 125.

NOTE: **File_Open()** does not automatically create a window for each file opened as does **{FILE, Open}**. Buffers (files) and windows are independent of each other, especially in the command mode.

The command option "ATTACH" causes a new window to be automatically created for each new buffer, similar to **{FILE, Open}**. (This assumes that **{CONFIG, Screen display, Auto-create windows for buffers}** is enabled.)

In practice, the "ATTACH" option is rarely used. Windows are not needed for files edited entirely with a command macro. The first time the buffer (file) is displayed in Visual Mode, a window will be auto-created for it.

Closing Files

Whereas **{FILE, Close}** closes both a file and its buffer, when you close a file in Command Mode, you can either remain in the current buffer or also close the buffer.

File_Close( )	Save and close the current file, and remain in the current edit buffer. If the current buffer contains text but has no assigned filename, you are prompted for one.
File_Quit( ) **Buf_Empty( )**	Empty the current edit buffer without closing it. Quit (abandon) any text or file in the buffer. Requests confirmation if the buffer has been altered. **File_Quit( )** and **Buf_Empty( )** are two names for the same command.
File_Quit(OK)	Skip the confirmation prompt.
Buf_Close( )	Save and close the current file and also close the edit buffer. If the buffer contains text but has no assigned filename, you are prompted for one. Switches to one of the remaining buffers. The main buffer "1" cannot be closed - only the file will be saved and closed.
Buf_Quit( )	Quit (abandon) any file in the buffer and also close the edit buffer. Switches to one of the remaining buffers. The main buffer "1" cannot be closed - only the text/file will be abandoned. Requests confirmation if the buffer has been altered.
Buf_Quit(OK)	Skip the confirmation prompt.

File_Close( ) and **File_Quit( )** are useful when you are finished editing one file and want to edit a new file in the current buffer.

You can perform the equivalent of **{FILE, Open (More), Same buffer}** with a combination of the **File_Close( )** and **File_Open( )** commands:

File_Close( ) File_Open("*filename*")

NOTE: **File_Close( )** is not the same as **{FILE, Close}**.

Save File and Continue Editing

The **File_Save( )** command performs the equivalent of the **{FILE, Save}** function. When you are spending a lot of time editing a file, it is a good habit to routinely save the file on disk and then continue editing it. Otherwise, all of

your edit changes will be lost should a power or hardware failure occur. This also protects you from your own mistakes.

File_Save( )	Save file to disk for continued editing.
File_Save(ALL)	Save all files to disk for continued editing. Equivalent to **{FILE, Save all}**

The **File_Save( )** command does not affect your current editing position, the text markers or the text registers.

NOTE: **File_Save( )** saves the file to disk and begins editing it again. Therefore, if a **File_Save( )** is later followed by a **File_Close( ) - Abandon**, you will only abandon those changes made after the **File_Save( )** command. Those changes made before the **File_Save( )** will have already been saved on disk.

You may find the **File_Save(BEGIN)** command useful; it has no equivalent in Visual Mode:

File_Save(BEGIN)	Save the file to disk and continue editing from the beginning of the file. It is equivalent to, but faster than **File_Save( ) BOF( )**.

SUGGESTION: If you are near the end of a very large file and need to begin editing from the beginning again, it is often faster to use the command **File_Save(BEGIN)** instead of **BOF( )**. This has the added benefit of saving the current file. It then restarts the editing at the beginning of the file.

Directory Display

The **Directory( )** command displays the filenames in any desired directory. Drive specifiers and the "wildcard characters" "**?**" and "*****" can be used. Some examples are:

Dir( )	Display the current directory.
Dir("a:")	Display the directory of drive "A".
Dir("*.asm")	Display (list) all ".asm" files in the current directory.

Deleting (Erasing) Files

Files can be deleted with the **File_Delete( )** command. In case the disk becomes full and VEDIT gives you a "NO DISK SPACE" or "NO DIRECTORY SPACE" error, you can first use the **Directory( )** command to determine what files can be deleted. Then use **File_Delete( )** to delete the unneeded files. Some examples are:

FDEL("oldfile.txt")	Delete one file.
FDEL("*.bak")	Delete all ".bak" files.

Before deleting the files, **File_Delete( )** displays a list of the files to be deleted and requests confirmation.

If you prefer, you can also delete files by shelling out to DOS with the **System( )** command (equivalent to **{MISC, DOS Shell}**) and then using DOS commands to delete the files.

NOTE: When deleting files, do not delete any "*.r*$$" or "*.r*R$" files while VEDIT is running! These are temporary files that VEDIT is using. Deleting these files will result in lost text.

Pathnames

Any filename may optionally include a standard "*pathname*" to any directory. For example:

File_Open("\business\letter.txt") Edit the file "letter.txt" in the directory "\business".

Dir("\business\") List all files in the directory "\business". Note that you must include the second "\".

The **Directory( )** command lists both files and directories — directories are indicated with a "\" following the name.

"\" and "/" Windows/DOS use the backward-slash "\" in pathnames, while UNIX and QNX use the forward-slash "/". If desired, you can enter all pathnames in VEDIT macros with forward-slashes; this makes it much easier to write macros that will work under all operating systems.

Changing Current Drive / Directory

If you are constantly accessing files in another directory, it may be easier to change to that directory with the **Chdir( )** command. That way you won't have to specify the pathname each time a file is referenced. You can also change to another drive.

Chdir("\business") Change the default (current) directory to be "\business".

Chdir("D:") Change to drive D:.

Chdir("D:\BUSINESS") Change to drive D: and the directory "BUSINESS".

You can verify which drive and directory are current with the command:

Name_Dir Display the current drive and directory.

You can also display the current drive and directory with **Chdir( )** without any arguments. However, **Name_Dir( )** has several display options that **Chdir( )** does not have.

Text Registers

The 100+ text registers serve two primary purposes. One is for "cut and paste" operations, where they temporarily hold a block of text. The second is to hold "*command macros*" which are sequences of macro language commands. In both cases, the registers are holding textual material; only the manner in which the text is used is different.

Text Register Commands

The text registers can be loaded directly from disk or saved to disk and their contents can be displayed on the screen or printed.

The command option "APPEND" indicates appending to the existing contents (if any) of the register. The command option "INSERT" indicates inserting at the beginning of the existing contents of the register.

Lines of text are copied to a register with the **Reg_Copy( )** command:

Reg_Copy(5,35)	Copy the next 35 lines to register "5".
Reg_Copy(4,-6,APPEND)	Append previous 6 lines to register "4".
Reg_Copy(5,9,INSERT)	Insert 9 lines at the beginning of register "5".

A text register is emptied with the **Reg_Empty( )** command:

Reg_Empty(2)	Empty register "2".

The **Reg_Ins( )** command inserts the contents of a register at the edit position:

Reg_Ins(2)	Insert register "2" at the edit position.

The **Reg_Save( )** command saves the contents of a text register to a file. A block of text may therefore be copied (or appended) to a text register, which is then saved as a new file.

Reg_Save(10,"\data\file.txt")

Save contents of register "10" in "file.txt" in directory "\data".

The **Reg_Load( )** command loads a register with a file from disk. This is often used to load command macros from disk.

Reg_Load(11,"\data\file.txt")

Load register "11" with "file.txt" in directory "\data".

Reg_Load(11,"file.txt",APPEND)

Append "file.txt" to the contents (if any) of register "11".

The contents of a text register can be displayed with the **Reg_Type( )** command:

Reg_Type(0)	Type out (display) contents of register "0".

Reg_Type( ) may expand control character, depending upon the current display mode. Since this is not always desired, the command form **Reg_Type(*r*,0)** is provided, which does not expand any control characters.

Reg_Type(9,0)	Type out (dump) the contents of register "9" without expanding any control characters.

The **Reg_Print( )** command prints the contents of a text register. One application is to print a file after first loading it into a text register.

Reg_Print(100)	Print contents of register "100".

A string constant can be placed into a text register with the **Reg_Set( )** command:

Reg_Set(10,"abcde")	The text string "abcde" is placed into register "10".
Reg_Set(10,"The end",APPEND)	The text string "The end" is appended to register "10".

Text can be directly copied from one text register to another as the following example illustrates:

Reg_Set(10,@11)	Copy the contents of register "11" to register "10".
Reg_Set(10,@11,APPEND)	Append the contents of register "11" to register "10".

The **Reg_Status( )** command displays the total number of bytes in all text registers followed by the number of bytes in each text register.

Using Text Registers in Filenames

The contents of a text register can be used as the entire filename or as a portion of a filename with any command that takes a filename argument. This makes it easy to have "variable" filenames. A register can also specify the DOS command to be executed with the **System( )** command.

Reg_Set(10,"myfile.txt") **File_Open(@10)**	Open the file "myfile.txt". Register 10 contains the entire filename.
Reg_Set(10,".txt") **File_Open("myfile|@(10)")**	Open the file "myfile.txt". Register 10 contains just the filename extension ".txt".

Besides the **Reg_Set( )** command, there are more useful ways to store the desired filename in a text register. The topic "Interactive Input and Output" in Chapter 4 (Programming Guide) describes how to interactively enter a filename from the keyboard.

System(@10)	The contents of register "10" are used as a DOS command to be executed.

Reg_Set(10,"txt") **System("dir *.|@(10)")**	Use the DOS "dir" command to display a directory of all files with a filename extension of ".txt".

Text Register Usage

With over 100 text registers available, it is easy to forget what each register contains. Several text registers are also reserved for special purposes. We recommend the following organizational scheme for using registers:

0	The "scratchpad" or default "cut and paste" register in Visual Mode.
1 - 9	Used as additional "cut and paste" registers from Visual Mode.
10 - 99	Used to hold command macros or as string variables in command macros.
100	Used by any auto-execution macro specified with the "**-x**" invocation option. It is also the default register for **{MISC, Load and execute macro}**. It should be reserved for the "main" macro that is running.
101 - 127	Reserved for special purposes. See Appendix G or the on-line help topic "USAGE" for a detailed description of these registers.

To protect users from unintentionally overwriting text registers, the Visual Mode can only access registers 0 through 100. The **Reg_Prot( )** command (see Chapter 4) can also be used by a command macro to write-protect the text registers it uses.

Intermediate Commands

Printing Text

Text can be printed with the **Print( )** command which takes a numeric argument identical to **Type( )** specifying how many lines are to be printed.

Print(*m*)	Print the following or previous '*m*' lines of text.
Print(40)	Print the following 40 lines.
BOF( ) Print(ALL)	Print entire edit buffer (file).

Any control characters in the text are expanded according to the current print mode set with **{CONFIG, Printer, Print mode}.** Typically, all control characters, except for <**Tab**> are sent as-is to the printer. Therefore, embedded "Escape sequences" can be used to control printer functions such as font changes and underlining. The <**Tab**> character is expanded to spaces according to the current tab stops.

When text is printed, lines are offset from the left edge of the paper by a selectable "*printer margin*". You can select the desired margins and other print configuration parameters with the **{PRINT, Config}** sub-menu or with the appropriate **Config( )** commands.

Config(P_TOP_MARG,2)	Set the top margin for a printed page to 2 lines.

The command option **Print(*n*,RAW)** prints in "raw" mode without adding any printer margins or expanding any control characters. It is ideal for printing files which have already been formatted for a printer, such as word processing file which was "printed" to a file, or the ".LST" files created by a compiler.

BOF( ) Print(ALL,RAW)	Print the entire file in "raw" mode without adding margins; send all characters as-is to the printer.

The **Print_Eject( )** command advances the printer to the next page. To keep VEDIT synchronized with the paper position in the printer, you should perform page ejects with **Print_Eject( )** and ***NOT*** use the printer's "Form Feed" or "Paper Advance" functions.

If VEDIT does get out of sync with your printer, press the "Form Feed" button on your printer and, if necessary, manually align the print head with the top of a page. Then issue the special **Print_Eject(0)** command.

Print_Eject( )	Eject - advance printer to next page.
Print_Eject(0)	Synchronize VEDIT with the printer at the top of a new page.

When writing a command macro to print labels, it helps to set the "Paper Length" to the size of the labels — typically nine lines. Then use **Print_Eject( )** to advance to the next label.

The supplied macro **`print.vdm`** prints the entire file and additionally prints the filename, date and page number at the top of each page. Examining this macro is instructive for understanding the print commands better.

Print_Finish( ) finishes and closes the current print job. Assuming **{CONFIG, Printer, Page eject on Finish/Eject}** is enabled, it first sends a page eject to the printer. If **{CONFIG, Printer, Enable printer setup strings}** is set to "2" or "3", it then sends the "Printer Finish string". Finally, it closes the print job. On most systems that spool the printer, nothing will print until this command closes the print-job. This is especially true with network printers.

When printing to a file, **Print_Finish( )** closes the file, saving it on disk.

Entering Control Characters

You can include control character in a string argument such as a search string. Since many control characters perform editing operations, first press **[ENTER CTRL]** (<**Ctrl-Q**>) and then the control character, e.g. <**Ctrl-X**>.

The following examples insert a <**Ctrl-H**> into the text and search for a <**Ctrl-H**>: (Assumes **[ENTER CTRL]** is <**Ctrl-Q**>.)

Ins_Text("<Ctrl-Q><Ctrl-H>")	Insert a <**Ctrl-H**>.
Search("<Ctrl-Q><Ctrl-H>")	Search for a <**Ctrl-H**>.

Because the IBM PC keyboard cannot produce a Null (value 00) character, it is a little more difficult to search and insert these characters. The pattern matching code "|**000**" and the regular expression "**d000**" permit searching for the Null character.

Search("|000")	Search for the Null (value 00) character.
Ins_Char(0)	Insert the Null character.

NOTE: You may find it easier to search for control characters by specifying their decimal or hexadecimal value. Both pattern matching and regular expressions support this. For example, "|003" is the pattern matching code to search for <Ctrl-C>.

DOS/Windows text files normally have lines ending with a "newline" consisting of the two characters "carriage return" and "line feed" — a <**CR**><**LF**> pair. However, when files are transferred from mainframe computers, the lines often end in a <**CR**> without the <**LF**>. These lone <**CR**>'s must be changed to <**CR**><**LF**> pairs. The easiest way to specify the control characters is as decimal values.

The command to change all lone <**CR**>'s to <**CR**> <**LF**> pairs is:

```
Begin_Of_File( )
Replace("|013","|013|010")
```

Re-routing Console Output

Any command macro console output, which normally goes to the screen, can be re-routed to the printer, the edit buffer, to the OS (DOS), a file, or a text register. Such re-routing is in effect until the next "COMMAND:" prompt or until re-routing is canceled. This capability is tremendously powerful and permits operations which would otherwise be very difficult or even impossible.

Out_Ins( )	Re-route to the edit buffer.
Out_OS( )	Re-route directly to the OS, bypassing the normal window handlers.
Out_File("*file*")	Re-route to a file.
Out_Print( )	Re-route to the printer.
Out_Reg(*r*)	Re-route to text register '*r*'.

The command **Out_Print(0)** or **Out_Print(CLEAR)** cancels the re-routing and allows normal console output. (Similar for the other commands.)

NOTE: The following command sequences are shown on one line because the "COMMAND:" prompt cancels any re-routing. In command macros executed from a text register, each command could be on its own line.

Out_Print( ) Directory("B:") Out_Print(CLEAR)

Print the directory of drive B.

Out_Print( ) Message("Hello") Out_Print(CLEAR)

Print the word "Hello".

Out_Ins( ) re-routes console output to the edit buffer, inserting it at the edit position. **Out_Ins(0)** or **Out_Ins(CLEAR)** cancels the re-routing. Some examples to try are:

Out_Ins( ) Date( ) Time( ) Out_Ins(CLEAR)

Insert current date and time into the edit buffer (file).

Out_Ins( ) Dir( ) Out_Ins(CLEAR)

Insert current directory into the edit buffer.

Out_Reg( ) can be used to place the name of the output file into a text register:

Out_Reg(20) Name_Write(NOMSG+NOCR) Out_Reg(0)

Place name of output file into text register 20.

The re-routing commands can be nested. Therefore, following the commands **Out_Print( ) ... Out_Ins( ) ... Out_Ins(0)**, console output will be re-routed to the printer.

The topic "Input Commands" in Chapter 3 describes how keyboard input can be redirected from a file with the **Redirect_Input( )** command.

Multiple File Editing

NOTE: In practice, it is much easier to handle multiple file editing by using the **{FILE}** and **{WINDOW}** menus. This topic describes details about multiple file editing as it relates to command macros.

VEDIT has 99 available edit buffers, each of which can have one file open for editing. The edit buffers are always in one of three possible states:

- Closed. We also say that a closed edit buffer is "*available*" or "*free*".
- Open without a file. The edit buffer may or may not contain text. If the buffer contains no text, we say that it is "*empty*".
- Open with a file open. The edit buffer is being used to edit a file.

Once an edit buffer is opened, it remains open until you explicitly close it with the **Buf_Close( )** or **Buf_Quit( )** commands or from Visual Mode. One edit buffer is always open; if only one edit buffer is open and you attempt to close it, it will close any file, but the buffer will remain open.

At any time, only one edit buffer is the "*current*" or "*active*" buffer. The name of the current edit buffer is always displayed on the status line: "#1" - "#32". All editing operations are performed on the "current" buffer.

The **Buf_Switch( )** command is used to switch to another edit buffer; if the selected buffer is not already open, it will be opened as an "empty" buffer. It is equivalent to **{FILE, Buffer switch}**.

Buf_Switch(*b*) Switch to edit buffer '*b*', opening it if necessary.

The **File_Open( )** command opens edit buffers as necessary to edit the specified files. Although it can then be difficult to predict in which buffer each file will be opened, this is usually not important.

If it is important to open specific files in specific buffers, you can use the **Buf_Switch( )** and **File_Open( )** commands together, opening one file at a time. The following example assumes that edit buffers 2 and 3 are either closed or empty:

```
Buf_Switch(2)
File_Open("file2.abc")
Buf_Switch(3)
File_Open("file3.abc")
```

Open the file "file2.abc" in edit buffer 2 and the file "file3.abc" in buffer 3.

Using Edit Buffers as Text Registers

Many text register operations can also be performed on edit buffers:

- You can execute the contents of an edit buffer as a command macro with **{MISC, Execute macro}** or the equivalent **Call( )** command. This lets you write macros in a window (buffer) and directly execute it, without first copying it to a text register. However, you cannot execute the macro in the current buffer; you must first switch to another buffer.

- You can insert the contents of another edit buffer into the current buffer with **{BLOCK, Insert register}** or the equivalent **Reg_Ins( )** command.
- You can save the contents of any edit buffer to disk with the **Reg_Save( )** command.

You cannot change the contents of an edit buffer except when it is the "current" edit buffer.

Therefore, you can perform **{BLOCK, Insert Register}** with edit buffers, but not **{BLOCK, Copy/Move to Register}**. This limitation on altering edit buffers helps prevent you from inadvertently altering the corresponding file.

To perform a text register operation on edit buffer '*b*', use the register name '*b*+BUFFER'.

Reg_Ins(2+BUFFER) Insert the contents of edit buffer "2" into the current edit buffer at the edit position.

TECHNICAL: In general, text register operations with edit buffers should only be performed with edit buffers that either have no file open or are reasonably small. In particular, if the open file is large, only the portion currently in memory will be used.

When editing multiple large files, VEDIT may perform file buffering when switching from one edit buffer to another — part of the edit buffer you are leaving will be written to disk to free more memory space. This auto-buffering on the **Buf_Switch( )** command can be disabled with the command form **Buf_Switch(*r*, LOCAL)**. (See **Buf_Switch( )** in Chapter 4 for more details.)

Moving Text Between Edit Buffers

You can copy or move text from one edit buffer to another by using an intermediate text register.

➤ **Copy text from one edit buffer to another in command macros:**

1. Make the edit buffer containing the text active with the **Buf_Switch( )** command, if necessary.
2. Copy the text from the edit buffer to an available text register with the **Reg_Copy( )** command.
3. Give the **Buf_Switch( )** command for the edit buffer which is to receive the text.
4. Place the edit position where the text is to be inserted. The **Line( )**, **Search( )** and **Goto_Pos( )** commands may be useful here.
5. Insert the text with the **Reg_Ins( )** command.

If the text being copied is already in a disk file, you may want to use the **Type_File( )** command to locate it and then use the **Ins_File( )** command to insert it directly from the disk file.

Window Commands

The **{WINDOW}** menu manages windows from Visual Mode. Command Mode permits windows to be managed with additional flexibility. The commands to create, delete, switch windows and change color are covered here. Additional commands related to window management are covered in Chapter 4 (Programming Guide).

Windows are created by either splitting an existing window in two, or by creating an overlapping window. A window may be as small as one line and/or 10 columns (not including the border drawn around the window).

Win_Split(*w,n,*BOTTOM)

Split the current window to create window '*w*' of '*n*' lines at the bottom of the current window. The reserved words "TOP", "BOTTOM", "RIGHT" and "LEFT" specify the location.

Win_Create(*w,l,c,nl,nc*)

Create the overlapping window '*w*' and switch to it. The window's top-left corner origin is at line '*l*' and column '*c*'. It's size is '*nl*' text lines and '*nc*' columns, not including the borders.

Win_Create(*w,y-org,x-org,y-size,x-size*,PIXEL)

Create the overlapping window '*w*' and switch to it. The window's top-left corner origin is at pixel position (*x-org*,*y-org*). It's width is '*x-size*' pixels and it's height is '*y-size*' pixels, including the borders. (Windows version only.)

Win_Create(*w*,0,0,0,0,PIXEL)

Create window '*w*' as a "full-size" overlapping window and switch to it. (Windows version only.)

Additional windows can be numbered from "2" to "35" or can be named with any ASCII character '$' or greater (value 36 to 127). We suggest that editing windows be numbered while special purpose windows be named. This prevents conflicts with the **{WINDOW}** split functions which always create numbered windows. You should avoid naming windows '0' through '9' since they will be difficult to distinguish from numbered windows.

Win_Split(2,30,RIGHT)	Create window numbered "2" of 30 columns at the right of the current window.
Win_Split('H',2,TOP)	Create window named "H" of 2 lines at the top of the current window.

Three window numbers/names are predefined:

1	Window 1 is referred to as the "*main window*" because it is auto-created upon startup and by **Screen_Init( )**. Therefore, it usually exists, but can be deleted if desired.
STATLINE	The status line is a one line window that always exists and cannot be deleted. You can switch to it with **Win_Switch( )**.

**$** If you create a window named '$' it becomes the dedicated Command Mode window. This was described earlier in this chapter.

Like most numeric arguments, the window number/name can be a numeric expression. However, as a special case, a window name may be specified with or without single quotes.

Win_Split('$',5,BOTTOM) Create the special Command Mode window of 5 lines at the bottom of the current window.

Win_Split($,5,BOTTOM) Alternate form — the single quotes can be left off.

The special window size of "0" (zero) indicates that the window is to be split in half. This has the added benefit that the new window will remain half the size of the current window even if the screen size changes, e.g. with **{VIEW, VGA/EGA toggle}**. **{WINDOW, Vertical Split}** uses this feature.

Win_Split(3,0,BOTTOM) Create window "3" by splitting the current window horizontally in half. For example, subsequently switching from 25 screen lines to 50 lines will double the size of the window.

The new window will normally have borders as determined by the setting of **{CONFIG, Display options, Window borders}**. The description of **Win_Split( )** in Chapter 4 (Command Reference) describes how to select the desired type of borders.

Once a window is created, you can switch to it with the command:

Win_Switch('*w*') Switch to window '*w*'.

Win_Switch('H') Switch to window "H".

WS(1) Switch to the default window "1".

Following a switch to a window, any command macro console output will be displayed in the new window. However, a simple **Win_Switch( )** command does not switch to a different edit buffer. When you enter Visual Mode, the current buffer will be displayed in its "*attached*" window, not the window you just switched to.

The command form **Win_Switch(*w*, ATTACH)** performs the equivalent of the **{WINDOW, Switch}** function. It also switches to the edit buffer currently attached to the selected window. If the buffer has two or more attached windows (e.g. when a file is displayed in two or more windows), it makes the selected window the primary editing window.

Win_Switch(2,ATTACH) Switch to window "2". If window "2" is currently being used to display a buffer/file, it also switches to that edit buffer.

Windows can be deleted with **Win_Delete( )** which is similar to the **{WINDOW, Delete}** function. Remember that this only deletes the window, it does not affect files or edit buffers.

Win_Delete(*w*)	Delete window '*w*'.
Win_Delete($)	Delete the Command Mode window.
Win_Delete( )	Delete the current window.

Some additional window/screen commands are:

Screen_Init( )	Initialize the screen by deleting all windows; only the default window "1" remains. Reset to the configured color attributes.
Screen_Init(ALL)	Initialize the screen by deleting all windows, even the default window "1".
Win_Zoom( )	Zoom the current window to full screen.
Win_Zoom(CLEAR)	De-zoom the window.

"Reserved" Windows

The command **Win_Reserved( )** creates a "reserved" window at the top or bottom of the screen that cannot be overlapped. This is useful for a help line or other window that must always be visible. With a reserved window, the screen is effectively smaller for all other windows; even zooming a window will not cover the reserved window. Cascading and tiling does not include a reserved window.

Win_Reserved(*w*,*n*,BOTTOM)

Create the reserved window '*w*' of '*n*' lines at the bottom of the screen. All other windows are resized to account for these reserved screen lines.

The Command Mode window is typically created as a reserved window to ensure that it is always visible.

Windows and Edit Buffers

Although windows are typically used to display edit buffers (files), windows and edit buffers operate independently of each other, especially in command macros. Command macros often create windows that display help information, menus or other text.

When an edit buffer is displayed in a Visual Mode, it selects a window according to these rules:

Rule 1: If the edit buffer is already "attached" to a window, and the window still exists, it selects this window.

Rule 2: If a window by the same ID number as the edit buffer exists (and is not already attached to another edit buffer), it selects and attaches itself to this window. According to Rule 1, it will continue to use this window as long as possible.

Rule 3: Assuming **{CONFIG, Display options, Auto-create window style}** is enabled, it creates a new full-screen overlapping window and attaches itself to this window.

Otherwise, it selects and attaches itself to the current window. However, if the current window is the special "$" Command Mode window, it selects the main "1" window instead.

When **{CONFIG, Display options, Auto-create window style}** is enabled (default), each edit buffer (file) is displayed in its own window. Initially these windows are full screen and overlap each other.

When disabled, several buffers can be attached to the same window; this happens when you edit several files without splitting the screen into windows.

In many cases edit buffer "1" will be displayed in window "1", buffer "2" in window "2" and so on. However this is not always true. Fortunately you rarely need to know which window number a buffer is attached to. When needed, command macros give you complete control over opening files in specific edit buffer and displaying them in specific windows.

For example, to simultaneously edit the files "report.txt", "table.txt" and "index.txt" in separate custom sized windows you could give the following commands (starting with invoking VEDIT):

vpw report.txt	Invoke VEDIT with first file.
[VISUAL EXIT]	Go into Command Mode.
Win_Split(2,8,BOTTOM)	Create window "2" of 8 lines
Win_Split(3,8,BOTTOM)	Create window "3" of 8 lines
Buf_Switch(2)	Begin editing in buffer "2"
File_Open("table.txt")	Edit the file table.txt
Buf_Switch(3)	Begin editing in buffer "3"
File_Open("index.txt")	Edit the file index.txt
Visual( )	Enter Visual Mode

You can now use either the **{FILE, Buffer switch}** (**<F4>**), **{FILE, Next buffer}** (**<F6>**), **{WINDOW, Switch}** (**<Alt-F5>**) or **{WINDOW, Next window}** (**<Ctrl-F6>**) functions to switch between files (and associated windows).

Advanced Window Commands

By Rule 1 above, an edit buffer is displayed in its "attached" window. The first time the buffer is displayed in Visual Mode, it will select or create a window, and attach itself. The buffer will remain attached to the window until either the window is deleted, or you explicitly detach it.

You can also explicitly attach windows to the current edit buffer.

Win_Attach(*w*)	Attach window '*w*' to the current edit buffer. This detaches the window from any other edit buffer.
Win_Detach(*w*)	Detach window '*w*' from any edit buffer.

You can attach more than one window to an edit buffer; this is how different regions of one edit buffer (file) can be displayed in multiple windows. This is one use of the **Win_Attach( )** command.

When an edit buffer is attached to multiple windows (or multiple windows are attached to an edit buffer; you can look at it either way), the window in which you are editing is the "primary" window while the others are the "secondary" windows. Therefore, by Rule 1 above, when an edit buffer is displayed in Visual Mode, it selects the "primary attached" window for its display.

The command **Win_Switch(*w*, ATTACH)** lets you switch to a "secondary" window and make it the new "primary" window. It is equivalent to the **{WINDOW, Switch}** function.

Win_Switch(*w*,ATTACH)

Switch to window '*w*' and, if it is attached to the current edit buffer, make it the primary window. This also moves the edit position in the edit buffer to the edit position corresponding to this window (which may involve file buffering).

When a window is already attached to an edit buffer, **Win_Attach( )** attaches additional windows as secondary windows. Should the primary window be deleted, one of the secondary windows automatically becomes the primary window.

Chapter 3

Programming Guide

This chapter covers the VEDIT macro programming language in much greater detail. It is organized as follows:

- Explains how "command macros" are stored and run from text registers.
- Details the **Repeat**, **While**, **Do-while** and **For** loops and the **If-then** and **If-then-else** statements which provide flow control.
- Details the extensive numeric capabilities including numeric, logical and relational operators which closely follow the "C" language.
- Covers logical groups of commands that are primarily used within the context of command macros. Advanced topics explain file buffering, running macros on huge data files, and event macros.
- Explains topics related to developing large, complex macros. Covers how macros can be debugged by using breakpoints and tracing.

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. 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", try running some of the examples in this chapter. By all accounts, you will then have written and used real "programs".

We use the term "*command macros*" or just "*macros*" instead of *programs* to refer to sequences of one or more macro language commands. (Some users are needlessly frightened away from the word *program*.)

Some macros are quite short, while others can be quite long and sophisticated, such as the supplied **`compare.vdm`** and **`sort.vdm`** macros. Useful macros do not need to be long — many useful macros consist of less than ten commands and are just a few lines long! The supplied file **`key-mac.lib`** contains dozens of short, useful macros.

Any command macro, no matter how long, can be entered directly from the keyboard while in Command Mode. Many short one line macros are in fact entered in response to the "COMMAND:" prompt — when you press **<Enter>** the macro is executed. However, it would be tedious and error prone to have to type in the entire macro each time you wanted to use it. Therefore, all but the simplest command macros are typically created and stored as files which are then loaded into text registers for execution.

Command Macros

While simple command macros can be implemented as keystroke macros (as described in Chapter 2), most command macros are stored as files and loaded into text registers when needed. Command macros sitting in the text registers can consist of just a single command or hundreds of commands.

There is nothing special about the way command macros are stored in text registers — there is no difference between text registers that contain blocks of text from a file and those that contain command macros. It is up to you to keep track of which registers contain text and which contain commands.

An existing command macro on disk is usually loaded into a text register with the **Reg_Load( )** command.

Reg_Load(10,"splitter.vdm") Load the file **`splitter.vdm`** into text register 10.

New macros are created just like any other file — by entering and editing it in Visual Mode. The macro is then copied to a text register for execution and testing. When done, the macro is saved to disk just like a normal text file.

The commands in a text register are executed with the **Call( )** command.

Call(10) Execute the macro in text register 10, starting at the beginning of the register.

Just like with other programming languages, complex macros should be broken down into several simpler ones. This is analogous to the programming concept of "subroutines". Since a macro can itself contain the **Call( )** command, it can execute other subroutine-like macros.

Each subroutine macro can be stored in its own text register, or multiple macros (subroutines) can be placed into one text register. In the latter case, the command form **Call(*r,* "*label*")** must be used to start execution at the label '*label*', instead of at the beginning of the register. Execution continues until the end of the register is reached, a **Return( )** or **Break_Out( )** command is executed, or an error occurs.

Call(10,"exchange") Execute the macro in text register 10, starting at the label "exchange".

Since a subroutine macro is often in the currently executing text register, the command form **Call("*label*")** can be used to execute the subroutine macro labeled '*label*' in the current register.

Call("exchange") Execute the subroutine macro labeled "exchange" in the current text register.

Command Macros in Visual Mode

Command Mode commands can also be accessed from Visual mode in three ways — via keystroke macros, via the **{MISC, Execute Macro}** function and via the **{USER}** menu. These methods were introduced in the topic "Easy as

1-2-3-4" in Chapter 2 of this manual and in Chapter 5 of the VEDIT User's manual.

Simpler command macros that are repeatedly used from Visual Mode are usually implemented as keystroke macros. Such a keystroke macro must begin with **[VISUAL EXIT]** (<**Ctrl-E**>) to enter Command Mode. However, the keystroke macro will return automatically to Visual Mode and a final "**V**" command is not needed.

When the command macro assigned to a keystroke macro is longer than a few hundred characters, the command macro should be saved as a file. The keystroke macro can then use the **Call_File( )** command to load the command into a text register and run it there. It is a combination of the **Reg_Load( )** and **Call( )** commands.

[VISUAL EXIT] **Call_File(10,"splitter.vdm")**	Load the command macro **`splitter.vdm`** into text register 10 and execute it.

NOTE: The following example command macros would be better handled as keystroke macros. It is only intended to illustrate how **{MISC, Load/execute macro}** can be used.

The following macro duplicates the current line of text 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.

Begin_Of_Line( ) Block_Copy( )	Macro to duplicate a line of text.

The next example macro moves the cursor to the beginning of the next sentence.

Search(".|S") Search("|F")	Macro to move the cursor to the beginning of the next sentence.

To execute a command macro with **{MISC, Execute Macro}** you must first place the commands into a text register you will not be using for other purposes. Command macros are often loaded from disk using the **Reg_Load( )** command.

If you are repeatedly using a command macro (from Visual or Command Mode), it is easy to have VEDIT start up with the macro pre-loaded. For example, to set up the two macros above add the following line to the **`ustartup.vdm`** file:

Reg_Set(10,/Begin_Of_Line( ) Block_Copy(1)/) **Reg_Set(11,/Search(".|S") Search("|F")/)**	Load "duplicate line" macro into register 10 and load "next sentence" macro into register 11.

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

{MISC, Execute macro} - 10	To duplicate a line.
{MISC, Execute macro} - 11	To move cursor to next sentence.

Loading Macros into Text Registers

To execute a command macro saved on disk, it can be loaded into a text register with the **Reg_Load( )** command and then executed with the **Call( )** command. For example, the commands to execute the supplied **`print.vdm`** macro using text register 10 are:

```
Reg_Load(10,"print.vdm")
 Call(10)
```

Call_File( ) is equivalent to the two commands above. It also has the advantage that if the macro file is not found in the current directory, it will search for it in the *VEDIT Home Directory*. It is equivalent to the **{MISC, Load/Execute}** function.

Call_File(10,"print.vdm")	Preferred shortcut for the two commands above.

Some command macros are written to have their subroutine macros in additional text registers. Each register of a multiple register macro could be loaded from a separate disk file, but this would be awkward. It is better to have the main macro set up the additional subroutine registers by using the **Reg_Set(*r*,"*text*")** command. Since the string argument '*text*' can be multiple lines long, an arbitrarily complex subroutine macro can be placed into register '*r*'.

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 **Reg_Set( )** commands.

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.

VEDIT follows the ANSI C convention that any text following "//" to the end of the line is treated as a comment and is ignored during macro execution. Note that the "//" must occur outside of any command — a "//" inside a string argument will not be treated as the beginning of a comment.

```
// This text is a comment and is ignored by VEDIT

Message('A "//" in a string doesn't confuse VEDIT.')
```

Flow Control

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

Search("the")
Type(0) Type( )

This simple sequence of commands could be a macro. 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 that are to be repeated. Any such looping involves flow control — you are controlling the order (flow) in which commands are executed.

In addition to "looping", there is "decision making". The macro tests some condition — if it is TRUE, the macro performs one operation — if it is FALSE, the macro performs a different operation. 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".

Looping and decision making is performed primarily with "*flow control statements*". The **Repeat** statement performs simple looping. The **If-then** and **If-then-else** statements perform decision making. The **While**, **Do-while** and **For** statements perform a convenient combination of looping and decision making.

VEDIT also has the **Goto** statement that jumps to a label. Macros should always be written to minimize use of **Goto** since they tend to make a macro difficult to understand.

Repeat Loop

Although VEDIT has the **While**, **Do-while** and **For** loops of the C programming language, the **Repeat** loop (not in C) is often easier to use. Here is its format:

```
repeat (n) {
    commands
    ...
}
```

Notice that there is an optional space between "**repeat**" and "**(**". Only flow control statements can have this space and we recommend using it. This helps distinguish these statements from commands. As a convention, we don't capitalize the first letter of these statements as we do commands. It is common practice to indent all lines between the "**{**" and "**}**".

A **Repeat** loop repeatedly executes the group of '*commands*' between the braces "**{...}**" a total of '*n*' times. Command execution then continues with any commands following the "**}**". In case '*n*' is zero (0), the '*commands*' are not executed at all.

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

```
repeat (10) { Print(20) Print_Eject( ) Line(20) }
```

The **Print(20)** prints 20 lines of text, **Print_Eject( )** starts a new page and **Line(20)** moves the edit position over the text just printed. Due to the **Repeat** loop, this sequence of commands is executed for a total of 10 times.

A **Repeat** loop may also occur within another **Repeat** loop. This is called a "*nested*" loop.

```
repeat (5) { Type(4) }
```

Display the same four lines over again five times.

```
repeat (3) {
    repeat (5) { Type(4) }
    Line(4)
}
```

Display the same four lines five times, then move to the next four lines and display them five times, and finally, move to the next four lines and display them five times.

It is important to note that if the repeat count '*n*' is zero (0), the '*commands*' are not executed at all. It is also important to note that '*n*' is evaluated only once as the following example illustrates:

```
#1=10
repeat (#1) {
    #1=1000
}
```

This loop will execute 10 times, not 1000 times.

A repeat count of "ALL" makes a **Repeat** loop continuously ("forever" or for "all occurrences") until a "break-out" condition occurs. Therefore, the form for a continuous loop is:

```
repeat (ALL) {
    commands
    ...
}
```

Loop "forever" until a break-out condition occurs.

The "break-out" might be the user pressing **[CANCEL]** (<**Ctrl-C**>) or, more likely, a condition such as a **Search( )** command not succeeding or an explicit break-out command such as **Break** or **Return( )**.

The following example displays (types) all lines that contain the word "teeth":

```
Begin_Of_File( )
repeat (ALL) {
    Search("teeth")
    Type(0) Type( )
    Line( )
}
```

Type(0) displays from the beginning of the line up to the edit position and **Type( )** displays from the edit position to the end of the line. Therefore, **Type(0) Type( )** displays the current line no matter where the edit position is on it. Without **Line( )**, the macro would just display the same line over and over again.

Ending Repeat (and other) Loops

The **Repeat** loop above is not an "infinite" loop because it will stop with the error message "CANNOT FIND ..." when no more occurrences of "teeth" can be found.

Similarly, a **Repeat** loop that advances through a file with the **Line( )** command will stop with the error message "END OF BUFFER" when the end of the file is reached.

Repeat loops can end in five ways:

- When the repeat count is exhausted.
- When a **Goto**, **Break_Out** or **Return** command is executed.
- When a **Break** statement ends the loop. Execution then continues with any commands following the ending "**}**".
- When an error occurs, such as a search failure or the **Line( )** command reaches the end-of-file.
- When the user presses **[CANCEL]** (<**Ctrl-C**>).

Since many command loops process a file line by line or by searching for the next occurrence of some text, the **Line( )**, **Search( )** and **Replace( )** commands normally stop execution with an error message if they are unsuccessful. This prevents casually written command loops from getting into infinite loops and doing unexpected things. This is also convenient when a simple command loop is entered at the "COMMAND:" prompt.

More refined macros can use the "NOERR" option with **Line( )**, **Search( )** and **Replace( )** to suppress the error. The success of the command can then be tested and appropriate action taken. The "NOERR" option must be used with care to avoid "infinite" loops.

Alternatively, the "ERRBREAK" option is often sufficient. If the command is unsuccessful, it performs the equivalent of a **Break** command which ends the current command loop. Execution then continues with any commands following the ending "**}**".

In the printing example above, if there were only five pages to print, the macro would end with the unfriendly error message "END OF BUFFER REACHED". The following macro ends gracefully with the message "Printing is done".

```
repeat (10) {
    Print(20)  Print_Eject( )  Line(20,ERRBREAK)
}
Message("Printing is done.")
```

IMPORTANT WARNING Improperly constructed command loops can result in an apparent "system crash" because the editor has entered an infinite loop. However, the editor constantly checks for **[CANCEL]** (**<Ctrl-C>**) which stops any command execution.

Therefore, before assuming that your system has crashed from within VEDIT, always press **[CANCEL]** (**<Ctrl-C>**) which will break out of any infinite loop or other lengthy operation.

Commands versus Repeat Loops

Although a **Repeat** loop could use a simple **Replace( )** command to replace all occurrences of a string, it is better to do the same thing by using the "ALL" option with the **Replace( )** command. It is simpler and executes much faster. Therefore, the second command below is the preferred one:

```
repeat (ALL) { Replace("teeths","teeth") }
```
Poor.

```
Replace("teeths","teeth",ALL)
```
Better, faster.

Of course, the **Replace( )** command will commonly appear inside iteration loops that also contain other commands. The **Type( )** command can be used to display the lines that were changed. For example, the command to change all occurrences of "teeths" to "teeth" and display those lines which changed is:

```
Begin_Of_File( )
repeat (ALL) {
    Replace("teeths","teeth")
    Type(0) Type( )
}
```

The **Ins_Text( )** command can use the "COUNT" option to insert multiple occurrences of the same text. For example, the following two equivalent commands insert the text " three times" in triplicate. The second command is preferable.

```
repeat (3) {
    Ins_Text(" three times")
}
```
Poor.

```
Ins_Text(" three times",COUNT,3)
```
Better, faster.

However, the **Ins_Text( )** command is needed in iteration loops for more complex insertions. For example, the following command line creates a table of 60 lines with each line consisting of 80 periods, e.g. ".....".

```
repeat (60) {
    Ins_Text(".",COUNT,80)
    Ins_Newline( )
}
```

In summary, before placing a single command inside of a **Repeat** loop, check if the same thing can be done with the command options "ALL" or "COUNT".

All command loops begin operation at the current edit position. Therefore, be sure to place the edit position correctly before executing an iteration loop. If you need to start at the beginning of the file, precede the loop with the **Begin_Of_File( )** command as in several of the examples above.

While and Do-While Loops

The **While** loop repeats for as long as a specified condition is TRUE. Its format is:

```
while (c) {
    commands
    ...
}
```

The statement evaluates the condition '*c*'. If TRUE, the '*commands*' are executed. Then the condition is re-evaluated. The loop repeats as long as the condition is TRUE or until a "break-out" condition occurs.

It is important to note that if '*c*' is initially FALSE, the '*commands*' are not executed at all.

The **Do While** loop is similar. Its format is:

```
do {
    commands
    ...
} while (c)
```

The statement executes the '*commands*'. It then evaluates the condition '*c*'. If TRUE, the loop is repeated. The loop repeats as long as the condition is TRUE or until a "break-out" condition occurs.

The difference between the **Do-while** and **While** loops is that with **Do-while** the '*commands*' are executed at least once and are executed before the condition is tested. This may seem like a trivial difference, but in practice it is an important one.

In practice, **While** loops are used more often than **Do-while** loops.

The following macro prints all text in the edit buffer with 50 lines printed per page. The **While** condition tests for the end-of-file condition.

```
Begin_Of_File( )              Goto BOF
while (! At_EOF) {            While NOT at EOF
    Print(50)                 Print 50 lines
    Line(50,NOERR)            Advance by 50 lines
    Print_Eject( )            Start a new page
}                             End of loop
```

Notice that this macro would do nothing if the current edit buffer (file) were empty. If the macro were written using a **Do-while** loop, an empty buffer would cause a blank page to be printed.

For Loop

NOTE: If you are not familiar with the C programming language, you may want to first read the following topics "Numeric Capability" and "Numeric Expressions", and then come back to this section.

A loop often involves: (1) initializing a numeric variable(s), (2) testing a condition to see if the body of the loop is to be executed, (3) incrementing (adjusting) the variable(s) and (4) re-evaluating the condition. The **For** statement does all this in a convenient manner:

```
for (cm1; c2; cm3 ) {
    commands
    ...
}
```

Notice that '*cm1*' and '*c2*' are each followed by a semicolon "**;**" (semicolon).

The statement executes the command(s) '*cm1*' once; this is the initialization. It then evaluates the condition '*c2*'. If TRUE, the '*commands*' are executed. The command(s) '*cm3*' are then executed. Finally, the condition is re-evaluated. The loop repeats, executing '*cm3*' each time, as long as the condition is TRUE or until some other "break-out" condition occurs.

Let's say you wanted to create a file containing the following lines:

```
This is line #1
This is line #2
...
This is line #100
```

It could be done with the following **While** loop. (The statement **#1++** increments numeric register #1.)

```
#1 = 1
while (#1 <= 100) {
    Ins_Text("This is line #")
    Num_Ins(#1,LEFT)
    #1++
}
```

Use of a **For** loop is preferable because all of the manipulation and testing of the numeric variable is done in one place.

```
for (#1 = 1; #1 <= 100; #1++) {
    Ins_Text("This is line #")
    Num_Ins(#1,LEFT)
}
```

If-then and If-then-else Statements

The **If-then** statement performs an operation if a condition is TRUE and skips the operation if the condition is FALSE. Its format is:

```
if (c) {
    commands
    ...
}
```

The statement evaluates the condition '*c*'. If TRUE, the '*commands*' are executed once. If the condition is FALSE, the '*commands*' are skipped and execution continues with any commands following the "**}**".

The condition '*c*' is often a relational expression such as "**#1 == 20**" that can only evaluate to 0 (FALSE) or 1 (TRUE). However, in practice '*c*' can be any numeric expression. If it evaluates to non-zero, it is TRUE. If it evaluates to 0 (zero), it is FALSE.

In the following example, the condition '*c*' is a command (internal value) that returns a positive number:

```
if (Reg_Size(10)) {
    Message("T-Reg 10 is in use.\n")
}
```

The "**!**" (Not) operator is often used in the condition to flip the truth value of an expression. The following two examples are equivalent:

```
if (Reg_Size(10)==0) {
    Message("T-Reg 10 is empty.\n")
}
if (! Reg_Size(10)) {
    Message("T-Reg 10 is empty.\n")
}
```

The **If-then-else** statement performs one alternative if a condition is TRUE and the other alternative if the condition is FALSE. Its format is:

```
if (c) {
    commands-1
} else {
    commands-2
}
```

The statement evaluates the condition '*c*'. If TRUE, the '*commands-1*' are executed once and '*commands-2*' are skipped. If the condition is FALSE, the '*commands-1*' are skipped and '*commands-2*' are executed. In either case, execution then continues with any commands following the second "**}**".

```
if (Reg_Size(10)) {
    Message("T-Reg 10 is in use.\n")
}
else {
    Message("T-Reg 10 is empty.\n")
}
```

The example above uses six lines to improve readability. The "**{**", "**}**" and "**else**" may be surrounded with any desired spaces, tabs and "newlines". At the expense of readability, you could squeeze it into two (or even one) lines.

```
if (Reg_Size(10)) { M("T-Reg 10 is in use.\n") }
else { M("T-Reg 10 is empty.\n") }
```

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

The **If-then-else** statement 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** statement. Its format is:

if (*c1*) {	If condition #1
** *alternative-1***	commands-1
} else { if (*c2*) {	Else If condition #2
** *alternative-2***	commands-2
...	.
...	.
} else { if (*cn*) {	Else If condition #n
** *alternative-n***	commands-n
} else { *default-case*	Else the default
}}}	'n' closing braces

Some programmers prefer to indent each alternative further and further, but actually each alternative is at the same "level".

Examples - Flow Control Statements

NOTE: If you are not familiar with the C programming language, you may want to first read the following topics "Numeric Capability" and "Numeric Expressions", and then come back to this section.

The following example of an **If-then** statement displays the message "Letter", followed by "All Done" if the character at the edit position is a letter. If the character is not a letter, it just displays "All Done". (Note: "**CC**" is the abbreviation for **Cur_Char**. "**&&**" is the logical "and" operator; "**| |**" is the logical "or" operator. The "**\n**" in the **Message( )** command starts a new line.)

```
if ((CC >= 'A' && CC <= 'Z') | | (CC >= 'a' && CC <= 'z')) {
    Message("Letter\n")
}
Message("All Done\n")
```

The above test for a letter can be simplified (and speeded up) with the command **Match("|A")** which returns 0 if the current character is a letter.

```
if (Match("|A") == 0) {
    Message("Letter\n")
}
Message("All Done\n")
```

The following example of an **If-then-else** statement displays the message "Letter" or "Not A Letter" depending upon whether the character at the edit position is a letter.

```
if (Match("|A") == 0) {
    Message("Letter\n")
}
else {
    Message("Not a Letter\n")
}
Message("All Done\n")
```

The following example of a nested **If-then-else** displays the message "Letter", "Digit' or 'Not Alphanumeric' depending upon the character at the edit position. It then displays the final message "All Done".

```
if (Match("|A") == 0) {
    Message("Letter\n")
} else { if (Match("|D") == 0) {
    Message("Digit\n")
} else {
    Message("Not Alphanumeric\n")
} }
Message("All Done\n")
```

The following example shows how a **For** loop can be used to process a file character-by-character. For each character, it displays "Letter", "Digit" or "Not Alphanumeric".

```
for ( Begin_Of_File( ); ! At_EOF; Char( ) ) {
    if (Match("|A") == 0) {
        Message("Letter\n")
    } else { if (Match("|D") == 0) {
        Message("Digit\n")
    } else {
        Message("Not Alphanumeric\n")
    } }
}
Message("All Done\n")
```

The following example of a **While** loop displays the line numbers of lines in the edit buffer (file) that are more than 80 columns long (including TAB expansion). The command (internal value) **Cur_Col** is used to check the length of each line, and **Cur_Line** is used to indicate which line it is.

```
Begin_Of_File( )              Goto beginning of file
while (! At_EOF ) {           While NOT at EOF
    End_Of_Line( )            Goto end of current line
    if (Cur_Col > 80) {       If too long, THEN
      Message("Line #")       Display header and
      Num_Type(Cur_Line)      the line number
    }                         End of THEN
    Line(1,NOERR)             Goto next line
}                             End of While
```

The macro above can be rewritten using a **For** loop that processes the file line-by-line.

```
for( Begin_Of_File( ); ! At_EOF; Line(1,NOERR) ) {
    End_Of_Line( )
    if (Cur_Col > 80) {
       Message("Line #")
       Num_Type(Cur_Line)
    }
}
```

Break-Out Commands

It is often desirable to exit a loop when a special condition occurs. This is a necessity with a **Repeat(ALL)** loop which would otherwise execute forever. This is commonly done with the **Break** command:

Break	Breaks out of any **While**, **Do-while**, **For** or **Repeat** loop and continues with any commands following the loop's final "**}**".

Notice that **Break** only exits the innermost enclosing loop. In the case of nested loops, any outer loops will continue executing. If there is no enclosing loop, a **Break** does nothing. Don't confuse **If-then (else)** statements with loops — an **If-then (else)** has no effect on **Break**.

The **Continue** does not exit the entire loop, but rather just the current iteration. This is similar to jumping to the final "**}**" so that the loop condition is re-evaluated. It is often used to skip certain cases.

Continue	Skips the current iteration of any **While**, **Do-while**, **For** or **Repeat** loop, causing the loop to be re-tested.

Similar to **Break**, if there is no enclosing loop, **Continue** does nothing.

As a variation of our previous example, let's say you wanted to create a file containing the following lines where every 10'th line is skipped:

```
This is line #1
This is line #2
...
This is line #9
This is line #11
...
This is line #99
```

Using the **Continue** command makes this easier than it would otherwise be: (The **%** is the "remainder" operator.)

```
for (#1 = 1; #1 <= 100; #1++) {
    if ((#1%10)==0) { Continue }
    Ins_Text("This is line #")
    Num_Ins(#1,LEFT)
}
```

The **Return( )** and **Break_Out( )** commands break out of the current macro regardless of whether they occur inside a loop.

Return(*n*)	Stops the current macro and returns to any parent (calling) macro, the "COMMAND:" prompt or Visual Mode. The internal value **Return_Value** is set to '*n*' for subsequent testing.
Break_Out( )	Stops all macro execution and returns to the "COMMAND:" prompt. If a "locked-in" macro is enabled, it returns to this macro instead.
Break_Out(EXTRA)	Stops all macro execution and returns to Visual Mode. If a "locked-in" macro is enabled, it returns to this macro instead.

Goto Statement

The **Goto *label*** statement performs a jump to a specified '*label*'. There are two forms for a label:

label:	A label is a string of characters followed by a colon "**:**".
:label:	This alternate form for a label executes faster.

The label may consist of any character except space, tab, semicolon and "newline".

The label must occur within the current text register. Although labels may occur inside loops, a **Goto** can only jump within the current loop or out of a loop. It cannot jump into the middle of a loop. In the case of nested loops, a **Goto** can jump from an inner level loop into the middle of an outer level loop, but not vice versa.

The use of **Goto** statements should be minimized because they can make a macro difficult to understand — you don't know if the **Goto** is jumping forwards or backwards, a few lines or many lines.

Nonetheless, there are good uses for **Goto** statements. For example, if an error condition is detected in the middle of a complex macro, you may want to jump to an error handler at the end of your macro. If you need to break out of nested loops, a **Goto** is often easier to use and understand than the alternatives.

Although VEDIT does not have the "switch" statement of C, you can achieve the same thing (and much more) using a **Goto** with a "variable" label. Just like a search string, a "variable" label has the syntax "**@*r***" to use text register '*r*' as the entire label name or "**|@(*r*)**" to use register '*r*' as a portion of the label name.

A **Goto** with a "variable" label is a good way of jumping to the commands for processing each selection of a menu.

```
Get_Input(10, /Enter "1" thru "3": /,NOCR)
Goto Item|@(10)
...
Item1:
Message('You entered "1"\n')
Return(1)
Item2:
Message('You entered "2"\n')
Return(2)
Item3:
Message('You entered "3"\n')
Return(3)
```

Processing End-Of-File Condition

A macro that processes a file character-by-character or line-by-line will eventually reach the end of the file. The end-of-file condition always has to be treated with some care to prevent infinite loops and unwanted side effects. Usually you simply want the macro to terminate.

One easy way of handling the end-of-file condition is to use a **While** loop that tests for end-of-file as its condition. Let's look at the previous printing example again:

```
Begin_Of_File( )              Goto BOF
while (! At_EOF) {            While NOT at EOF
    Print(50)                 Print 50 lines
    Line(50,NOERR)            Advance by 50 lines
    Print_Eject( )            Start a new page
}                             End of loop
```

The **While** loop condition "**! At_EOF**" becomes FALSE when the end-of-file is reached and the loop terminates.

The "NOERR" option on the **Line( )** command prevents it from breaking out with the error "END OF BUFFER REACHED" when it reaches the end-of-file. Instead, we let **Line( )** place the edit position at the end-of-file and let the **While** condition terminate the loop.

Alternatively, the "ERRBREAK" option may be used to exit the current loop when a **Line( )** command attempts to go past the end-of-file. Although the **While** loop above is easier to understand, the printing macro could also be written as:

```
Begin_Of-File( )
repeat (ALL) {
    Print(50)
    Line(50,ERRBREAK)
    Print_Eject( )
}
```

In more complex macros it may not be easy to test for end-of-file as the condition of the loop. You will then have to test for it explicitly and take appropriate action.

if (AT_EOF) { Break }	Exit the current loop if end-of-file reached.
if (AT_EOF) { Return( ) }	Return from (exit) the current macro if end-of-file reached.
if (AT_EOF) { goto done }	Jump to the label "done:" if end-of-file reached.

Processing Unsuccessful Search

The purpose for a loop is often to search for all occurrences of something in a file and process each occurrence. After the last occurrence is processed, the **Search( )** will be unsuccessful and the macro should terminate the loop or take some other action. Since the last successful search probably did not leave us at the end-of-file, we cannot use end-of-file as the condition for terminating the loop.

In order to avoid "infinite" loops with casually written macros, an unsuccessful **Search( )** or **Replace( )**, by default, breaks out of the macro and gives the error "CANNOT FIND...". For "quick and dirty" macros this is often just fine.

For more refined macros, you must use one of the command options "ERRBREAK" or "NOERR". Here is how different **Search( )** or **Replace( )** options deal with an unsuccessful search/replace.

Search("*string*")	If unsuccessful, breaks out of the macro and gives the error "CANNOT FIND".
Search("*string*",ERRBREAK)	If unsuccessful, breaks out of any current loop, just like the **Break** command. No error message.
Search("*string*",NOERR)	Error handling is completely suppressed. You can only tell via the return value or the internal values **Error_Flag** or **Error_Match** whether the search was successful.

This variation of a previous example displays all lines that contain the word "teeth", but doesn't end with the error "CANNOT FIND":

```
Begin_Of_File( )
repeat (ALL) {
    Search("teeth",ERRBREAK)
    Type(0) Type( )
    Line( )
}
```

If your macro, upon an unsuccessful search, needs to take some special action, use the "NOERR" option. You must then test the results of the search. The following equivalent macro fragments illustrate two ways of doing this:

```
if ( Search("teeth")==0 ) goto error1
...
error1:
Message('Cannot find "teeth".\n')

Search("teeth")
if (Error_Match) goto error1
...
error1:
Message('Cannot find "teeth".\n')
```

The first example uses the return value from **Search( )** which is 0 (zero) if unsuccessful. The second example uses the command (internal value) **Error_Match** which is 1 (TRUE) if unsuccessful.

Error_Match is used instead of **Error_Flag** because **Error_Match** is only set/reset by search commands whereas **Error_Flag** is set/reset by every command. Therefore, **Error_Match** returns the result of the last search.

Using Visual Mode in Command 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.

```
repeat (ALL) { Search("temporary",ADVANCE) 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.

```
repeat (ALL) { Replace("-name-","Mr. Jones") V }
```

The Visual Mode has two ways of exiting back to Command Mode in order to work with iteration loops. **[VISUAL EXIT]** (<**Ctrl-F10**>) simply exits and lets any command iteration continue. **[VISUAL ESCAPE]** (<**Alt-F10**>) exits to Command Mode, but also aborts any command loop or command macro. The latter is used when you realize that the command loop is not doing what was intended and you want to abort it. For example, to change all occurrences of the word "and" to "or", the following command may have been given:

```
repeat (ALL) { Replace("and","or")  V }
```
Not quite right!

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 command macro, and the following correct command can then be given:

```
repeat (ALL) { Replace("and","or",WORD)  V }
```
Right.

Numeric Capability

VEDIT has extensive numeric capabilities. "Numbers" used almost anywhere in VEDIT can be "*constants*", "*variables*" and algebraic "*expressions*".

HINT: The "on-line calculator" lets you evaluate any numeric constant, variable or expression by simply typing it at the "COMMAND:" prompt preceded by a "." (period). Or precede it with "$" to display the result in hexadecimal.

Numeric Constants

There are four types of numeric constants. There is a distinction between "Integer" and "Signed Integer" because some commands take simple integers while others take signed integers.

Types of Numeric Constants

Type	Example	Description
Integer	345	A simple integer in the range 0 to 2,147,483,647.
Signed Integer	-239 -59021	A signed integer in the range +/- 2,147,483,647. (Integers are 32 bits wide.)
ASCII Constant	'A' ^B	(Quote A Quote) (Caret B) The value of an ASCII 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>**.
Reserved-word	MAXNUM	This is a shorthand for the maximum integer 2,147,483,647. All reserved words are numeric constants.

Examples - Numeric Constants

Many VEDIT commands take a numeric argument. For example, the **Type( )** command types (displays) lines of text:

Type(12)	Type the next 12 lines of text.
Type(-23)	Type the previous 23 lines of text.
Type(ALL)	Type the rest of the file.

The following example shows one way to insert a **<Ctrl-S>** into the text: (Type "^" and then "**S**")

Ins_Char(^S)	Insert a **<Ctrl-S>** into the text.

Numeric Registers (Variables)

There are 128 numeric registers or "variables" named "**#0**" thru "**#127**". #0 thru #99 are for general purpose use. Similar to the corresponding text register numbers, #100 thru #127 are reserved for special purposes, such as keystroke macros and event macros.

The "=" assignment operator is used to assign values to the numeric registers:

#20 = 754 Sets numeric register 20 to the value 754.

Several numeric registers can easily be set to the same value as the following example illustrates:

#21 = #22 = #24 = #24 = #25 = 0

Set numeric registers 21 thru 25 to 0 (zero).

You can display the value of a numeric register by entering its name at the "COMMAND:" prompt. The on-line calculator feature will then display its value.

It is very common to add a number to a numeric register. This could be done as follows:

#20 = #20 + 123 Add 123 to numeric register 20.

The following is a shorthand for this common operation. It also executes faster.

#20 += 123 Shorthand — add 123 to numeric register 20.

VEDIT supports four assignment shorthands that can be used with numeric registers.

+=	Add a number to a numeric register.
-=	Subtract a number from a numeric register.
*=	Multiply the numeric register by a number.
/=	Divide the numeric register by a number.

The "++" and "--" operators make it easy to increment or decrement a numeric register:

#30++ Increment the value of numeric register 30.

#40-- Decrement the value of numeric register 40.

When a numeric register is part of a numeric expression, the "++" and "--" operators increment/decrement the register *after* its original value is used.

#10 = 100
#11 = #10++ Following these statements, register 10 will have value 101 and register 11 will have value 100.

NOTE: Although there are 127 numeric registers; registers 100 thru 127 are reserved for keystroke macros and special purposes and you should not expect them to maintain their value for longer than one keystroke. Appendix G and the on-line help topic "USAGE" list for what purpose these registers are reserved.

Numeric Register Indirection (Arrays)

Numeric registers are usually specified with a constant, e.g. "**#20**" is always register twenty. Occasionally it is useful to use a variable register number.

#@10	Indirect specification of a numeric register. E.g. if numeric register 10 contains 95, then "#@10" specifies numeric register 95.

This allows setting up a (small) array of numbers. For example, let's say you wanted to count the capital letters in a file. You could use numeric register 0 to count "A", register 1 to count "B" and so on; register 25 would count "Z".

The following macro commands would find the next capital letter and count it:

Search("|U") **#99 = Cur_Char-'A'** **#@99++**	Search for the next capital letter and count it using the array consisting of numeric registers 0 through 25.

Internal Values (Commands)

Each command executed returns a numeric value. A large number of commands perform no function other than to return a value, usually some type of internal status information. These commands are called "*internal values*". You can think of them as read-only numeric variables.

The on-line help gives a brief description of all commands that can be considered *internal values*. Chapter 4 (Command Reference) gives a detailed description of each command.

Some commands such as **Block_Begin( )** perform a dual purpose. If an argument is specified, it sets the block-begin marker. If no argument is specified, it only returns the value of the current block-begin marker.

Block_Begin(100)	Set the block-begin marker to file position 100. Returns the value of the new block-begin marker.
Block_Begin	Only returns the value of the current block-begin marker (-1 if not set).

While **Block_Begin(*n*)** and similar commands return their new value, the **Config(*name,n*)** command returns its previous value. This makes it easy to save the previous configuration value and restore it later.

#90 = Config(W_RT_MARG,70) **...** **Config(W_RT_MARG,#90)**	The right margin is set to column 70; the original value is saved in numeric register 90. The original right margin value is later restored.

Numeric Expressions

VEDIT computes both "numeric expressions" (similar to a calculator) and "conditional expressions" which evaluate to a truth value (TRUE or FALSE) for use with flow control statements.

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

There are three types of operators — two evaluate to a numeric value and one evaluates to a conditional value.

Numeric operators	Take one or two numeric operands and evaluate to a signed integer.
Relational operators	Take two numeric 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:

12 / 3 + 7	(Numeric) The operators are "/" and "+" and the expression value is 11.
35 > 14	(Relational) The operator is ">" and the expression value is 1 or TRUE.
(35 > 14) && (17 == 23)	(Logical) 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.*** Any operator could be used with any operand, but it is usually not meaningful to use conditional operands with numeric or relational operators.

Most important is how the evaluated numeric value is used by the following command or flow control statement. If a command expects a numeric expression, but is given a conditional expression, the command will only see the two values "0" and "1". If a numeric expression is used where a conditional is expected, the numeric values "1" and "0" will be interpreted as TRUE and FALSE. Generally, any other positive value will also be interpreted as TRUE, while any negative value will be interpreted as FALSE.

Numeric Operators

The numeric operators are:

+	Addition
-	Subtraction (also performs unary minus function)
*	Multiplication
/	Division
%	Remainder of division
&	Bitwise AND
|	Bitwise OR
^	Exclusive OR (XOR)
<<	Left Shift
>>	Right Shift
~	Bitwise complement (also called 1's complement)

Examples of these operators and the resulting values are:

12 + 19	31
54 - 36	18
-4 * 16	-64
14 / 4	3
14 % 4	2
14 & 7	6
14 | 7	15
14 ^ 7	9
100 << 4	1600
256 >> 4	16
~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 **Remainder**. If you are only interested in the remainder, use the remainder operator "**%**", which returns a signed remainder with the same sign as the quotient.

The "**-**" operator performs subtraction when between two operands, or may precede an operand to change its sign.

NOTE All numeric values have a range of **+/-2,147,483,647** (32 bit accuracy). If a calculation falls outside of this range, division by zero is attempted, or if 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 (Either "!=" or "<>" can be used.)
>=		Greater than or equal to
>		Greater than

Examples of these operators and the resulting values are:

4 < 12	1 or TRUE
-13 <= -4	1 or TRUE
#1 == #1+5	0 or FALSE
-9 != 9	1 or TRUE
-5 >= 0	0 or FALSE
10 > 10	0 or FALSE

NOTE!!! Don't use "=" when you need "==". The statement "#10==90" tests if numeric register 10 contains 90. The statement "#10=90" assigns 90 to numeric register 10 and is always TRUE. This is a very common mistake made in both C and VEDIT programming.

Logical Operators

The logical operators are:

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

The following examples show how the operators are used:

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 as in the following examples. Numeric variables are used to make the examples more realistic. Note that numeric variables can be used as conditional operands when they contain only the values "1" or "0".

For the following examples assume: **#1 = 12**, **#2 = -7**, **#4 = 0** , **#5 = 1**

(#1 > 10) && (#2 <= -7)	1 or TRUE
(#2 != 0) && #4	0 or FALSE
(#1 == #2) | | #5	1 or TRUE
(#1 == #2) | | (! #5)	0 or FALSE
! (#4 && #5)	1 or TRUE

Operator Precedence

In expressions with 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 inside the parentheses will be evaluated before the entire parenthesized expression is itself used as an operand. Additional parentheses can be used to improve the readability of an expression since it is not always obvious what the precedence of operators is (unless C was your mother language). With operators of the same precedence, the leftmost one will be evaluated first.

Table of Operator Precedence

Highest:	! ~ ++ -- + -	Unary
	* / %	Multiplication, Division, Remainder
	+ -	Addition, Subtraction
	<< >>	Shift
	< > == etc.	Relationals
	&	Bitwise AND
	| ^	Bitwise OR, Exclusive OR (XOR)
	&&	Logical AND
	||	Logical OR
Lowest:	=	Assignment

Additional Numeric Features

Displaying Numbers

The **Num_Type( )** command is used to display numbers.

Num_Type(#10)	Display the value of numeric register 10, right justified..
Num_Type(#10,LEFT)	Display the value of numeric register 10, left justified.
Num_Type(#10,FILL)	Use leading "0" for any padding instead of spaces.
Num_Type(#10,NOCR)	Suppress the "newline" (CR+LF) following the number.
Num_Type(#10,HEX)	Display the number in hexadecimal, left justified, in the format "0Xhhhh: hhhh" with as many 'hh' hex digits as needed.
Num_Type(#10,HEX+NOMSG)	Display the number in hexadecimal, left justified, in the format "hhhhhhhh" with as many 'hh' hex digits as needed; the "0X" and ":" are suppressed.

Insert (Line) Numbers into Text

Similar to the **Num_Type( )** command, **Num_Ins( )** inserts the number into the text at the edit position. For example, the following macro inserts 200 lines of the form "This is line number nnnn", where `nnnn' increments for each line:

```
for (#1 = 1; #1 <= 200; #1++) {
    Ins_Text("This is line number  ")
    Num_Ins(#1)
}
```

Some applications require line numbers at the beginning of each text 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.

```
#1 = 100                    //Set starting line number
Begin_Of_File( )            //Goto BOF
repeat (ALL) {              //Start a Repeat-forever loop
    Num_Ins(#1,NOCR)        //Insert line number for this line
    Ins_Char(' ')           //Need extra space
    Line(1,ERRBREAK)        //Goto beginning of next line
    #1 += 10                //Add the increment
}                           //End of Repeat loop
```

Reading Numbers in Text

The **Num_Eval( )** command reads and evaluates the number at the edit position. The command can read and evaluate numeric constants only or entire numeric expressions.

#10 = Num_Eval()	Read and evaluate the numeric expression at the edit position and save it in numeric register 10.
#10 = Num_Eval(ADVANCE)	Advance the edit position past the evaluated numeric expression.
#10 = Num_Eval(SUPPRESS)	Read and evaluate the simple numeric constant at the edit position and save it in register 10.

If the edit position were at the text "12345ABCDE", use of the "**SUPPRESS**" option would make no difference. However, it would make a difference if the text were "12345+6789ABCDE".

Command Return Values as Numeric Arguments

A powerful feature of the C language duplicated in VEDIT is for the return value of one command to be the numeric argument of another command. The following example illustrates this:

```
#90 = Buf_Switch(Buf_Free)
...
Buf_Switch(1)
...
Buf_Switch(#90)  Buf_Close( )
```

The first line switches to an unused edit buffer. The command (internal value) **Buf_Free** returns the number of a free buffer. This is used as the argument to **Buf_Switch( )**. Since **Buf_Switch( )** returns the ID number of the new edit buffer, it can be saved in numeric register 90 for later use.

Since the value of an assignment statement is the assigned value, the first line could also be written as:

```
Buf_Switch(#90 = Buf_Free)
```

Numeric Register Stack (Technical)

Similar to the text register stack, there is a numeric register stack on which a macro can save numeric registers and later restore them. This is often done by subroutine macros that don't want to disturb their parent macro's registers. Up to 100 values can be "pushed" (saved) on this stack.

NOTE: Unlike the text registers, saving numeric registers on the stack does not change or empty their contents.

The command to save numeric registers on the stack is **Num_Push(*x, y*)** where '*x*' is the first register to be saved and '*y*' is the last register to be saved. Similarly, the command **Num_Pop(*x, y*)** restores the specified numeric registers.

Num_Push(10,19)	Save numeric registers 10 through 19 on the stack.
Num_Push(30,30)	Save the single register 30 on the stack.
Num_Push(0,99)	Save all registers on the stack.
Num_Pop(0,99)	Restore all registers from the stack.
Num_Pop(30,30)	Restore the single register 30 from the stack.
Num_Pop(10,19)	Restore registers 10 through 19 from the stack.

The **Num_Pop( )** commands should normally occur in the reverse order from the **Num_Push( )** commands.

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 Display Commands

Messages can be displayed on the screen with the **Message( )** command. This command is abbreviated as **M( )**. Messages and prompts can have embedded "**\n**" to display multiple line messages.

Message("*text* ")	Display '*text*' in the current window. Since the '*text*' may be several lines long, detailed menus can be displayed.
M("Part 1 is done")	Display a message on screen.
M("Line 1\nLine 2\n")	Display a two line message on the screen. The message ends with a "newline".

Message( ) is the primary command for displaying messages such as user prompts and menus. It can also be used to display progress messages or debugging messages during the execution of command macros.

Messages can temporarily be displayed on the status line:

M("*text* ",STATLINE)	Briefly display '*text*' on the status line. *text*' can only be one line long.

The status line will be restored when VEDIT is ready for the next keystroke. Alternatively, you can leave the message on the status line for one keystroke:

Statline_Message("*text* ")	Display '*text*' on the status line for one keystroke.

Spaces and "newlines" can also be displayed:

Type_Space(*n*)	Display '*n*' spaces.
Tab_Out(*n*)	Display spaces out to column '*n*'. If already at or past column '*n*', display two spaces.
Type_Newline(*n*)	Display '*n*' "newlines".

Single characters can be written to the screen with the **Char_Dump( )** command. Its primary purpose is to display a character whose value is in a numeric register.

Char_Dump(*n*)	Displays (dumps) the single character with decimal value '*n*' followed by a "newline".
Char_Dump(*n*,NOCR)	Displays (dumps) the single character without the "newline".

Char_Dump(#1,NOCR)	Display the character in numeric register "1"; suppress the "newline".

Char_Dump( ) can also be used to display special control and graphic characters since they are "dumped" to the screen and are not expanded.

Char_Dump(1) **Char_Dump(129)**	Display two graphic chars with values 1 and 129.

The "cursor" at which the following text will be displayed can be positioned anywhere in the current window.

Win_Hor(*n*)	Moves the cursor horizontally to column '*n*'.
Win_Vert(*n*)	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. The erased parts of the screen use the "erase attribute" set with the **Win_Color( )** command.

Win_Clear( )	Erases the entire window and moves the cursor to the upper left corner ("home" position).
Win_EOL( )	Erases from the cursor to the end of the window (screen) line.
Win_EOS( )	Erases from the cursor to the end of the window (screen).

Several "internal values" assist when manipulating windows inside of macros. **Win_Cols** and **Win_Lines** return the size of the current window. **Win_Hor** and **Win_Vert**, without arguments, return the current cursor position in the window. **Win_Color** returns the color attribute for the text in the current window. You may want to save this value in a numeric register before changing colors so that you can restore the original color later.

Changing Screen/Window Color

The **Win_Color(*n*)** command changes the color attribute of the text in the current window to '*n*'. Different windows can be displayed in different colors. The initial colors are set during configuration.

For non-IBM PC versions, **Win_Color(1)** sets reverse video and **Win_Color(0)** sets normal video.

For monochrome IBM PC, **Win_Color(112)** sets reverse video and **Win_Color(7)** sets normal video. Other values set screen attributes such as intense and underline.

The complete list of color attributes is available in the on-line help for **{CONFIG, Colors}**.

Details About Command Mode Output

Many commands such as **Directory( )**, **Num_Type( )** and **Message()** display output. While this output usually is simply displayed in the current window, VEDIT sometimes switches to another window to display the output.

For example, when a keystroke macro with Command Mode commands is run from within Visual Mode, VEDIT attempts to display it in a Command Mode window.

These rules determine exactly where Command Mode output is displayed:

1. If no windows exist (e.g. during startup), the Command Mode window "$" is created, and the output goes to this window.

   (You will notice this if you "vpw -xwildfile.vdm".)

2. If the current window already has Command Mode output, it continues to use this window. (This is rather obvious).

   It also uses the current window if it was just selected with the **Win_Switch( )** command.

   Else, the current window is being used for Visual Mode, and...

3. If the dedicated Command Mode window "$" exists, it switches to this window.

4. If there is another Command Mode window, it switches to the last used window. However, it does not switch to the status line.

   Else, there are no Command Mode windows, and...

5. It converts the current (Visual Mode) window into a Command Mode window, scrolls to the bottom and displays the output.

   (This is always the case if you only have one window.)

If needed, use the **Win_Switch( )** command to force output to the desired window.

If there are no Command Mode windows, the current Visual Mode window is converted to a Command Mode window. Upon reentering Visual Mode, you will get the "Press any key to continue..." prompt so that you can see the Command Mode output. (The command **Visual_Macro(NOMSG)** can be used to suppress the prompt.)

Input Commands

Macros can accept keyboard input using three commands. Each input command includes a "*prompt*" to the user.

Table of Input Commands

Command	Type of Input	Example of Input
Get_Key("*prompt*")	Single Keystroke	Y
	Function key	**[CURSOR DOWN]**
Get_Num("*prompt*")	Number	-123
	Expression	(123 + 345) / 18 - 1
Get_Input(*r*,"*prompt*")	Text String	newfile.txt

The "*prompt*" can be one or more lines long. A multiple line prompt can be entered on multiple lines or the "*prompt*" can include "**\n**" to indicate "newlines".

The command option "STATLINE" on these commands causes the prompt to appear on the status line. In this case the '*prompt*' must be a single line long.

Sometimes it is more convenient to prompt with the **Message( )** command. In this case the non-existent prompt can be entered as two quotes **""** without any characters between them, or it can be left off completely.

Get_Num( ) and **Get_Input( )** require pressing **<Enter>** following the input line. The input line can also be edited in the normal fashion and **[CURSOR UP]** will recall previous input lines.

Get_Key("*prompt*") Prompts on a new line with '*prompt*'. Returns the value of the next keyboard character or function code. Simple keys have value 00 - 255. Function-codes have value > 255.

Get_Num("*prompt*") Prompts on a new line with '*prompt*'. Returns the value of the number (or numeric expression) that was entered.

Get_Input(*r*,"*prompt*") Prompts on a new line with '*prompt*'. The user's input line including the **<Enter>** is stored in text register '*r*'. **<Enter>** is converted to the "newline" character(s).

Get_Input( ) is often used to prompt the user for a filename.

```
Get_Input(10,"Enter filename: ",STATLINE)
File_Open(@10)
```

Prompt user on the status line for a filename; save it in register 10 and then open that file.

The "newline" can be left off the stored line by using the command form:

Get_Input(*r*,"*prompt*",NOCR)

Store the input line in text register '*r*', but without the "newline".

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

Get_Input(*r*,"*prompt* ",APPEND)

Append the input line to register '*r*'.

A default input string can be specified and the length of the input string can be limited.

Get_Input(10,"Enter up to 8 chars: ","Default",COUNT,8)

Prompt with a default input string of "Default", the maximum input string length is 8 chars.

NOTES: The topic "Edit Function Codes" in the Appendices lists how each edit function is decoded by the **Get_Key( )** command. Each has a two letter code and an equivalent numeric code, e.g. **[CURSOR UP]** is coded as "CU" and 'C'+'U'*256 = 21827. All unused function keys are decoded as "B4" = 13378.

The internal values **Key_Status** and **Previous_Key** are also related to reading keyboard input.

Checking for Valid User Input

It is often important for a program to check that the user has entered a valid response to a prompt. For example, if the program prompts the user to enter "1" through "4", it should check that other numbers or characters were not entered.

A command loop is typically used to wait for valid input. The following example prompts for a "Yes/No" response and waits for a valid response.

```
Message("\nSelect [Y]es or [N]o : ")
Repeat(ALL) {
    #80=Get_Key("") & 0xDF
    if (#80=='Y' | | #80=='N') { Break }
    Alert()
}
```

Notice that the prompt is displayed with the **Message( )** command instead of with **Get_Key( )**. This has the advantage that the prompt is only displayed once in case invalid entries are made. Otherwise it would be re-displayed on following screen lines after each invalid entry.

The line "**#80=Get_Key("") & 0xDF**" reads the next key into numeric register 80. The "**& 0xDF**" converts lower case letters into upper case for simpler testing. If the response is valid, it breaks out of the loop; otherwise the loop executes **Alert( )** to give a beep and waits again for the next keyboard character.

The following example gets a valid numeric input between "1" and "4".

```
Message('Enter "1" - "4" : ')
while ( (#80=Get_Key(""))<'1' | | #80>'4' ) {
    Alert( ) }
Char_Dump(#80)
```

The condition of the **While** loop reads the next key into numeric register 80. If it is invalid, the condition is TRUE and the **Alert( )** command is executed to give an error beep. A valid input is accepted and echoed to the screen with the **Char_Dump(#80)** command.

The following example shows a typical way of prompting the user for a file name and ensuring that the file exists. The **Repeat** loop repeatedly prompts the user until a valid file name is entered. This example then simply displays the specified file.

```
repeat (ALL) {
    Get_Input(10,"Enter filename: ")
    if ( File_Exist(@10) ) { Break }
    Message("Cannot find file!  Try Again.\n")
}
Type_File(@10)
```

Select Filename with a Dialog Box

Files can be selected from a dialog box with the **Get_Filename( )** command.

Get_Filename(*r*,"*filespec*")

Select a filename using a dialog box. '*filespec*' is the initial "filter". When selected, the complete drive, directory and filename is placed into text register '*r*'.

The following program fragment illustrates how **Get_Filename( )** is used:

```
Get_Input(10,"Enter filename: ")   //Prompt for filename
Get_Filename(10,@10)               //If wildcards, select from dir
if ( ! Reg_Size(10) ) {            //If no filename...
    goto error                     //  Goto error handler
}
File_Open(@10)                     //Open file for editing
```

Get_Filename( ) only displays the dialog box if '*filespec*' contains the wildcard characters "*" or "?". If it does not, **Get_Filename( )** simply copies the '*filespec*' into register '*r*'.

Custom Dialog Boxes (Windows only)

In the Windows version, a macro can prompt for user input with a true "Windows" dialog box by using the flexible **Dialog_Input_1( )** command. It can create a dialog box with a title, text, up to ten buttons, and optional check boxes, radio buttons and input strings.

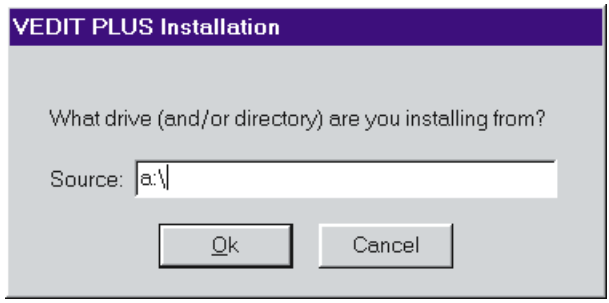
The dialog box is dynamically sized according to the specified text, the number of buttons, and the other specified items. The text can be one or more lines long; about 2000 characters should be considered a practical upper limit. The dialog box can be precisely positioned with respect to the screen or the VEDIT application window.

The command has a very detailed syntax which is fully described in the on-line help.

The command returns the number of the button pushed (1 - 10) or 0 if **<Esc>** was pressed to cancel the dialog box. Furthermore, text register '*r*' is set to the text entered in the first optional input line. (If there is no input line, '*r*' must still be specified, but the register will not be changed.)

Consider the following example from the VEDIT installation program:

```
#103=DI1(20,^`VEDIT 6.0 Installation`,
`What drive (and/or directory) are you installing from?`,
`??Source:`,
`[&Ok]`,`[Cancel]`^,@(1),APP+CENTER,0,0)
```



It will display the following dialog box:

After the user makes a selection, numeric register 103 will be set to 1 is "[OK]" was selected, 2 if "[Cancel]" was selected, or 0 if **<Esc>** was pressed. Text register 20 will be set to the user's input line, which defaults to "a:\".

Many of the supplied macros use the **Dialog_Input_1( )** command, usually abbreviated as **DI1( )**, including **`color.vdm`**, **`compile.vdm`**, **`detab.vdm`**, **`keyedit.vdm`**, **`runshell.vdm`** and **`sallbuff.vdm`**. Refer to these files and especially **installw.vdm** for useful examples..

Input (keyboard) Redirection

The input to the **Get_Input( )**, **Get_Key( )** and **Get_Num( )** commands normally comes from the user typing on the keyboard. However, it can also come from an "input redirection file". This allows you to fully automate a process which otherwise requires keyboard input from a user.

The input redirection file simply consists of the desired input lines. Two special "command" characters can occur at the beginning of a line:

%	Specifies that the "newline" (CR+LF or LF) at the end of the line is to be ignored. This allows single characters to the **Get_Key( )** command to be entered on a separate line.
!	Specifies a comment line; the entire line is ignored.
%%	Use "%%" when a literal "%" is needed at the beginning of a line.
!!	Similarly, use "!!" when a literal "!" is needed.

Furthermore, an input line can be followed with a comment by entering one or more spaces/tabs, "!" and the comment. In this case the whitespace preceding the "!" is considered part of the comment and is ignored.

The **Redirect_Input(**"*file*"**)** command starts the input redirection using the specified file. Input redirection stops when the end of the input redirection file is reached or the **Redirect_Input(CLEAR)** command is executed.

For example, let's say that we need to regularly change the string "<Italic>" to "<Bold>" in all the ".txt" and ".prn" files in a particular directory. The WILDFILE.VDM macro is ideal for this; it only prompts for the "wildcard" filenames and the search and replace strings. To fully automate the process, we can feed the following input redirection file, named **`typeset.inp`** to the WILDFILE macro.

```
! VEDIT input redirection file for WILDFILE macro
!
! Replace "<Italic>" with "<Bold>" in all
! c:\typeset\*.txt and c:\typset\*.prn files.
!
c:\typeset\*.txt    ! Select first group of files
c:\typeset\*.prn    ! Select next group of files
                    ! Blank line ends file selection
%r                  ! Select [R]eplace
<Italic>            ! The search string
<Bold>              ! The replace string
%d                  ! Select [D]isplay
%n                  ! Select [N]o to "More" option
                    ! Press any key to continue
3                   ! Select [3] to exit macro
```

The following command starts VEDIT and runs the WILDFILE macro with an input redirection file. ("**rinp**" is the short name for **Redirect_Input**.)

vpw -e -c'rinp("typeset.inp") call_file(100,"wildfile.vdm")'

A Windows user could create an icon for the command above and thereby perform the entire search/replace by simply clicking on the icon.

Block Operations

Determining File Position

The current edit position in the file is accessed with the command (internal value) **Cur_Pos**. This position is an "offset" from the beginning of the file, with the first character having a position of 0 (zero). You can easily display the current edit position by using the on-line calculator feature at the "COMMAND:" prompt:

Cur_Pos	Display current edit position as an "offset" from the beginning of the file.

You can save the current edit position in either a block marker, text marker or numeric register. This is very similar to setting a text marker or block marker in Visual Mode. For example:

Block_Begin(Cur_Pos)	Save current edit position as the block-begin marker.
Set_Marker(1,Cur_Pos)	Save current edit position in text marker "1".
#4 = Cur_Pos	Save current edit position in numeric register "4".

The command **Goto_Pos(*n*)** moves the edit position to the '*n*'th position in the file. Therefore, **Goto_Pos(0)** is equivalent to **Begin_Of_File( )**. **Goto_Pos( )** is often used to move to the position saved in a text marker:

Goto_Pos(Marker(2))	Move the edit position to the position saved in text marker 2.

Setting Block and Text Markers

The block-begin and block-end markers used by the **{BLOCK}** menu can be set and cleared in Command Mode. Similarly, the text markers can be set and cleared in Command Mode.

Block_Begin(*n*)	Set the block-begin marker to position (offset) '*n*' in the file.
Block_Begin(Cur_Pos)	Set the block-begin-marker to the current edit position.
Block_Begin(CLEAR)	Clear both the block-begin and block-end markers. (Any negative value clears the markers.)
Block_End(*n*)	Set the block-end marker to position (offset) '*n*' in the file.
Block_End(CLEAR)	Clear only the block-end marker.

Set_Marker(*m*, *n*)	Set text marker '*m*' to position '*n*' in the file.
Set_Marker(*m*, CLEAR)	Clear text marker '*m*'.

The block markers can be accessed via the internal values **Block_Begin** and **Block_End** and the text markers can be accessed via **Marker(*m*)**.

The text and block markers are "relative" positions in the file that "stick" to particular characters. They adjust automatically as text is inserted or deleted. Therefore, if you need to save positions in the text while inserting and deleting, you must save them in either the text or block markers. In this case numeric registers cannot be used because they contain "absolute" positions.

Block Commands

VEDIT has a variety of block oriented commands. Blocks of text are defined by two file positions (offsets) - a beginning position '*p*' and an end position '*q*'. A non-columnar block includes the character at position '*p*' and all characters up to (but not including) position '*q*'.

Block_Copy(*p,q*)	Copy a block of text to the current edit position.
Block_Move(*p,q*)	Move a block of text to the current edit position.
Block_Fill(*ch,p,q*)	Fill a block with spaces or another character.
Case_Lower_Block(*p,q*)	Convert all upper case letters in the block to lower case.
Del_Block(*p,q*)	Delete a block of text.
Detab_Block(*p,q*)	Convert all tabs in the block of text to spaces.
Print_Block(*p,q*)	Print a block of text.
Reg_Copy_Block(*r,p,q*)	Copy a block of text to register '*r*'.
Strip_High(*p,q*)	Strip Bit 8 (high bit) from a block of text.
Translate_Block(*p,q*)	Translate all characters in the block using the built-in or user loadable translation table.
Type_Block(*p,q*)	Type (display) a block of text.
Write_Block("*file*",*p,q*)	Write a block of text to the file '*file*'.

Some simple complete examples of block commands are:

Del_Block(4,9)	Delete the block consisting of the 5th through 9th characters in the file. (Counting starts at zero.)

```
#1 = 123                    Copy the 124th through 234th characters
#2 = 234                    in the file into text register "7".
Reg_Copy_Block(7,#1,#2)
```

Type_Block(CurPos,CurPos+24) Type out the next 24 characters.

The counting scheme used here may seem a little unnatural because it starts at zero and excludes the character at the end position. However, there are good reasons for doing it this way (other than that computer people love counting from zero). In practice the block positions are often set from **Cur_Pos** following a **Line( )** or **Search( )** command. Everything then works out very well.

Using "0,File_Size" as the block parameters selects the entire file. This is similar to selecting **{BLOCK, Select all}**.

Translate_Block(0,File_Size) Translate the entire buffer (file).

NOTE: The examples in this manual assume that the first block argument '*p*' is less than the second argument '*q*'. In practice they can be reversed. Just remember that the character at the "larger" position (further into the file) is excluded from normal "character" blocks.

Columnar Blocks

Most of the block commands take the command options "**COLUMN**" or "**COLSET**" to select a columnar block operation similar to those in Visual Mode. With the option "**COLUMN**", '*p*' and '*q*' specify both the beginning and end of the block, and the left and right columns for the block. In other words, they specify the "corners" of the block.

Block_Copy(*p,q*,COLUMN) Copy a columnar block of text to the current edit position. '*p*' and '*q*' specify the corners of the block.

Block_Move(*p,q*,COLUMN) Move a columnar block of text to the current edit position.

Del_Block(*p,q*,COLUMN) Delete a columnar block of text.

Type_Block(*p,q*,COLUMN) Type (display) a columnar block of text.

Reg_Copy_Block(*r,p,q*,COLUMN) Copy a columnar block of text to register '*r*'.

Write_Block("*file*",*p,q*, COLUMN) Write a columnar block of text to the file '*file*'.

Columnar blocks can also be selected with the "COLSET" option, which is followed by two explicit column numbers '*c1*' and '*c2*'. In this case '*p*' and '*q*' specify the first and last lines of the block, and '*c1*' and '*c2*' specify the columns (inclusive) of the block. '*p*' and '*q*' are still file positions, but they can be anywhere on the line; they do not need to be in the corners.

The following example locates a block beginning with "first" and ending with "last". It then copies columns 10 through 40 (inclusive) in the block, including from the lines containing "first" and "last", to text register "9".

```
Search("first")
Block_Begin(Cur_Pos)
Search("last",ADVANCE)
Block_End(Cur_Pos)
Reg_Copy_Block(9,BB,BE,COLSET,10,40)
```

Line-range Blocks

Most of the block commands take the command option "LINESET" to select a block consisting of entire lines. In this case, the block parameters are not file positions, but rather line numbers. We refer to them as '*l1*' and '*l2*'.

This option is very useful when you need to operate on entire lines and know their line numbers. For examples:

Write_Block("part1.txt",100,199,LINESET)

Write line numbers 100 through 199 (inclusive) of the current buffer (file) to the new file "part1.txt".

Block_Copy(1,10,LINESET)

Copies (duplicates) the first 10 lines of the current buffer (file) at the beginning of the current line.

Case_Upper_Block(101,MAXNUM,LINESET)

Converts to upper case all text starting with line 101 through the end of the file.

"LINESET" can be used in conjunction with "COLSET" to specify a columnar block by line numbers and column numbers. For example:

Reg_Copy_Block(9,10,20,LINESET+COLSET,30,40)

Copy the columnar block consisting of lines 10 through 20 and columns 30 through 40 to text register 9.

Using "LINESET" is similar to marking blocks in Visual Mode with **{BLOCK, Set line marker}**. When copied to a text register, it is saved as a "line" block; therefore, when it is inserted, it is always inserted at the beginning of the current line.

NOTE: Block commands using "LINESET" take longer to process because VEDIT must calculate line numbers. However, you would only notice the difference when performing many commands on multi-megabyte files.

Overriding Text Register Block Type

The **Reg_Ins( )** command inserts stream, columnar and line blocks just like **{BLOCK, Insert register}**. Normally, a block that is copied to a register is inserted as the same type of block. For special purposes, the command options "**RAW**", "**COLUMN**" and "**LINEBLOCK**" let you override this. Note: the option is ***not*** "**LINESET**".

Reg_Ins(*r*)	Insert register '*r*' at the current edit position. Insert the same type of block (stream, column or line) as was saved.
Reg_Ins(*r*,COLUMN)	Insert register '*r*' as a columnar block even if it was saved as a stream or line block.
Reg_Ins(*r*,RAW)	Insert register '*r*' as a stream block even if it was saved as a columnar or line block.
Reg_Ins(*r*,LINEBLOCK)	Insert register '*r*' as a line block at the beginning of the current line, even if it was saved as a stream or columnar block.

Setting Block Markers by Searching

The **Search( )** command 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 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:

Smith, Charles
3219 Space Ct.
Albany, NY 14311
-Broadcast Producer

Burnett, Tammie
642 Sunset Blvd.
Miami, FL 32103
-Travel Consultant

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:

Search("Smith")	Find the word "Smith" in the current edit buffer.

Since this positions the edit position at the first character of "Smith", we can save the current position as the block-begin marker:

Block_Begin(Cur_Pos) Set the block-begin marker to the beginning of "Smith".

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 could search for two "newlines":

Search("|L|L") Search for a blank line, leaving the edit position at the end of the previous line.

However, the edit position is not where we need it for setting the block-end marker. It precedes the blank line and the block should include the blank line. We could use the command **Block_End(Cur_Pos+Chars_Matched)**. However, it is preferable to use the search option "ADVANCE" which advances the edit position past the matched text.

Search("|L|L",ADVANCE) Search for a blank line, leaving the edit position past the blank line.

This position can now be used as the block-end marker:

Block_End(Cur_Pos) Set the block-end marker past the record including the blank line.

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

RegCopyBlock(4,BlockBegin,BlockEnd,APPEND+DELETE)

Append the entry to register 4.

The "APPEND" option causes the block to be appended to the existing contents of the register. The "DELETE" option causes the block to be deleted from the edit buffer.

Finally, using a Repeat loop to repeat this operation for all occurrences of "Smith", we have the command sequence (we have used some abbreviation):

```
Reg_Empty(4)                        //Empty T-Reg 4
repeat (ALL) {                      //Begin loop
   Search("Smith",ERRBREAK)         //Find "Smith"
   Block_Begin(Cur_Pos)             //Save beginning position
   Search("|L|L",ADVANCE)           //Search for a blank line
   Block_End(Cur_Pos)               //Save ending position
   RCB(4,BB,BE,APPEND+DELETE)       //Append entry to T-Reg 4
}                                   //End of loop
```

The option "ERRBREAK" on the **Search("Smith",ERRBREAK)** command causes the macro to break out of the **Repeat** loop when no more occurrences of "Smith" are found.

Text Register Commands

Reading Environment Variables

DOS/Windows, UNIX/XENIX and QNX have environment variables. The DOS "SET" command is used to create environment variables and give them values. As described in the VEDIT User's Guide, VEDIT uses the optional environment variables "VEDIT" and "VEDPATH" ("VEDIT_PATH" under QNX). A command macro can read these and any other environment variables with the **Get_Environment( )** command.

Get_Environment(*r*,"*name*") Read the value of environment variable '*name*' into text register '*r*'.

This feature allows a command macro to alter its operation according to how an environment variable is set.

Text Register Stack

(This is a moderately technical topic for advanced programmers.)

The contents of any desired text registers can be saved on a special stack. This allows a macro to save the contents of text registers that it needs for its own purposes and, when the macro is done, restore the original contents. (This implements what programmers call "local variables".) Saving text registers is particularly important when writing low level "sub-routine" macros which are called from other macros. It allows low level macros to be written without concern for which registers a higher level macro is using. Up to 128 registers can be saved on this special stack.

NOTE: The registers saved on the stack are also emptied unless the "**SET**" option is used. Edit buffers cannot be saved on the stack.

The command to save text registers on the stack is **Reg_Push(*r, s*)** where '*r*' is the first register to be saved and '*s*' is the last register to be saved. Note that a range of registers is specified — the "lowest" possible register is "0" and the "highest" is "124".

After saving, the text registers are normally emptied. However the command form **Reg_Push(*r,s*,SET)** maintains the registers.

Similarly, the command **Reg_Pop(*r,s*)** restores the specified text registers. Note that a range of registers is specified — the "lowest" possible register is "0" and the "highest" is "109".

Reg_Push(10,19) Save registers 10 through 19 on the stack.

Reg_Push(20,20) Save the single register 20 on the stack.

Reg_Push(0,99) Save all general purpose registers on the stack.

Reg_Pop(0,99)	Restore all general purpose registers from the stack.
Reg_Pop(20,20)	Restore the single register 20 from the stack.
Reg_Pop(10,19)	Restore registers 10 through 19 from the stack.

It is up to the programmer to "push" and "pop" the registers in the correct order — in particular the **Reg_Pop( )** commands should appear in reverse order from the **Reg_Push( )** commands. For example, the commands to save registers 4-8, 20-25 and 101-109, and later restore them are:

Reg_Push(4,8)	Save registers 4 through 8.
Reg_Push(20,25)	Save registers 20 through 25.
Reg_Push(101,109)	Save registers 101 through 109.
...	
...	
Reg_Pop(101,109)	Restore registers 101 through 109.
Reg_Pop(20,25)	Restore registers 20 through 25.
Reg_Pop(4,8)	Restore registers 4 through 8.

Experienced programmers can use **Reg_Push( )** and **Reg_Pop( )** that are not in reverse order to achieve special effects. For example, the following commands quickly move the contents of registers 20 through 25 to registers 30 through 35:

Reg_Push(20,25)	This "trick" copies registers 20-25 to registers
Reg_Pop(30,35)	30-35. It also empties registers 20-25.

Registers can be saved more than once on the stack. Attempts to "pop" more registers than were "pushed" are ignored. **Reg_Push**, without arguments, returns the number of registers currently pushed on the stack.

Registers 123 and 124, corresponding to the **{TOOLS}** and **{USER}** menus, can be saved on the register stack when used with the "**SET**" option. This allows a macro to save the current menus, change them for use during the macro, and then restore the menus, when the macro is done.

The **Key_Cfg_Push( )** and **Key_Cfg_Pop( )** commands save and restore the current keyboard layout on the text register stack. Similarly, **Win_Cfg_Push( )** and **Win_Cfg_Pop( )** save and restore the current window layout.

```
Key_Cfg_Push( )          //Save keyboard layout
Win_Cfg_Push( )          //Save window arrangement
Reg_Push(123,124,SET)    //Save {USER} and {TOOLS}
Reg_Push(10,20)          //Save locally used regs
...
...                      //Body of macro
...
Reg_Pop(10,20)           //Restore original regs
Reg_Pop(123,124)         //Restore {USER} and {TOOLS}
Win_Cfg_Pop( )           //Restore window arrangement
Key_Cfg_Pop( )           //Restore keyboard layout Return
```

Match and Compare

The **Match( )** command compares the text at the edit position against a "search string". The comparison can be for "equality" or lexical "greater than" or "less than". The search string can also contain pattern matching codes or, if the "REGEXP" option is specified, regular expressions.

Match("*string*") Compares the text at the edit position with '*string*'. The comparison can be quite sophisticated because '*string*' may contain pattern matching codes or regular expressions. **Match( )** has a form and options similar to the **Search( )** command. Returns {0,1,2,3} according to the results of the comparison.

The comparison is successful or "equal" when '*string*' completely matches the text at the edit position. The text strings are considered "equal" even though the edit buffer can clearly be longer than '*string*'. For **Match("*string*", COUNT, *n*)**, the '*string*' must match '*n*' times in a row. If the strings do not match completely, the internal values **Error_Flag** and **Error_Match** are set to TRUE. The results of the comparison are returned and saved in **Return_Value**:

0 If successful.

1 If the text is lexically "greater than" '*string*'.

2 If the text is "less than" '*string*'.

3 If the match failed on a pattern match code or regular expression. This is needed because "greater than" or "less than" are meaningless with pattern matching.

When successful, the internal value **Chars_Matched** is set to the number of text characters that were matched.

Match( ) Command Return Values

Edit Buffer	'*string*'	Result	Return	Chars_Matched	Error_Flag
biggest...	big	Equal	0	3	0
bigger...	big dog	Greater	1	-	1
bigger...	biggest	Less	2	-	1
biggest...	big|A	Equal	0	4	0
big dog...	big|A	Not Equal	3	-	1

The command option "ADVANCE" moves the edit position past the matching characters, but only if the entire match is successful.

Match("|W",ADVANCE) If the edit position is at any whitespace (spaces and tabs), advance past it and return 0. Else just return non-zero.

It may be helpful to consider how the **Match( )** and **Search( )** commands are different:

- **Match( )** does not search through the text trying again if the match at the current edit position is unsuccessful.
- If **Match( )** is unsuccessful, no error message is given. Therefore, the option "NOERR" is not applicable.
- The return value of **Match( )** is the result of the comparison. The return value of **Search( )** is the number of successful matches, particularly when the "ALL" or "COUNT" options are used.

The **Compare( )** command is similar to the **{SEARCH, Compare buffers}** function. It performs a character by character comparison between the current edit buffer and a text register or edit buffer. There is no pattern matching or regular expressions. The comparison does not distinguish between upper and lower case letters unless the "CASE" option is used.

Compare(*r*) Compares the text at the edit position with the text register (or edit buffer) '*r*'. Performs a simple character by character comparison. Advances the edit position past all matching characters. Returns {0,1,2} according to the results of the comparison.

If '*r*' is a simple text register, the entire register must match for the comparison to be successful. If '*r*' is an edit buffer, the comparison starts from '*r*'s edit position and the rest of the buffer must match for the comparison to be successful. The result of the comparison {0,1,2} is returned and saved in **Return_Value** as with the **Match( )** command.

0 If successful — the text matches the entire contents of '*r*'.

1 If the text is lexically "greater than" '*r*'.

2 If the text is "less than" '*r*'.

Unlike the **Match( )** command, **Compare( )** always moves the edit position past those characters that match, regardless of whether the entire comparison is successful. The internal value **Chars_Matched** is set to the number of characters that matched; it is set to 0 (zero) when the very first character does not match.

Compare(20) Compare the text at the edit position with text register 20. Return {0,1,2} according to the result of the comparison. Advance the edit position.

Compare( ) is useful for moving the edit positions 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.

Compare(2+BUFFER) Compare the text at the edit position with the text at the edit position of edit buffer 2. Advance the edit positions in both buffers over all matching characters. Return {0,1,2} according to the result of the comparison.

If you want to compare text against a text register, but want the characteristics of the **Match( )** command (e.g. pattern matching), use the command form **Match(|@(*r*))**.

The **Reg_Compare(*r,"text"*)** command performs a simple character by character comparison between the entire text register '*r*' and '*text*'. The comparison is not case sensitive unless the "CASE" option is used.

Reg_Compare(*r,"text"*) Compare the contents of text register '*r*' with '*text*'. This is a character by character comparison without Pattern matching or Regular expressions. '*r*' cannot be an edit buffer.

The result of the comparison {0,1,2} is returned and saved in **Return_Value** as with the **Match( )** command.

0 If successful — register '*r*' contains '*text*'.

1 If register '*r*' is lexically "greater than" '*text*'.

2 If register '*r*' is "less than" '*text*'.

Reg_Compare( ) can be used after a **Get_Input( )** command to test what the user entered.

Reg_Compare(10,"yes") Returns 0 if text register 10 contains "yes"; case is not important. Else returns non-zero.

Reg_Compare(10,@11) Returns 0 if text registers 10 and 11 have the same contents; else returns non-zero.

Additional Commands

Displaying Input/Output Filenames

Name_Read( ) displays the input (read) filename and **Name_Write( )** displays the output (write) filename on the screen. **Name_File( )** 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 "newline". The preceding message can be suppressed with the "NOMSG" option and the following newline with the "NOCR" option.

Name_Write	Display output filename and a "newline".
Name_Write(NOMSG)	Display output filename without preceding message.
Name_Write(NOMSG+NOCR)	Display output filename without preceding message and without a "newline".

The predefined string values "**PATHNAME**", "**FILENAME**" and "**FILE_EXT**" permit a macro to access the full pathname, just the filename or just the filename extension with any command that takes a string argument.

Directory( ) Command

The **Directory( )** command displays the current drive name and directory followed by the filenames in four (4) columns. The option "NOMSG" suppresses the drive and directory line and displays the files one per line.

Dir("",NOMSG)	Display files one per line; suppress header.

Dir("",NOMSG) is often used in conjunction with **Out_Ins( )** to insert the directory into the edit buffer. This way a command macro can 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:

Out_Ins( ) Dir("*.asm",NOMSG) Out_Ins(CLEAR)

Insert all ".asm" filenames into the edit buffer, one per line.

The directory display normally includes subdirectories plus "hidden" and "system" files. The command option "SUPPRESS" prevents these types of files from being included.

The command **File_Exist(*filespec*)** tests for the existence of the file(s) '*filespec*' and returns the number of files that matched the file specification. It returns 0 if no files were found. It is primarily used inside macros that must know if a particular file exists.

Sound Generation

The **Alert( )** command creates the default "beep" on the IBM PC speaker as long as the configurable "Beep level" is not set to zero.

Alert( ) Create the default "beep" on the IBM PC speaker.

Under Windows 3.1/95/98 (but not NT) and DOS, programmable sounds of any desired frequency and duration can also be created.

Sound(*n*,*k*) Creates a sound (tone) of frequency '*n*' hertz and duration '*k*' milliseconds.

Sound(*n*,*k*,EXTRA) The sound is followed by 30 milliseconds of silence.

Sound(262,1000) Plays the note "middle C" for one second.

The supplied macro file **`marylamb.vdm`** is an example that plays a simple tune.

WordStar Files (Strip 8th bit)

WordStar (tm) files and files from other word processors often contain characters which have their "High" or "8th" bit set. These are often difficult to edit with VEDIT because the high bit characters are displayed as graphics characters. Such files can be converted to normal text files with the **Strip_High( )** command which "strips" the 8th bit:

BOF( ) Strip_High(ALL) Strip the 8th bit from every character in the file.

Strip_High(10) Strip the 8th bit from all characters in the next 10 lines.

If the paragraphs from the word processor are justified, they are easier to edit if you first "unjustify" them to remove the extra spaces between words. This is described in the VEDIT User Guide (Chapter 4, Word Processing Functions).

The supplied **`wordstar.vdm`** macro converts a normal text file into a WordStar file. It changes all single carriage-returns into "soft carriage-returns" which have their high bit set. Multiple carriage-returns between paragraphs are left unchanged. It also ensures that each soft carriage-return is preceded by at least one space.

Modify Keyboard Layout

The keyboard layout can be modified with commands. Keystroke macros can be added and deleted, the basic Edit Function assignments can be changed and the entire keyboard layout can be saved to disk and loaded from disk.

The **Key_Save( )** and **Key_Load( )** commands are similar to the **{CONFIG, Keyboard Layout, Save to Disk}** and **{CONFIG, Keyboard Layout, Load from Disk}** functions.

Key_Load("*file*") Load new keyboard layout from the file '*file*'.

Key_Save("*file*") Save current keyboard layout, including any keystroke macros, in "text" format to the file '*file*'.

Key_Save("*file*",BINARY) Save the keyboard layout in "binary" format.

The **Key_Add( )** command provides more flexibility in adding keystroke macros than **{CONFIG, Keyboard Layout, Add Keystroke Macro}**.

Key_Add("*Key-seq*","*Edit-seq*") Add the key assignment to the end of the keyboard layout table.

Key_Add("*Key-seq*","*Edit-seq*",OK)

Skip confirmation to overwrite any existing assignment to '*Key-seq*'.

Here are some examples:

Key_Add("Alt-A","[VISUAL EXIT] Line(0) Block_Copy(1)")

Add a keystroke macro which defines **<Alt-A>** to be a function which duplicates the current line of text.

Key_Add("F1","[HELP]",OK)

Define **<F1>** to be the basic Edit Function **[HELP]**; any previous assignment to **<F1>** is removed without confirmation.

The command option "INSERT" inserts the new key assignment at the beginning of the keyboard layout; ***it also permits multiple assignment to the same Function/control key***. In the case of multiple assignments, VEDIT will use the first one in the keyboard layout. The primary use for the "INSERT" option is to make a temporary change to the layout without disturbing the original layout. The **Key_Pop( )** command is then usually used to remove these temporary assignments.

Key_Add("*Key-seq*","*Edit-seq*",INSERT)

Insert the key assignment at the beginning of the keyboard layout table.

Key_Pop(*n*) Pops (removes) the first '*n*' key assignments from the beginning of the keyboard layout table.

For example, a menu might prompt the user to make a selection by pressing "1" through "4" or pressing **<F1>** through **<F4>**. One way of handling this is by temporarily assigning digits to the function keys.

```
Key_Add("F1","1",INSERT)
Key_Add("F2","2",INSERT)
Key_Add("F3","3",INSERT)
Key_Add("F4","4",INSERT)
#1 = Get_Key('Press "1" - "4" or <F1> - <F4> : ')
Key_Pop(4)
```

When using **Key_Pop( )**, it is important to pop (remove) the correct number of key assignments, otherwise the original layout will be disturbed. You may find the command **Key_Pop(ALL)** useful for removing all key assignments, in effect initializing the keyboard layout.

NOTE: After any change to the keyboard layout, VEDIT ensures that at a minimum, the functions **[RETURN]** and **[ESCAPE]** are defined. If necessary, they are assigned to **<Enter>** and **<Esc>** respectively. This ensures that you can always exit from Visual Mode and save your files. Press **<Esc>** to bring up the **{ESCAPE}** menu, press **<Space Bar>** to select the desired item and press **<Enter>**.

You can also remove single key assignments from anywhere in the keyboard layout with the **Key_Delete( )** command. In the case of multiple assignment to the same Function/control key, it will delete the first one in the table.

Key_Delete("*Key-seq*") Delete the keyboard assignment to '*Key-seq*'.

Key_Delete("*Key-seq*",NOERR)
Suppress error if '*Key-seq*' not assigned.

Key_Delete("Alt-A",NOERR)
Delete any key assignment to <**Alt-A**>.

Save / Restore Edit Position

A macro often needs to save the current edit position so that it can come back to it later. This could be done with the text markers, but this would interfere with markers that have already been set. It is often easier to use the **Save_Pos( )** and **Restore_Pos( )** commands.

Save_Pos( ) Save the current edit position on a special stack of "text markers". You can save up to 5 positions on the stack.

Restore_Pos( ) Restore the edit position from the most recent position saved with Save_Pos( ). If the stack is empty, the command has no effect.

Restore_Pos(RESET) Empty (reset) the edit position stack. Use this when your macro may have lost track of how many positions have been saved on the stack.

Technical Topics

File Buffering in Command Mode

File buffering in Command Mode is normally performed automatically. However, explicit read/write and file open/close commands — **File_Read( )**, **File_Write( )**, **File_Open_Read( )**, **File_Open_Write( )** and **File_Truncate ()** — give the experienced user additional flexibility. For example, these commands permit complex file splitting and merging by giving precise control over how many lines are read from and written to disk.

File_Open( ) always reads in the entire file from disk or until the edit buffer is nearly full. **File_Save( )**, **File_Close( )** and **Buf_Close( )** always perform all necessary reading and writing to properly save files regardless of their size.

Explicit Read/Write Commands

The command **File_Open_Read( )** opens a file for reading, but does not actually read anything in. The file can be read with **File_Read ( )**. Similarly, the command **File_Open_Write( )** opens a file for writing, but does not write anything out. Text can be written out with **File_Write( )**. Forward file buffering in Command Mode, therefore, can be done with successive **File_Read( )** and **File_Write( )** commands.

The **File_Read(*n*,REVERSE)** and **File_Write(*n*, REVERSE)** commands give you explicit control over backward file buffering. (See "Appendix A - File Management" in the VEDIT User's Manual.) The command **File_Write(*n*,REVERSE)** writes out '*n*' lines of text from the end of the edit buffer to the temporary "*.r*R$" file, creating it if necessary. Its main purpose is to make more memory space available for **File_Read(*n*, REVERSE)** which then reads back '*n*' lines of text that was written earlier to the output file.

CAUTION: Because of the complexity of the **File_Read( )** and **File_Write( )** commands, we suggest you not use them until you are thoroughly familiar with VEDIT's file handling. Generally, the **Begin_Of_File( )**, **End_Of_File( )**, **Line( )**, and **Search( )** commands can perform any additional file buffering you will need in Command Mode.

Although auto-buffering will make memory space available in the edit buffer for large insertions, you can also manually free additional memory with the **Mem_Free( )** command. It takes three forms:

Mem_Free(*n*)	Buffers out to disk until '*n*' bytes of memory are free.
Mem_Free(1)	Buffers out to disk until the current file is reduced in size to approximately 8 Kbytes.
Mem_Free	Only returns the number of bytes currently free in the edit buffer.

Mem_Free( ) does not buffer out any text which is within 2000 bytes of the edit position. If you specify too large a value, **Mem_Free( )** will not be able to make the total requested amount of memory free, but will make as much free as possible. The returned value is the actual number of bytes free. Or use **Mem_Status( )** to see how much memory actually is free.

Mem_Free(40000)	Perform forward and/or backward file buffering to make 40,000 bytes of memory free in the edit buffer (if possible).

Undo in Command Macros

Edit changes made in command macros can be undone (when enabled) with the commands **Undo_Delete( )**, **Undo_Edit( )** and **Undo_Line( )**, which are equivalent to the functions in the **{EDIT, Undo, Delete} menu.**

While the Undo facility is always enabled during Visual Mode, it can be fully or partially disabled in command macros with **Config(CM_E_UNDO, *n*)**. By default, you can undo all commands except for **Call( )** (macro execution).

The only reason to disable the Undo is to gain a little speed, which is not significant in Visual Mode. However, since most command macros will easily overflow even 1000 undo levels, nothing is lost by disabling Undo during macro execution.

Anytime the Undo is disabled, it will also reset itself because it is not possible to leave some editing steps out. Therefore if you completely disable Undo in command macros with **Config(CM_E_UNDO,0)**, each time you get the COMMAND: prompt, the Undo (in the current edit buffer) will be reset for both Command and Visual Modes.

In Command Mode, each command line executed is considered a single edit operation which can be undone with one **Undo_Edit( )** command.

Should any command sequence overflow the Undo buffer, the Undo will automatically disable itself for the duration of the command sequence. This gains speed and prevents you from undoing just a portion of the command sequence, which may not be meaningful.

Each "edit level" can undo one basic edit operation such as a **Line( )**, **Ins_Text( )**, **Del_Line( )** or **Search( )** command. However the **Replace( )** command uses three undo levels for each replacement. **Format_Para( )** uses three undo levels for each line in the paragraph. **Reg_Insert( )** with a columnar block uses a minimum of three undo levels per line inserted, and even more if spaces are converted to tabs. Therefore, it is possible that a single **Reg_Ins( )** command (or equivalent **{BLOCK, Register insert}** function) could overflow and reset the Undo facility. (A non-columnar **Reg_Ins( )** uses only one undo level.)

Any keystroke macro executed from Visual Mode that contains macro language commands can be undone (assuming enough levels are available), ***except*** those which contain a **Call( )** command. Also the **{MISC, Execute Macro}** and **{MISC, Load/Execute Macro}** functions, which internally use the **Call( )** and **Call_File( )** commands, can generally not be undone.

Search/Replace 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. (In practice it would be easier to use the supplied WILDFILE macro; however this example will give you an idea of how the WILDFILE macro works.)

NOTE: The following macro is supplied in the file **`multi-sr.vdm`**.

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 job, but it can be automated with a command macro. The macro begins with a main macro and is followed by a subroutine macro. The main macro contains the commands to edit each of the ten files and, for each file, execute the subroutine macro which contains the global search/replace commands for each of the 20 words. Once the macro is entered and begins executing, all 200 (10 times 20) operations are automatically performed.

SUGGESTION ALWAYS make a backup copy of the files before running complex macros. It is very easy for a small syntax error or other problem to destroy the files being processed!

For this example, the macro will be created in edit buffer 9. Use the **{FILE, Buffer switch}** function or **Buf_Switch( )** command to switch to edit buffer 9 and enter the following macro from Visual Mode: ("*word1*" is the first misspelled word and "*fix1*" is its correct spelling, etc.)

```
File_Open("file1.txt")
Call("fixup")
File_Close( )
File_Open("file2.txt")
Call("fixup")
File_Close( )
.......
File_Open("file10.txt")
Call("fixup")
File_Close( )
Return( )

:FIXUP:
Replace("word1","fix1",BEGIN+ALL+NOERR)
Replace("word2","fix2",BEGIN+ALL+NOERR)
Replace("word3","fix3",BEGIN+ALL+NOERR)
.......
Replace("word20","fix20",BEGIN+ALL+NOERR)
Return( )
```

It is a good idea to save newly entered macros to disk before using them because it is possible, due to a programming error, for a macro to erase itself. Once a macro works properly, you may also want to save it for future use.

➤ **To save the newly entered command macro to disk.**

1. Select **{FILE, Save as}**.
2. Enter the desired filename at the prompt. For this example, enter "globalsr.vdm".

The macro works by opening each file, one at a time, performing all search/replace commands, and writing the file back to disk. It then continues with the next file.

Each **Replace( )** command uses three important options:

BEGIN	Starts the search/replace at the beginning of the file. Otherwise, it would start wherever the previous command left the edit position.
ALL	Replaces all occurrences in the file. Otherwise it would only replace the first occurrence.
NOERR	Specifies that search errors are to be suppressed. Otherwise, if any word is not found the entire macro terminates.

You are now ready to execute the macro. While macros are usually loaded into text registers and executed, you can also execute a macro in an edit buffer. However, you cannot execute a macro in the current edit buffer; therefore, you will first have to switch to another buffer. Here are the commands:

Buf_Switch(1)	Switch to another buffer because a macro in the current buffer cannot be executed.
Call(9+BUFFER)	Execute the macro in buffer 9. The "+BUFFER" is needed to distinguish between text registers and edit buffers.

NOTE: The next topic "Multiple Replace in Huge Files" describes how to optimize this macro for speed when editing very large files.

Multiple Replace in Huge Files

As described in the previous topic "Search/Replace in Multiple Files", to perform a series of search and replace operations on a file, you could use the commands:

```
Replace("word1","fix1",BEGIN+ALL+NOERR)
Replace("word2","fix2",BEGIN+ALL+NOERR)
Replace("word3","fix3",BEGIN+ALL+NOERR)
.......
Replace("word20","fix20",BEGIN+ALL+NOERR)
```

While this will work with large files, you will notice some delay, particularly if you have many **Replace( )** commands.

The reason for the delay is that VEDIT has to read/write the entire file from disk for each search/replace operation. This is time consuming; on a 100+ megabyte file, each search/replace could take 10 minutes or more.

Although a bit more complex, a much faster way is to shuffle sections of the file through memory just once and perform all search/replace operations on each section. The following example illustrates this:

```
while (! AT_EOF) {
 File_Read(0)
 Replace("word1","fix1",BEGIN+ALL+NOERR+LOCAL)
 Replace("word2","fix2",BEGIN+ALL+NOERR+LOCAL)
 Replace("word3","fix3",BEGIN+ALL+NOERR+LOCAL)
.......
 Replace("word20","fix20",BEGIN+ALL+NOERR+LOCAL)
 File_Write(ALL)
}
```

NOTE: A fully documented and more up-to-date version of this macro is supplied as the file **`huge-sr.vdm`**.

The key difference in the **Replace( )** commands is the "LOCAL" option which restricts the search to the portion (section) of the file currently in memory. It also causes the "BEGIN" option to start the search at the beginning of the file section currently in memory. In other words, the "LOCAL" option prevents any disk read/write from being performed.

Explicit read/write command (see previous topic) perform all file buffering. The **File_Read(0)** command reads into memory as much of the file as will fit. **File_Write(ALL)** writes the entire file section in memory back to disk; the next **File_Read(0)** will then read the next section of the file into memory. The **While** loop repeats everything until all file sections have been processed and the end-of-file is reached.

HINT: When processing huge files with VEDIT, it will be much faster if the file resides on a local hard disk instead of on a network.

Event Macros

VEDIT has seven special macros that can automatically execute when a specific "event" occurs:

File-Open Configuration The macro in text register 115 is executed immediately after each file is opened. It is typically used to configure VEDIT according to the filename extension. It is set up by the `startup.vdm` file. This is fully described under "File-open Configuration" in Chapter 4 of the User's Manual.

File-Open The macro in text register 110 is executed after each file is opened and after the File-open configuration macro in register 115 is run. It can be used to automatically convert certain files whenever they are opened.

The four file-open/close macros must be enabled with **Config(F_E_F_MACRO)**.

File Close The macro in text register 111 is executed for each file that is closed. It is executed just before the file is saved to disk.

File Pre-Open The macro in text register 112 is executed just before each file is opened. It is typically used to shell out and inform a version control system that a file needs to be checked out.

File Post-Close The macro in text register 113 is executed immediately after each file is closed. It is typically used to shell out and inform a version control system that the modified file needs to be checked back in.

Buffer Switch The macro in text register 114 is executed immediately after each buffer switch in Visual Mode or the macro command **Buf_Switch(*r*,**EVENT).

Template Editing One or more Template-editing macros can be loaded into VEDIT. The macro configured for the current buffer is executed immediately after each text character that is entered in Visual Mode. It must be enabled with **{CONFIG, Programming, Enable template editing}** or **Config(PG_TEMPLAT)**.This is fully described under "Template Editing" in Chapter 4 of the User's Manual.

File-Open Event Macro

When multiple files are opened at once, e.g. invoking VEDIT with "**vpw *.c**", the File-open event macro will be executed for *each* file. Any **Config( )** commands in the event macros should include the "LOCAL" option so that they only affect the current edit buffer.

File-Close Event Macro

The File-close event macro is executed just before the file in the current buffer is saved to disk. It can be used to perform almost any desired operation. For example:

- Check that the last line ends in a "newline".
- Strip trailing spaces from the entire file.
- Translate the file. The File-open and File-close event macros can be used together to translate a file for easier editing and then translate it back.

For example, the following File-close event macro (to be placed into text register 111) checks if the last line ends in a "newline". If not, it prompts whether it should be inserted.

```
End_Of_File()
Char(-Newline_Chars)
if ( Match("|L") != 0 ) {
    End_Of_File()
    Update( )
    #109 = Get_Key('Insert missing "newline"? [Y]es [N]o ')
    if (#109=='Y') { Ins_Newline( ) }
}
```

The following single command File-close event macro strips trailing whitespace (spaces and tabs) from the file.

```
Replace("|W|>","",BEGIN+ALL)
```

File Pre-Open and Post-Close Event Macros

The File-pre-open" and File-post-close event macros are designed to facilitate checking files out of and back into a version control system.

When the File-pre-open event macro runs, the predefined string value **PATHNAME** is set to the name of the file about to be opened. The macro can then shell out to DOS and past this filename to the version control program. Similarly, when the File-post-close event macro runs, **PATHNAME** is still set to the name of the file just closed.

The following (contrived) example illustrates how the File-pre-open and File-post-close event macros could be used. (The DOS "attrib" command roughly simulates what a version control program does.) Assume that the **`ustartup.vdm`** file contains the following code to set up these macros:

```
Reg_Set(112,`
    System("attrib -r |(PATHNAME)",OK)
`)                  //Turn off Read-only attribute
Reg_Set(113,`
    System("attrib +r |(PATHNAME)",OK)
`)                  //Turn Read-only attribute back on
Config(F_E_F_MACRO,1)   //Enable file open/close event
macros
```

With this startup code, VEDIT can edit a file which has the Read-only attribute set. (We didn't say that this example was a good idea.) After the file is closed (saved to disk), it is set back to Read-only.

Template Editing Macro

The Template-editing macro is executed immediately after each normal text character is entered in Visual Mode. It is often used to expand a shorthand abbreviation, but it can be used for other purposes too.

To expand a shorthand abbreviation, the macro typically examines the last few characters in the text, including the character just entered, and if recognized, replaces the shorthand with the full expansion. To reduce incorrect expansions, the abbreviation must typically be surrounded by "separators" — non-alpha-numeric characters.

The **Match( )** command is typically used to examine the characters in the file. Alternatively, the internal value **Previous_Key( )** can be used to examine the last few keys pressed.

For example, the following Template-editing macro checks if the cursor immediately follows the four characters "if (". If true, this is expanded to:

```
if ( ) {
    ...
}
```

The cursor is placed inside the "( )". Note that the **Ins_Text( )** command inserts a multi-line string argument.

```
Char(-5)
if (Match("|Sif (")==0) {
    Char() Del_Char(4)
    Ins_Text("if ( ) {
    ...
}
")
    Char(-(12+3*Newline_Chars))
}
else { Char(5) }
```

Event Macro Programming Guidelines

Since these special event macros execute without being explicitly called, you must write them with additional care. It is easy to forget that they are executing automatically and consequently not understand why a simple command macro (or even keystroke macro) is not working properly.

It is particularly important to avoid conflicts between any text/numeric registers used by the event macros and other macros. For this reason we highly recommend that you set aside the following resources for exclusive use by event macros. This should nearly eliminate such conflicts. It also eliminates the need for the event macros to save/restore these registers. However, if the event macros need additional registers, they will have to be saved and restored, e.g. with the **Reg_Push( )** and **Reg_Pop( )** commands.

- Text registers 107 through 109.
- Numeric registers 107 through 109.
- "Extra" edit buffer Extra_Buffer_4.
  (Windows: buffer 103; DOS: buffer 36).

The following restrictions and guidelines should also be observed:

- Event macros cannot open files. Any required setup should be performed by the **`ustartup.vdm`** macro.
- Event macros should not open and close normal edit buffers. However, the "extra" edit buffers can be used.
- If an event macro switches between edit buffers and/or windows, it should switch back to the original buffer/window before returning.
- The Template-editing macro must execute fast enough so that there is little or no typing delay. If many abbreviations must be checked, the logic of the macro may need to be optimized. For example, if the last character is not a "space" the macro can immediately return.

Developing Complex Macros

Writing Macros in Edit Buffers

When developing complex macros, it is often easiest to enter and modify the macro as a normal file in an edit buffer. The macro can be executed in the edit buffer; there is no need to first copy it to a text register. However, two restrictions must be observed:

- The current edit buffer cannot be directly executed as a macro. Attempting it gives the error message "INVALID EDIT BUFFER OPERATION". To get around this, simply switch to another edit buffer that does not contain macro commands (usually the main buffer #1), before performing the **Call( )** command or **{MISC, Execute macro}** function.
- Although a macro generally can issue a **Buf_Switch(*b*)** command to edit another buffer, it may not issue the command if buffer '*b*' is itself an executing macro. Doing so stops execution and gives the error "MACRO ERROR IN *r*" where '*r*' is the register/buffer containing the offending **Buf_Switch( )** command.

Since the current edit buffer cannot be directly executed, the following keystroke switches to an "extra" buffer and then executes the original buffer. It then switches back.

This macro is listed in **`key-mac.lib`** which details how to add a keystroke macro to VEDIT. (Remember, it must be added as one line.)

```
[VISUAL EXIT]
#103=Buf_Num
Buf_Switch(Buf_Free(EXTRA))
Call(#103+BUFFER)
Buf_Switch(#103)
```

Alternatively, the macro could be added to the **{USER}** menu by adding the following three line to "user.mnu". (The commands must be on one line.)

```
1
Execute current buffer
#103=Buf_Num Buf_Switch(Buf_Free(EXTRA))
   Call(#103+BUFFER) Buf_Switch(#103)
```

Self-Modifying Command Macros

The above restrictions apply, because in general, VEDIT does not allow self-modifying macros. In other words, a macro may not modify the contents of its own text register.

A macro in one text register can perform a **Call( )** to macros (analogous to "subroutines") in other text registers. When this happens, the currently executing text register and the text register 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.) The rule is:

- A macro cannot alter the contents of any text register which is currently "executing", e.g. itself or a parent macro.

Attempting to violate this rule (see the exceptions below) results in the error message: "MACRO ERROR IN *r*" where '*r*' is the text register containing the offending command. When possible, the error message also displays the offending command.

Two exceptions allow a macro to empty the currently executing text register or a register containing a parent macro.

Reg_Empty(*r*,EXTRA)	Empty text register '*r*', even if it is executing.
Break_Out(DELETE)	Stop all macro execution and empty the currently executing text register. This is a convenient way to exit and delete a macro at the same time.

However, it is common for a macro to create or modify a macro in a text register which is not currently executing and, after modification, execute it. For example, a "main" macro will often load "subroutine" macros from disk, or enter them into text registers with the **Reg_Set( )** command.

Chaining to a Command Macro

The **Call(*r*)** command performs a "call" to the "subroutine" macro in register '*r*'. When the "subroutine" macro finishes, execution returns to any commands following the **Call( )**. As an alternative, the **Chain(*r*)** command will "chain" or "jump" to a macro in another register without returning.

Chain(*r*)	Chain (jump) to the macro in register '*r*' without "returning" to the current macro after '*r*' is done executing. Don't follow **Chain( )** with other commands — they won't be executed.

The **Chain( )** command is used in the supplied **`compare.vdm`** and **`sort.vdm`** macros. These macros are first loaded into one text register, generally register 100. (This can be done with auto-execution.) When executed, the macro first sets up all of the needed text registers using the **Reg_Set( )** command. Last, the macro uses the **Chain( )** command to "jump" to the register containing the main macro. Register 100 is then emptied from the main macro.

The similar **Chain_File( )** command first loads a macro from disk into a text register and then chains to it.

Chain_File(*r*,"*file*")	Load '*file*' into register '*r*' and chain (jump) to it; '*r*' can be the currently executing register. Don't follow **Chain_File( )** with other commands — they won't be executed.

The **Chain_File( )** command can be used to load and execute a succession of start-up macros using only a single text register, typically register 100.

Using the "Extra" Edit Buffers

The Windows version of VEDIT has 99 general purpose edit buffers and 26 "extra" buffers. (The DOS version has 32 edit buffers and four "extra" buffers.)

1 - 99 General purpose edit buffers. Each can be used to edit a file.

100 - 102 "Extra" edit buffers that can be used by any macro. They are generally not used to edit files, but are used as temporary buffers.

103 "Extra" edit buffer reserved for use by the File-open/close event macros and the Template-editing macros.

104 - 125 More "Extra" edit buffers that can be used by any macro.

We highly suggest referencing the first four "extra" buffers by their names **Extra_Buffer_1**, **Extra_Buffer_2**, **Extra_Buffer_3** and **Extra_Buffer_4**, or the abbreviations **XBUF1**, **XBUF2**, **XBUF3** and **XBUF1**.

Complex macros often need a few temporary edit buffers for manipulating blocks of text and performing operations too complex for text registers. You can use the normal (1 - 99) buffers for this, but the "extra" buffers have some advantages:

- The extra buffers cannot be accessed from Visual Mode. This protects them from accidental alteration.
- The extra buffers don't appear in the pick list for **{FILE, Buffer switch}**. This reduces confusion when a user is running a complex macro such as the compiler support.
- The **{FILE, Exit}** function and **Exit( )** command don't prompt whether these buffers should be saved. (They can't!)
- The extra buffers are available even if a file is open in every normal buffer.

To open a file for editing in an extra buffer, you must use the command form **File_Open(*file*,FORCE)**. The "**FORCE**" option is a safety feature because these buffers are not automatically saved.

Like all edit buffers, the extra buffers can be executed as command macros via the command **Call(*r*+BUFFER)**.

"Locked-in" Macros

When macro execution stops, either normally or due to an error, you are normally returned to the "COMMAND:" prompt. Alternatively, you can have the macro in a particular text register executed in place of the "COMMAND:" prompt. This macro is often some type of a menu. This feature is used in many of our supplied macros. It permits menu driven macros to be written in which the user will never see the "COMMAND:" prompt, or even need to know anything about VEDIT.

The command **Reg_Lock_Macro(*r*)** sets text register '*r*' to be the "locking-in" macro. Only register 0 (zero) cannot be selected. In practice, registers 99, 100 or 101 are often used as the locked-in macro.

Reg_Lock_Macro(99) Lock-in the macro in register 99 to execute in place of the "COMMAND:" prompt.

Reg_Lock_Macro(CLEAR) Disable the locked-in macro.

When a locked-in macro is enabled, anytime VEDIT would normally present the "COMMAND:" prompt, it instead executes the specified text register.

NOTES: A "locked-in" macro must be used with care because it is possible to get into an infinite loop that will require resetting the computer. Be sure to provide a method of exiting the macro and/or VEDIT. Pressing **[CANCEL]** (**<Ctrl-C>**) does not break out of a locked-in macro! It is best to thoroughly test the macro before inserting the **Reg_Lock_Macro( )** command.

To help avoid infinite loops, the locked-in macro is automatically disabled if syntax errors are encountered in a command macro. The command form **Reg_Lock_Macro(*r*,EXTRA)** disables this safety feature; however you are unlikely to ever need it.

On-Line Help and Web site

Remember to use the on-line help for more information about a particular command. It is constantly updated and describes every command and every command option, including any recent additions that are not in this manual.

It is organized differently from this manual and includes additional topics, and examples.

In the non-Windows versions of VEDIT, the on-line help file **`vphelp.hlp`** is just a text file; you can easily edit it and print any desired portions.

The "User Conference" on our web site at www.vedit.com has a section devoted to macro language questions. You can ask questions here which we or other users will answer.

Debugging Macros

Hopefully you will have the opportunity to write your own command macros. Of course, if you write complex macros, sooner or later you will have to debug a macro. The debugging of many macros is often obvious, involving just correcting typing errors or simple syntax errors.

Unfortunately, there will occasionally be macros, especially long and complex ones, whose flaws are not immediately visible. You know that a macro needs to be debugged when:

- You receive an error message while a macro is executing.
- There is no error message, but the results of the macro are not what you intended.

For those macros, VEDIT has several debugging features to help you. The most useful feature is the ability to single-step through the macro execution, tracing the commands one by one.

Many errors cause macro execution to stop and return you to the "COMMAND:" prompt. It may not be obvious which command caused the error. To display the most recently executed commands type "**??**" in response to the "COMMAND:" prompt. You can also press **[CANCEL]** (<**Ctrl-**\>) or <**Ctrl-C**> or <**Ctrl-Break**> to abort a macro and display the most recently executed commands.

?? Display 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 see its context. The message "(*r*)" is also displayed, where '*r*' is the text register containing the most recent macro commands. If no macro was executing, "( )" is displayed.

Trace Mode

When the "**??**" command is not enough to determine the flaw in a macro, you can trace (or single step) through the macro 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. (However, the "**?**" cannot appear within the arguments of a command.) In programming terminology, placing a "**?**" within the macro is called "*setting a breakpoint*". You can also set a "conditional breakpoint".

? Sets a breakpoint which enters trace mode when the "**?**" is encountered in the macro.

?(*expr*) Sets a conditional breakpoint — if the expression '***expr***' is TRUE, it enters Trace mode; otherwise the macro continues running normally. The "(" must immediately follow the "?". '***expr***' is usually a conditional expression such as "**#1>200**" or "**!At_BOF**", but it can also be the return value from a com-

mand, such as "**File_Exist(`output.tmp`)**". You can think of "**?(*expr*)" as a shortcut for "if(*expr*) { ? }".**

Examples are:

?(#1200) Enter Trace mode if numeric register 1 has a value of greater than 200.

?(File_Exist(`output.tmp`))

Enter Trace mode if the file "output.tmp" exists.

You can trace from the beginning of the macro by entering the following command at the "COMMAND:" prompt:

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

When "**?**" is encountered, Trace mode is turned on and one command at a time is executed. Before each command is executed, it is displayed on the screen, followed by a list of each of its evaluated arguments. For each command, you can control the tracing process by pressing one of the following keys:

<Enter> Process the current command and remain in Trace mode. This "single steps" through the commands.

<Space> Same as **<Enter>** unless the current command is a **Call( )**, in which case Trace mode is disabled while the "subroutine" macro is executed; Trace mode is resumed when the macro terminates. (Any "**?**" encountered in that macro will be ignored). This allows skipping over a **Call( )** command.

U Toggles the special "Update" mode on and off. When on, an **Update( )** command is performed after tracing each command. This is useful when the same editing window is used for both displaying the buffer and for tracing. This Update mode is usually not need if you are tracing with the special Command Mode window.

V Enters Visual Mode to view the contents of the edit buffer(s). If desired, you can also make any edit changes. Use **[VISUAL EXIT]** (**<Ctrl-E>**) to return and continue tracing; use **[VISUAL ESCAPE]** (**<Ctrl-Shift-E>** or **<Alt-F10>**) to abort and return to the "COMMAND" prompt.

From Visual Mode use **{HELP, Text registers}** to view the contents of the registers. If you switch to other edit buffers, be sure to switch back to the original buffer before returning.

$** Creates the special Command mode window "$" as a 5-line reserved window at the bottom of the screen, and continue tracing in that window. Similar to selecting **{ESCAPE, Command mode window}.

<Esc> Turns Trace mode off and resumes normal execution with the current command.

[CANCEL] (**<Ctrl-\>** or **<Ctrl-C>**) Aborts processing and turns the Trace mode off. Returns to the "COMMAND:" prompt.

?	Displays the next command line following the command about to be traced. Each additional "**?**" displays an additional line. Lets you preview the upcoming commands.
-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.

- Did you use the assignment "**=**" when you meant to test for equality with "**= =**"? This is a very common mistake in VEDIT (and C). For example, the statement "**if (#10=123) { ... }**" is meaningless — it is always true; perhaps you meant to have "**if (#10==123) { ... }**".
- Check for missing/incorrect string delimiters. This could allow commands to be interpreted as text, or text to be interpreted as commands.
- Is the end-of-file condition being handled correctly? Is it a special case? If it is, are you testing for the end of file properly? The internal value **At_EOF** is useful here.
- Do your search operations handle an unsuccessful search properly? Are the "NOERR" and "ERRBREAK" search options being used correctly?
- Be sure not to confuse **If-then (else)** statements with loops if you are using the **Break** command or the "ERRBREAK" command option.
- Are the "special event" macros interfering? You may want to disable **{CONFIG, Programming, File-type specific config}** and **{CONFIG, Programming, Enable template editing}** to be sure.
- Are other features set up by the **`startup.vdm`** file interfering? You may want to invoke VEDIT with the "-ixxx" and "-g" options to disable the **`startup.vdm`** macro and use the default configuration settings.

  vpw -ixxx -g
- Remember that the Windows/DOS "newline" consists of the two characters Carriage-Return and Line-Feed; the UNIX "newline" consists of just a Line-Feed; the Mac "newline" consists of just a Carriage-Return. The internal value **Newline_Chars** should be used when a macro must know how many characters there are in a "newline".
- Avoid using the braces "**{**" and "**}**" in string arguments or comments. See the topic below.
- 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".

Using "{" and "}" in String Arguments

Since the macro language is interpreted (not compiled), VEDIT must often scan for the "**{**" and "**}**" braces to process flow control statements. For the sake of speed, a simple scan is made and extra braces that occur in string arguments or comments confuse the process.

For example, the following macro, while perfectly valid, will confuse VEDIT:

```
if (#80==0) {                          This macro confuses VEDIT.
    Ins_Text("Inserting a }")
}
```

When the **if** condition is false, VEDIT will attempt to skip the "then" statement by scanning for the closing "**}**". Unfortunately, the "**}**" in the argument for the **Ins_Text( )** command will be used. The following ")" will then give an "Invalid Command" error.

There are two easy solutions to this problem:

```
if (#80==0) {                          Preferred solution.
    Ins_Text("Inserting a ")
    Ins_Char(125)
}

if (#80==0) {                          An alternative solution.
    //Comment needed to match - {
    Ins_Text("Inserting a }")
}
```

In the first and preferred solution, the "**}**" is removed from the string argument and inserted into the text with the **Ins_Char( )** command. In the second solution, a comment is used to supply a matching brace that offsets the brace in the following string argument.

Only unmatched braces in string arguments will cause problems. Therefore, the following macro will cause no problems:

```
if (#80==0) {                          This macro causes no trouble.
    Ins_Text("Matching { } are OK")
}
```

Cleanup/Converting Macros

Most macro language commands have both a full name and an abbreviated name. The supplied macro **cmd-conv** can convert a command macro ".vdm" file between these two forms. Commands with full names are generally easier to read, while abbreviated names can save memory space, especially for macros that are part of keystroke macros.

cmd-conv.vdm can also convert (reasonably sized) command macros into the one-line format needed for keystroke macros.

➤ **To run CMD-CONV from within VEDIT:**

1. Open the macro that you wish to convert as a normal file.

   If desired, use **{FILE, Save as}** to save the converted macro under a different name.

2. Select **{MISC, More macros, CMD-CONV}**.. The macro then displays a menu of the operations it can perform:

```
                VEDIT MACRO COMMAND CONVERSION

  [1] Convert Full Commands To Full Commands; Just clean up
  [2] Convert Full Commands To Abbreviated Commands
  [3] Convert Abbreviated Commands To Full Commands
  [4] Convert Macro To A One Line Keystroke Macro
  [5] EXIT, Do Not Convert
```

Item "[1]" cleans up a VEDIT macro by converting all commands to a uniform syntax with only the first letter of each command word capitalized.

Item "[2]" converts all commands to their shortest possible abbreviation. Item "[3]" converts all abbreviated commands to their full name.

Item "[4]" converts a command macro into a one line macro that can be used as a keystroke macro. It precedes the macro with "[VISUAL EXIT]" and strips all comments and unneeded whitespace to save memory space. It also converts all commands to their shortest possible abbreviation. This converted macro can then be pasted into a VEDIT.KEY file for loading as a keystroke macro.

Although VEDIT supports keystroke macros that are individually up to 4000 characters long, we don't recommend keystroke macros longer than about 500 bytes. A keystroke macro can use **Call_File( )** to call larger macros.

Notes: The topic "VEDIT.KEY Layout File" in Chapter 8 of the VEDIT User's Manual describes how to create and edit a VEDIT.KEY file.

cmd-conv.vdm should not be considered foolproof. Be sure to keep a backup of any macro that you convert until you are sure that it was correctly converted.

We would like to thank Peter Freed of Data Base Management Systems, Inc. for writing **cmd-conv.vdm** and sharing it with us.

Preserving Your Files

VEDIT is designed to make it as unlikely as possible for you to accidentally lose a file or your edited text. However, nothing is foolproof. The best safeguard is to save your edited file to disk at least once an hour using **{FILE, Save}** or **{FILE, Save all}** and backing up your work at the end of each day.

You can have VEDIT automatically perform **{FILE, Save all}** for you on a regular basis by setting **CONFIG, File handling, Auto-save interval}** to the desired interval in minutes. We highly recommend using this feature.

You should use the following commands with additional care:

- The **{MISC, DOS shell}** function and equivalent **System( )** command let you enter DOS for as long as you wish. However, it does not save the files you are currently editing. Therefore, be sure to return to VEDIT (with the DOS command "exit") to save your files. It is easy to forget that you are still in VEDIT and turn the computer off — losing your edit changes. As a safeguard, you should save your files using **{FILE, Save all}** or the equivalent command **File_Save(ALL)** before entering DOS.
- The **Buf_Empty( )** and **Buf_Quit** commands abandon the edit changes you have just made. They are very useful, but you don't want to use them when you really wanted to save your changes. These commands request confirmation before abandoning your text. Also, they do not change or erase the original files on disk.
- The **File_Delete( )** command deletes files similar to the DOS "DEL" command. **File_Delete( )** is actually safer to use since it displays the filenames about to be erased and requests confirmation.
- The rarely used **File_Truncate( )** command is primarily intended for splitting large files into smaller ones. **File_Truncate** is almost always preceded by **File_Write( )**. Don't use **File_Truncate( )** to quit or save your editing. This command requests confirmation.

As an additional safeguard, VEDIT normally creates a "backup" when you modify an existing file. The backup can be created by either copying the original file to a backup directory, e.g. "**c:\vedit\backup**", or by renaming the original file with a filename extension of ".BAK" in the same directory.

Therefore, if you accidentally erase an important file, you will still have the backup (unless you explicitly erased it). To use the backup, just rename it to the desired filename; it is best not to directly edit a ".BAK" file.

If desired, you can disable VEDIT's "backup" feature by setting **{CONFIG, File handling, Enable backup files}** to "0". However, we highly recommend that you keep the "backup" feature enabled.

Chapter 4

Command Reference

Alert( ) / Sound(*n*,*k*)

Options: EXTRA

Usage: Alert( ) Sound(400,500)

Description: **Alert( )** sounds a "beep" on the IBM PC speaker. The command can be used inside a command macro to alert the user to an upcoming prompt, an error or other special condition.

Sound(*n*,*k***)** generates a sound (tone) of frequency '*n*' hertz and duration '*k*' milliseconds. **Sound(***n*,*k*,**EXTRA)** adds 30 milliseconds of silence after the sound. (DOS and Windows 95/98 only)

Notes: **Alert( )** can be forced silent by setting **{CONFIG, Misc, Beep level}** or **Config(U_BEEP_LVL)** to "0".

Examples:

Alert( )	Create a "beep" on the IBM PC speaker.
Sound(262,1000)	Plays the note "middle C" for one second.

APPH App_Height (Windows only)
APPW App_Width
APPX App_X_Org
APPY App_Y_Org

Usage: Internal Values

Description: **App_Height** and **App_Width** return the size of the VEDIT application (program) window in pixels.

App_X_Org and **App_Y_Org** return the horizontal and vertical pixel position of the upper left-hand corner of the VEDIT application (program) window. The upper left corner of the screen is 0,0.

See Also: Commands: Desktop_Height, Desktop_Width, Win_Height, Win_Width, Win_X_Org, Win_Y_Org, Win_Move( )

Examples:

Win_Move(APP,App_X_Org+10,App_Y_Org,APPW,APPH)

Move the entire VEDIT program to the right by 10 pixels.

At_BOB / At_BOF / At_BOL
At_EOB / At_EOF / At_EOL

Usage: Internal Values

Description: **At_BOB** is the beginning-of-buffer flag. It returns 1 if the edit position is at the beginning of the portion of the file currently in memory. Otherwise, it returns 0.

At_BOF is the beginning-of-file flag. It returns 1 if the edit position is at the beginning of the file. Otherwise, it returns 0.

At_BOL is the beginning-of-line flag. It returns 1 if the edit position is at the beginning of a line (or record). Otherwise, it returns 0.

At_EOB is the end-of-buffer flag. It returns 1 if the edit position is at the end of the portion of the file currently in memory. Otherwise, it returns 0.

At_EOF is the end-of-file flag. It returns 1 if the edit position is at the end of the file. Otherwise, it returns 0.

At_EOL is the end-of-line flag. It returns 1 if the edit position is at the end of a line (or record) or at the end of the file. Otherwise, it returns 0.

Return: Always returns a value of 1 or 0 (1 = TRUE, 0 = FALSE).

Examples: **if (At_EOF) {Break_Out}**

Test if the edit position is at the end-of-file. If it is, stop the macro and return to Command/Visual mode.

Atoi(*r*)
NER Num_Eval_Reg(*r*)

Options: SUPPRESS

Usage: #11 = Atoi(10)

Description: **Atoi(*r*)** evaluates the numeric expression in text register '*r*' and returns its value. It performs an ASCII to Integer conversion.

The command evaluates not only simple integers, but also numeric expressions. For example, if text register 9 contains the text "123+45/5", then **Atoi(9)** will return 132.

Atoi(*r*,SUPPRESS) evaluates only the first unsigned integer. For example, if text register 9 contains the text "123+45/5", then **Atoi(9,SUPPRESS)** will return 123.

Return: Returns the value of the numeric expression evaluated. **Chars_Matched** is set to the number of characters in the expression.

Notes: **Atoi( )** and **Num_Eval_Reg( )** are two names for the same command.

See Also: Commands: Itoa( ), Num_Eval( ), Name_Dir( )

Examples:
```
Get_Input(10,"Enter number: ")
#11 = Atoi(10)
```
Prompt the user to enter a number; convert it from ASCII and place it in numeric register 11.

BOF Begin_Of_File( )
BOL Begin_Of_Line( )

Options: LOCAL

Usage: Begin_Of_File( ) Begin_Of_File(LOCAL) BOL( )

Description: **Begin_Of_File( )** moves the edit position to the beginning of the file. **Begin_Of_File(LOCAL)** only moves the edit position to the beginning of the portion of the file currently in memory.

Begin_Of_Line( ) moves the edit position to the beginning of the current line (or record). If the edit position is already at the beginning of the line, the command is ignored.

Notes: **{GOTO, Beginning of File}** is identical to **Begin_Of_File( )**.

See Also: Commands: File_Save( ), End_Of_File( )
Topic: Backward File Buffering

Examples:
```
Begin_Of_File(LOCAL)
Type(12)
```
Move the edit position to the beginning of the edit buffer and type the first 12 lines.

BB / BE Block_Begin(*n*) / Block_End(*n*)
CB / CE Column_Begin(*n*) / Column_End(*n*)

Options: CLEAR, EXTRA

Usage: Block_Begin(123) CE(Cur_Pos)

Description: **Block_Begin(***n***)** sets the block-begin marker to file position '*n*'.

Block_End(*n***)** sets the block-end marker to file position '*n*'.

Block_End(CLEAR) clears the block-end marker while **Block_Begin(CLEAR)** clears ***BOTH*** the begin and end markers. These are the same block markers used in Visual Mode by the **{BLOCK}** functions.

Block_Begin(CLEAR+EXTRA) only clears the markers if **{Config, Emulation, Block marker emulation mode}** or **Config(E_BM_MODE)** is set to "0". Otherwise, the command is ignored and the block markers are unchanged.

Column_Begin(*n***)** sets the block-begin marker's column to '*n*'. Similarly, **Column_End(***n***)** sets the block-end marker's column to '*n*'.

Return: Without an argument, **Block_Begin** and **Block_End** return the file position (offset) of the current block markers, or -1 if not set.

Column_Begin and **Column_End** return the column position of the current block markers, or 0 if not set.

See Also: Commands: Set_Marker( )
{BLOCK} menu
Topics: "Block Operations" in Chapter 3

Examples:

Block_Begin(123)	Set the block-begin marker at the 124th character in the file. (Counting starts at 0).
BE(Cur_Pos)	Set the block-end marker at the current character.

BCP Block_Copy(*p,q*)
BMV Block_Move(*p,q*)

Options: Listed below

Usage: Block_Copy(123,456) Block_Move(BB,BE,COLUMN)

Description: **Block_Copy(***p,q***)** copies a block of text in the same buffer (file) to the current edit position. The block consists of the text starting with the '*p*'th character in the file up to, but not including the '*q*'th character. The edit position is advanced past the inserted text.

Block_Copy(*m***)** duplicates the following '*m*' lines, while **Block_Copy(-***m***)** duplicates the preceding '*m*' lines.

Block_Move( ) is similar, but also deletes the original block of text after it is copied.

Block_Copy/Move Command Options:

COLUMN	Performs a columnar block copy/move. '*p*' and '*q*' define the corners of the block.
COLSET, *c1***,** *c2*	Perform a columnar block copy/move. '*p*' and '*q*' define the first and last lines of the block, while '*c1*' and '*c2*' define the first and last columns of the block.
*l1***,** *l2***,LINESET**	Copy the block consisting of lines '*l1*' through '*l2*' (inclusive) to the edit position.
LINESET+ COLSET	Copy the columnar block consisting of lines '*p*' through '*q*' and columns '*c1*' through '*c2*' to the edit position.
BEGIN	Forces the edit position to remain at the beginning of the block of text that was copied/moved; otherwise it advances past the end of the block.
EXTRA	Copy the block; the block and column markers are either reset, maintained or moved to the new block, depending upon the setting of **{CONFIG, Emulation, Block marker emulation mode}**. Similar to **[BLOCK COPY]**.

OVERWRITE Overwrite the characters at the edit position with the block of text being copied. The edit position is advanced past the copied block.

DELETE **Block_Copy(*p*,*q*,DELETE)** is identical to **Block_Move(*p*,*q*)**.

DELETE+FILL Copy (move) the block; then replace (fill) the original block with spaces.

Notes: **Block_Copy(*p*,*q*)** allows copying text without using a text register. It is similar to the **{BLOCK, Copy to cursor}** function.

See Also: Commands: Reg_Copy( ), Reg_Ins( )
Topics: "Block Operations" in Chapter 3

Examples: **Block_Copy(123,456)**

Copy the block of text between the 123rd and 456th positions in the file to the current edit position.

Block_Copy(BB,BE,COLUMN)

Copy the columnar block of text between the block-begin and block-end markers to the edit position. The file position of the block-begin and block-end markers specify the "corners" of the columnar block.

Block_Copy(100,200,LINESET+COLSET,20,60)

Copy the columnar block consisting of lines 100 through 200 and columns 20 through 60 to the edit position.

BE Block_End - See Block_Begin

BFL Block_Fill(*ch,p,q*)

Options: COLUMN, COLSET, INSERT, LINESET, RESET

Usage: Block_Fill(' ',123,456) Block_Fill('x',BB,BE,COLUMN)

Description: **Block_Fill(*ch,p,q*)** overwrites the block of text between file positions '*p*' and '*q*' with character '*ch*'. The edit position is not moved.

If '*ch*' is a space and **Config(E_EXP_TAB)** is set to "0" - "3", the filling is done with the optimum number of tabs and spaces. Otherwise, only spaces are used.

The command options "**COLUMN**", "**COLSET**" and "**LINESET**" specify a columnar or line-range block for filling.

Other Block_Fill Command Options:

NORESTORE The edit position is moved just past the end of the block.

RESET The Block_Begin and Block_End markers are cleared.

INSERT Insert '*q*' - '*p*' instances of '*ch*' at file position '*p*'. This inserts a block consisting of fill characters.

INSERT+ COLUMN Insert a columnar block consisting of fill characters '*ch*'. File positions '*p*' and '*q*' define the "corners" of the columnar block to be inserted. This is equivalent to **{BLOCK, Edit/translate, Insert empty block}**.

See Also: Commands: Block_Copy( ), Reg_Copy( )
Topics: "Block Operations" in Chapter 3

Examples: **Block_Fill(' ',20,30,LINESET+COLSET,9,16)**

Blank out columns nine through sixteen on lines twenty through thirty (inclusive).

BM Block_Mode

Options: CLEAR, COLUMN, LINEBLOCK

Usage: Block_Mode Block_Mode(CLEAR) Block_Mode(COLUMN)

Description: **Block_Mode** returns the type of block mode set for Visual Mode: 0="stream" mode, COLUMN="column" mode, LINEBLOCK= "line" mode.

Block_Mode(CLEAR) sets "stream" mode for blocks in Visual Mode. **Block_Mode(COLUMN)** sets "column" mode; **Block_Mode(LINEBLOCK)** sets "line" mode.

Notes: **Block_Mode** does not affect how blocks are processed by macro commands, it only affects how blocks are highlighted and processed in Visual Mode. Its primary use is inside keystroke macros.

BMV Block_Move( ) - See Block_Copy( )

BSA Block_Save_As("*file*",*p,q*)

Options: COLUMN, COLSET, LINESET, OK

Usage: Block_Save_As("file1",0,Cur_Pos)
Block_Save_As("file2",#10,#20,COLUMN)

Description: **Block_Save_As("***file***",***p,q***)** copies the block of text starting with the '*p*'th character up to, but not including, the '*q*'th character to the file. Counting starts with 0. The text in the edit buffer is unchanged.

If '*file*' already exists, a confirmation prompt to overwrite the file is given; **Block_Save_As(...,OK)** skips the confirmation prompt.

The command forms **Block_Save_As("***file***",***p,q***,COLUMN)** and **Block_Save_As("***file***",***p,q***, COLSET,***c1,c2***)** copy columnar blocks to the file. The operations of these options are explained in the **Reg_Copy_Block( )** command. During the copy, Tab characters are expanded to spaces and short lines are padded with spaces — all lines in the resulting file will have the same length.

Use the closely related **Write_Line( )** to write lines of text to a file.

Notes: Although VEDIT's text registers are limited in size (64K or 256K), **Block_Save_As( )** can write blocks of any size to disk. These blocks can then be inserted into the same or another file with **Ins_File( )**. In this way, you can use files as unlimited size "text registers".

See Also: Commands: Clip_Copy_Block( ), Ins_File( ), Reg_Copy( ), Reg_Copy_Block( ), Write_Block( )
Topics: "Block Operations" in Chapter 3

Examples: **Block_Save_As("part1",0,Cur_Pos)**

Write the block from the beginning of the file up to the current edit position to the file "part1".

BOL_Pos
EOL_Pos

Usage: Internal values

Description: **BOL_Pos** returns the file position at the beginning of the current line. **EOL_Pos** returns the file position at the end of the current line, just before the newline character(s).

These internal values make is easier to perform block operations on the current line.

See Also: Commands: Begin_Of_Line(), End_Of_Line(), Cur_Pos
"Block Operations - Determining File Position" in Chapter 3.

Examples: **Reg_Copy_Block(9,BOL_Pos,EOL_Pos)**

Copy the current line without the final newline character(s) to text register 9; the edit position can be anywhere on the line.

Break
Break_Out

Options: EXTRA

Usage: if (Error_Flag) { Break } if (Error_Flag) { Break_Out }

Description: **Break** exits the current **While**, **Do-while**, **For**, or **Repeat** loop and continues with any commands following the loop's ending "**}**".

Break only exits the innermost enclosing loop. In the case of nested loops, any outer loops will continue executing. If there is no enclosing loop, it does nothing. **If-then (else)** statements do not effect **Break**.

Break_Out stops all command macro execution and returns to the "COMMAND:" prompt; if a "locked-in" macro is enabled, this macro executes in place of the "COMMAND:" prompt. (**Break_Out** internally performs a **Visual_Macro(CLEAR)** to disable auto-returning to Visual Mode.)

Break_Out(EXTRA) stops all macro execution and returns to Visual Mode. (It internally performs a **Visual_Macro(8)** to force a return to Visual Mode.)

Break_Out(EXTRA+CONFIRM) returns to Visual Mode with a "Press any key to continue" prompt if the window contains Command Mode output. (It internally performs a **Visual_Macro(8+4)**.)

Break_Out(DELETE) stops all macro execution and empties the currently executing text register. This is a convenient way to exit and delete a macro at the same time.

See Also: Commands: Return( ), Continue, Goto, Visual_Macro( )
Topics: "Flow Control - Break-out Commands" in Chapter 3

Browse_Mode(*n*)

Options: CLEAR, SET

Usage: Browse_Mode(SET) Browse_Mode

Description: **Browse_Mode(SET)** enables browse mode, thereby not allowing any modifications of the file in the current edit buffer. **Browse_Mode(CLEAR)** disables browse mode (when possible), allowing full editing operation.

Browse_Mode, without parameters, only returns the value of the browse mode switch in the current edit buffer.

Return: Returns 0 (zero) if the current edit buffer is not in browse mode; otherwise, the following masks (in hexadecimal) indicate why the buffer is in browse mode:

80H: "-b" invocation option
20H: File currently open in browse-only mode
10H: Disk currently open in browse-only mode
08H: File has read-only attribute
02H: File was opened with File_Open("file",BROWSE)
01H: **{FILE, Browse mode}** or **Browse_Mode(SET)** enabled

BC Buf_Close

Options: ALL, CONFIRM, DELETE, EVENT, MAINBUF, NOEVENT, NOMSG

Usage: Buf_Close( ) BC(NOMSG) Buf_Close(ALL)

Description: **Buf_Close( )** saves and closes the file being edited in the current edit buffer. It also exits the current edit buffer, unless it is the last open buffer (at least one buffer is always open). If the edit buffer has been altered, but has no assigned filename, it prompts for one.

Buf_Close(ALL) saves and closes all files/buffers being edited. Use this with caution because it only saves those edit buffers that have

assigned filenames. It also saves all buffers/files without prompting before each is saved.

Buf_Close( ) normally gives the message "Saving: *filename*"; this message can be suppressed with **Buf_Close(NOMSG)**.

Buf_Close(CONFIRM) prompts whether a modified buffer is to be saved. If the user selects [No-abandon], it performs a **Buf_Quit( )** instead.

Buf_Close(DELETE) deletes all windows attached to the file(s) being closed.

If **Config(F_E_F_MACRO,1)** is enabled, the "File Close" and "File Post-Close" event macros will also run. **Buf_Close(NOEVENT)** suppresses the file-close event macros, even if enabled. **Buf_Close(EVENT)** executes the file-close event macros, even if disabled.

When editing multiple files, **Buf_Close( )** will switch to the next open edit buffer. However, in practice it may be difficult to predict which buffer this will be. Follow the **Buf_Close( )** command with **Buf_Switch( )** to switch to the desired edit buffer. **Buf_Close(MAINBUF)** always switches to the main edit buffer.

Notes: **Buf_Close(CONFIRM+DELETE)** is equivalent to the **{FILE, Close}** function.

An unaltered file is not rewritten to disk.

While a "File Open" event macro is commonly used, "File Close" event macros are rarely used.

The error "NO DISK SPACE" results if there is insufficient disk space to save the entire file.

See Also: Commands: File_Close( ), File_Save( ), File_Open( ), Buf_Quit( ), File_Open_Write( ), Buf_Empty( ), Is_Altered, Is_Open_Write
{FILE} menu
Topics: "Event Macros" in Chapter 3

Examples:

```
File_Open("file.txt")
...
Buf_Close( )
```

A file is opened for editing. Any desired editing is performed. The modified file is then saved to disk.

BY Buf_Empty
FQ File_Quit

Options: EVENT, OK, NOEVENT

Usage: Buf_Empty( ) BE(OK)

Description: **Buf_Empty( )** quits (abandons) the current file, without saving any changes. However, unlike **Buf_Quit( )**, it stays in the same edit buffer and is often followed by a **File_Open( )** command to edit a new file.

You are prompted for confirmation to abandon the file if it was altered. **Buf_Empty(OK)** skips the confirmation prompt.

If **Config(F_E_F_MACRO,1)** is enabled and a file is open in the buffer, the File-close event macro in T-Reg 111 is executed just before the file is abandoned and closed. **Buf_Empty(NOEVENT)** suppresses the file-close event macro, even if enabled. **Buf_Empty(EVENT)** executes the file-close event macro, even if disabled.

Buf_Empty( ) and **File_Quit( )** are different names for the same command. If a file is open in the buffer, it is more intuitive to use **File_Quit( )**; if the buffer is only being uses as a scratchpad, it is more intuitive to use **Buf_Empty( )**. However, you can use either command.

Notes: The notes for the **Buf_Quit( )** command apply.

See Also: Commands: Buf_Close( ), Buf_Quit( ), File_Close( )

BF Buf_Free
BX Buf_Next
BN Buf_Num
BNA Buf_Num_Altered
BNW Buf_Num_Window
BPV Buf_Previous
BSTAT Buf_Status(*r*)
BT Buf_Total
BW Buf_Window
BWN Buf_Win_Next
BWP Buf_Win_Previous

Usage: Internal Values

Description: **Buf_Free** returns the ID number of the next available free edit buffer in the range 1 - 99. **Buf_Free(EXTRA)** returns the ID number of the next free "extra" buffer in the range 100 - 125. Returns "-1" if none available.

Buf_Next returns the ID number of the next open edit buffer (which is not currently executing as a command macro). Does not include the "extra" buffers 100 thru 125. Returns the current buffer's ID number if only one buffer is open.

Buf_Num returns the ID number of the current edit buffer: 1 - 125.

Buf_Num_Altered returns the total number of altered edit buffers. It includes edit buffers that have no assigned filename. It does not include any "extra" buffers 100 thru 125.

Buf_Num_Window returns the number of windows attached to the current edit buffer.

Buf_Previous returns the ID number of the previous open edit buffer (which is not currently executing as a command macro). Does not include the "extra" buffers 100 thru 125. Returns the current buffer's ID number if only one buffer is open.

Buf_Status(*r*) returns the status of edit buffer '*r*':

-1	'*r*' is not open
0	*r*' is open, but is not attached to any window
'*n*'	'*r*' is attached to window '*n*'

Buf_Total returns the total number of open edit buffers. It includes edit buffers that have no assigned filename. It does not include the "extra" buffers 100 thru 125.

Buf_Window returns the current edit buffer's primary attached window ID number. This is the window in which the current buffer is edited when in Visual Mode. Returns 0 if the buffer is not attached to any window.

Buf_Win_Next returns the ID number of the next window attached to the current edit buffer. If only one window is attached, it returns the ID number of the primary window. See also **Win_Next**.

Buf_Win_Previous returns the ID number of the previous window attached to the current edit buffer. If only one window is attached, it returns the ID number of the primary window. See also **Win_Previous**.

Examples:

Buf_Switch(Buf_Next) Switch to the next open edit buffer.

Win_Switch(Buf_Win_Next,ATTACH)
Switch to the next window attached to the current edit buffer and make it the primary window.

BQ Buf_Quit

Options: ALL, DELETE, MAINBUF, OK

Usage: Buf_Quit BQ(OK) Buf_Quit(ALL)

Description: **Buf_Quit( )** quits (abandons) the current file without saving any changes. It also closes the edit buffer (except for the main edit buffer). You are prompted for confirmation to abandon the file if it was altered. **Buf_Quit(OK)** skips the confirmation prompt. **Buf_Quit(ALL)** quits all edit buffers (abandons all files) after prompting for confirmation. **Buf_Quit(ALL+OK)** skips the confirmation to abandon all files; use with caution!

When editing multiple files, **Buf_Quit( )** will switch to the next edit buffer. However, in practice it may be difficult to predict which buffer this will be. Follow the **Buf_Quit( )** command with **Buf_Switch( )**

to switch to the desired edit buffer. **Buf_Quit(MAINBUF)** always switches to the main edit buffer.

Buf_Quit(DELETE) also deletes all windows attached to the edit buffer being closed.

Notes: **Buf_Quit( )** is often used after examining a file that you don't want to change. In this case it is safer to quit, rather than exit with **Buf_Close( )**, in case you accidentally did change something.

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.

If you quit with a **Buf_Quit( )** sometime after a **File_Save( )** command, you will only abandon those changes made after the **File_Save( )** command. Those changes made before the **File_Save( )** command will have already been saved on disk.

See Also: Commands: File_Save( ), Buf_Empty( )

Examples:

```
File_Open("savefile.1")
...
Buf_Quit( )
```

You only want to examine the file without changing it. When done, you quit to leave the file unchanged.

BSTAT Buf_Status( ) - See Buf_Free

BS Buf_Switch(*r*)

Options: ATTACH, EVENT, EXTRA, LOCAL, NOMSG, SUPPRESS

Usage: Buf_Switch(4) BS(20,SUPPRESS) Buf_Switch(#1)

Description: **Buf_Switch(***r***)** switches to edit buffer '*r*' and the file which may be open in '*r*'. If buffer '*r*' is not already open, it will be opened as an empty buffer.

Buf_Switch(*r***,ATTACH)** switches to edit buffer '*r*' and creates a new overlapping window if the buffer is not already attached to a window. The window is only created if **{CONFIG, Screen display, Auto-create windows for buffers}** is enabled.

Normally, with sufficient memory, each edit buffer has its own memory segment(s). However, with limited memory, buffers may share a memory segment. When switching from one buffer to another in the same memory segment, the old edit buffer is reduced in size to approximately 8 Kbytes. **Buf_Switch (***r***,LOCAL)** switches to edit buffer '*r*' without performing any file buffering.

Buf_Switch(*r***,SUPPRESS)** prevents VEDIT from grabbing more memory when opening a new buffer. This form should be used when the edit buffer is used for purposes other than editing an entire file.

Buf_Switch(*r***,EXTRA)** causes VEDIT to grab additional memory when switching to a new "Extra" buffer (33 - 36). Otherwise, "extra" buffers share existing memory.

Buf_Switch(*r*) executed from the "COMMAND:" prompt displays the message "Editing buffer *r*"; **Buf_Switch(*r*, NOMSG)** suppresses the message. However, when executed from within a command macro, the message is automatically suppressed.

Buf_Switch(*r*,EVENT) executes the special "Buffer switch event macro" in register 114 immediately after the switch is made.

Return: Returns the ID number of the new/current edit buffer.

Notes: **Buf_Switch(*r*)** only switches to a different edit buffer; it does not itself switch to a different window although Visual Mode will use a different window for the new edit buffer. Use the **Win_Switch( )** command to switch to a different window; you may also need to use the **Win_Attach( )** command to attach a window to an edit buffer.

See Also: Commands: Mem_Free( ), Win_Switch( ), Win_Attach( ), Buf_Num, Buf_Window
"Multiple File Editing" in Chapter 2, "Event Macros" in Chapter 3

Examples:

Buf_Switch(4)	Switch to edit buffer 4.
Buf_Switch(1,LOCAL)	Switch to edit buffer1 without performing any file buffering.
Buf_Switch(#98)	Switch to the edit buffer specified by the value in numeric register 98.

BSTAT Buf_Status - See Buf_Free
BT Buf_Total - See Buf_Free
BW Buf_Window - See Buf_Free

Cab_Extract("*file.cab*")

Usage: Cab_Extract("a:\vp-fils.cab")

Description: **Cab_Extract("*file.cab*")** extract all compressed files in the .CAB '*file.cab*' into the current directory. .CAB files are similar to ".ZIP" files. The installation disks of most Windows programs use .CAB files as a way of compressing and organizing the files to be installed. This command is used by the **`installw.vdm`** macro to install the VEDIT files.

Notes: Although '*file.cab*' can be a full pathname, the files are expanded into the current directory. If necessary, precede this command with **Chdir( )** to change to the desired directory.

Examples:

Chdir("c:\vedit") **Cab_Extract("a:\vp-fils.cab")**	Extract the files in the floppy disk file **`vp-fils.cab`** into the "c:\vedit" directory.

Call(*r,"label"*)

Usage: Call(4) Call(20+BUFFER) Call("SEARCH")

Description: **Call(***r***)** executes the contents of text register '*r*' as a command macro, starting at the beginning of the register. A macro may invoke another macro, which in turn may invoke another, up to a nesting depth of 25. **Call(***r* **+BUFFER)** allows edit buffer '*r*' to be executed as a macro.

Call(*r***,"***label***")** begins macro execution at the '*label*' instead of at the beginning of register '*r*'.

Call("*label***")** executes the "subroutine" macro '*label*' in the current text register or edit buffer.

The error "CANNOT MODIFY EXECUTING MACRO" results if a macro attempts to change a text register that contains an executing command macro.

Return: Always returns a value of 1. Use **Return_Value** to get the return value of any macro that returned with a **Return(***n***)** statement.

See Also: Commands: Reg_Set( ), Chain( ), Reg_Load( ), Reg_Save( ), Call_File( ), Macro_Num
Topics: "Command Macros" in Chapter 3
{MISC, Execute Macro} function

CALLF Call_File(*r*,"*file*")

Usage: Call_File(10,"print.vdm") CALLF(1,"compare")

Description: **Call_File(***r***,"***file***")** loads the file '*file*' into text register '*r*' and executes it immediately as a command macro. **Call_File(***r***,"***file***")** is equivalent to the commands **Reg_Load(***r***,"***file***",EXTRA)** and **Call(***r***)**.

If the file is not found in the current directory, VEDIT will look for it first in the *User Macro Directory*, then in the *VEDIT Macro Directory*, and last in the *VEDIT Home Directory*.

The default extension ".VDM" may be omitted. Thus, **Call_File(10, "compare")** executes the **`compare.vdm`** macro. To load a file that does not have an extension, you ***MUST*** include the "." (period). For example, the command to load/execute the file **`filer`** in register 4 is **Call_File(4,"filer.")**

Return: Always returns a value of 1. Use **Return_Value** to get the return value of any macro that returned with a **Return(***n***)** statement.

Notes: **Call_File(***r***,"***file***")** is similar to **{MISC, Load/Execute Macro}**.

See Also: Commands: Call( ), Chain_File( ), Reg_Load( )
Topics: "Command Macros" in Chapter 3

Examples: **Call_File(10,"print")** Load the print formatter macro **`print.vdm`** into text register 10 and execute it.

CLB Case_Lower_Block(*p,q*)
CSB Case_Switch_Block(*p,q*)
CUB Case_Upper_Block(*p,q*)

Options: COLUMN, COLSET, LINESET, NORESTORE, RESET

Usage: Case_Lower_Block(BB,BE)
Case_Switch_Block(BB,BE,COLSET,CB,CE)

Description: **Case_Lower_Block(***p,q***)** converts all letters to lower case in the block of text between file positions '*p*' and '*q*'. Non-letter characters are not changed. The edit position is not changed.

Case_Switch_Block(*p,q***)** switches the case of all letter.

Case_Upper_Block(*p,q***)** converts all letters to upper case.

Case_Lower_Block(*p,q***,NORESTORE)** moves the edit position past the end of the converted block.

Case_Lower_Block(*p,q***,RESET)** clears the **Block_Begin** and **Block_End** markers (if they are set).

The command options "**COLUMN**", "**COLSET**" and "**LINESET**" specify a columnar or line-range block for conversion.

See Also: Commands: Block_Copy( ), Reg_Copy( )
Topics: "Block Operations" in Chapter 3

Examples: **Case_Lower_Block(0,File_Size)**

Convert all letters in the current file (buffer) to lower case.

Chain(*r*)

Usage: Chain(10)

Description: **Chain(***r***)** jumps to execute text register '*r*' as a command macro. It differs from the **Call( )** command in that command execution does not return to the macro with the **Chain( )** command. Therefore, there should be no commands following the **Chain( )** command because they will never be executed. The text register with the **Chain( )** command can later be overwritten or emptied.

See Also: Commands: Call( ), Reg_Empty( ), Chain_File( )
Topics: "Command Macros" in Chapter 2

Examples: **Chain(10)** "Jump" to execute text register 10 as a command macro.

CHAINF Chain_File(*r*,"*file*")

Usage: Chain_File(20,"trans2.vdm")

Description: **Chain_File(***r***,"***file***")** loads '*file*' into text register '*r*' and then jumps (chains) to '*r*' and executes it as a command macro. It does not return

to the register issuing the **Chain_File(***r***,"***file***")** command. No commands should follow **Chain_File(***r***,"***file***")** because they will not be executed.

If the file is not found in the current directory, VEDIT will look for it first in the *User Macro Directory*, then in the *VEDIT Macro Directory*, and last in the *VEDIT Home Directory*.

Chain_File(*r***,"***file***")** is a special combination of the **Reg_Load( )** and **Chain( )** commands because '*r*' can be the currently executing text register.

Notes: **Chain_File(***r***,"***file***")** is a convenient way for a macro to chain to other macros without needing additional text registers.

Examples:

Chain_File(100,"trans2.vdm")	Load the file TRANS2.VDM into register 100 and execute it as a command macro.

C Char(*m*)

Usage: Char(12) Char(-4) C(12000)

Description: **Char(***m***)** moves the edit position by '*m*' characters. Movement is forward if '*m*' is positive or backward if '*m*' is negative.

Notes: No error is given if **Char(***m***)** attempts to go beyond the limits of the edit buffer or file.

Remember that every line normally ends in a "newline" which takes up one (UNIX/Macintosh) or two (Windows/DOS) character positions. The internal value **Newline_Chars** returns the number of characters in the "newline". With Windows/DOS text files, the "newline" is two characters (Carriage-Return and Line-Feed) and **Newline_Chars** has a value of "2".

See Also: Commands: Chars_Matched(), Del_Char( ), Line( ), Goto_Pos( )

Examples:

if (At_EOL) { ** Char(Newline_Chars)** **} else { Char( ) }**	Simulate **[CURSOR RIGHT]** by treating the "newline" as a single character, even if it is two.

CD Char_Dump(*n*)
TC Type_Char(*n*)

Options: COUNT, NOCR

Usage: Char_Dump(132) CD(#10,NOCR) Char_Dump(Buf_Num) Type_Char(205,COUNT,25)

Description: **Char_Dump(***n***)** dumps (displays) the character with ASCII value '*n*' followed by a "newline". Control and graphics characters are not expanded; they are displayed literally.

Char_Dump(*n***,NOCR)** suppresses the "newline".

Type_Char(*n*) displays the character with ASCII value '*n*'. Control and graphics characters are displayed according to the window's current display mode.

Type_Char(*n*,COUNT,*x*) displays the character '*x*' times.

Notes: '*n*' is often a numeric variable such as "#*x*".

Type_Char( ) is similar to **Char_Dump(*n*,NOCR)** and is usually the preferred command. It is equivalent when the window's display mode is "0".

See Also: Commands: Reg_Type( ), Out_OS( ), Message( )
Topics: "Screen Display & Keyboard Characters" in Chapter 4 of the VEDIT User's Manual

Examples:

Char_Dump(#1) Displays the character whose value is stored in numeric register 1. It is followed by a "newline".

Type_Char(205,COUNT,25)
Uses the IBM PC box drawing character to display a double-line 25 columns wide.

CMAT Chars_Matched

Usage: Internal Value

Description: **Chars_Matched** returns the number of characters matched by successful **Search( )**, **Replace( )**, and **Match( )** commands. It also returns the number of characters matched by **Compare( )** and **Reg_Compare( )** and the number of characters scanned by **Num_Eval( )**.

Examples:

Char(Chars_Matched) Advance the edit position by the number in Chars_Matched.

Chdir("*path*")

Options: NOMSG, NOCR

Usage: Chdir("d:\vedit") Chdir

Description: **Chdir("*d:\path*")** changes the "current" (logged in) drive and/or directory to the specified one. This allows files to be accessed without having to specify the drive and/or pathname each time.

Chdir without any arguments displays the current drive and directory, same as **Name_Dir( )**. **Chdir(NOMSG)** suppresses the "Current directory:" heading. **Chdir(NOCR)** suppresses the following "newline".

See Also: Commands: File_Open(*file*,CHGDIR), Name_Dir( )
The "[ ] Change directory" option in the File-open dialog box.

Examples:

Chdir("c:\windows") Changes to the Windows directory.

CCB Clip_Copy_Block(*p,q*) (Windows Only) Clip_Ins( )

Options: COLSET, COLUMN, DELETE, FILL, INSERT, LINESET, NORESTORE, RESET

Usage: Clip_Copy_Block(BB,BE) Clip_Ins(COLUMN)

Description: **Clip_Copy_Block(*p*,*q*)** copies the block of text starting with the '*p*'th character in the file up to, but not including, the '*q*'th character to the Windows clipboard.

Clip_Copy_Block( ) has the same **COLUMN**, **COLSET**, **DELETE**, **FILL**, **INSERT**, **LINESET**, **NORESTORE** and **RESET** options as the **Reg_Copy_Block( )** command.

Clip_Ins( ) inserts the (text) contents of the Windows clipboard into the edit buffer and advances the edit position.

Clip_Ins( ) has the same **BEGIN**, **COLUMN**, **LINEBLOCK** and **OVERWRITE** options as the **Reg_Ins( )** command.

Notes: The maximum block size is about 45% of the physical memory size. The DOS version can use **Reg_Copy_Block( )** and **Reg_Ins( )** commands with the **CLIPBOARD** option; however the maximum block size is then only 64K.

Only text should be copied to the clipboard. Binary data should not be copied; the copy will be truncated at the first Null character.

See Also: Commands: Reg_Copy_Block( ), Reg_Ins( )

Color_Highlight Color_Prompt

Usage: Internal Values

Description: **Color_Highlight** returns a color suitable for highlighting selection letters and keynames in a help window created with the macro language.

(Windows) It returns **Config(CW_HIGHLIGHT)**, which by default is red on white (value 124).

(DOS) It returns the value of **Config(C_HIGHLIGHT)**, which by default is bright-white on grey (value 127).

Color_Prompt returns a color suitable for displaying prompts in windows created with the macro language; the default is bright-white on green (value 47). It return the value of **Config(CW_PROMPT)** (Windows version) or **Config(C_PROMPT)**.

See Also: The **`display.vdm`** and **`keyedit.vdm`** macros as examples.

Examples: **Win_Color(Color_Highlight)** Change the window's display color.

CB Column_Begin(*n*) - See Block_Begin( )
CE Column_End(*n*) - See Block_End( )

CM Column_Mode

Options: CLEAR, SET

Usage: Column_Mode Column_Mode(SET) Column_Mode(CLEAR)

Description: **Column_Mode** returns 1 (TRUE) if Visual Mode is currently set to highlight a columnar block. Otherwise, it returns 0.

Column_Mode(SET) sets "column" mode for blocks in Visual Mode. **Column_Mode(CLEAR)** sets "stream" mode for blocks in Visual Mode.

Notes: **Block_Mode** is more versatile than **Column_Mode** and is the preferred command.

See Also: Commands: Block_Mode()

Examples:
```
if (Column_Mode) {
   Reg_Copy_Block(5,BB,BE,COLSET,CB,CE)
} else {  Reg_Copy_Block(5,BB,BE)  }
```
If the currently highlighted block is a columnar block, copy it to register 5 as a columnar block. Otherwise, copy it as a stream block.

Compare(*r*)

Options: CASE

Usage: Compare(8) Compare(20,CASE) Compare(2+BUFFER)

Description: **Compare(***r***)** compares (matches) the text at the edit position to the contents of text register (or edit buffer) '*r*'. It performs a character-by-character comparison without pattern matching. If '*r*' is a text register, the comparison is with the entire register and the edit position is advanced past all matching characters. If '*r*' is an edit buffer, the comparison starts at the edit position in buffer '*r*' and both edit positions are moved past all matching characters.

Compare(*r***,CASE)** forces the compare to be case sensitive, otherwise it is case insensitive.

Return: Results of the comparison are returned and saved in **Return_Value**:

0 The comparison is successful, the register/buffer matches.
1 The text is lexically "greater than" '*r*' or '*r*' is empty.
2 The text is lexically "less than" '*r*'.

Chars_Matched is set to the number of matching characters; is set to 0 (zero) if the very first character does not match. If '*r*' is empty, the command returns 1.

Notes: Two differences between **Match( )** and **Compare( )** are that **Match( )** uses pattern matching and does not move the edit position unless the entire match is successful. **Compare(***r***)** matches as much as possible, advancing the edit position for a partial match. **Compare( )** is useful for moving the edit position in two edit buffers past all characters which match, regardless of whether the two edit buffers match completely.

See Also: Commands: Match( ), Return_Value, Chars_Matched
Topics: "Match and Compare" in Chapter 3

Examples: **Compare(10+BUFFER)** Matche text at the edit position with the text at the edit position in edit buffer 10 and move both edit positions past all characters that match.

CF Config(*name*,*n*)

Options: ALL, LOCAL

Usage: Config(F_AUTO_SAVE,20) Config(w)

Description: **Config(***name***,***n***)** accesses the configuration parameters described in Chapter 8 of the VEDIT User's Manual. Refer there for a description of the configuration parameter names accessed by these commands.

Config(*name***,***n***)** changes the value of '*name*' to '*n*'. **Config(***name***)** with no arguments returns the current parameter value.

For edit buffer-dependent parameters, **Config(***name***,***n***)** changes the settings for the current edit buffer and all subsequently created edit buffers. **Config(***name***,***n***,LOCAL)** changes the settings for the current edit buffer only. **Config(***name***, *n*,ALL)** changes the settings for all open and all subsequently created edit buffers.

Return: **Config(***name***,***n***)** returns the value of the '*name*' parameter ***before*** it was changed to '*n*'. **Config(***name***)** returns the current setting of '*name*'.

Notes: You must use **{CONFIG, Save config}** or the **Config_Save( )** command to make any configuration changes permanent.

DOS: Use **{CONFIG, Misc, Save into VEDIT}** or the **Config_VEDIT( )** command to save the configuration changes into the VEDIT.EXE file. Some of the hardware configuration parameters (beginning with "H_") will not take effect until the next time VEDIT is started up.

See Also: Commands: Config_Display( )
Topics: "Configuration" in Chapter 8 of the VEDIT User's Manual

Examples: **Config(S)** Display the screen configuration parameters.

Config(C_MENU,31) Change the color of the pull-down menu to bright white on blue.

CFD Config_Display

Options: EXTRA, GLOBAL, LOCAL, SHORT, SUPPRESS

Usage: Config_Display()

Description: **Config_Display( )** displays all configuration parameters including strings and the tab stops. For edit-buffer dependent values, it displays the buffer's "local" value.

Config_Display(EXTRA) displays all parameters as complete "Config(...)" commands with the descriptive text.

Config_Display(EXTRA+SHORT) displays short "Config(...)" commands without the descriptive text.

Config_Display(EXTRA+SUPPRESS) suppresses some parameters (mostly DOS version hardware parameters) that should not be save in the **`vedit.cfg`** file. It is used internally by the **Config_Save( )** command.

Config_Display(GLOBAL) displays the current value of all configuration parameters. For edit-buffer dependent values, it displays the "global" value. Each newly opened buffer will initially have the "global" values.

Config_Display(LOCAL) only displays the edit-buffer dependent "local" values.

See Also: Commands: Config( )
Topics: "Configuration" in Chapter 8 of the VEDIT User's Manual

CFL Config_Load("*file*")
CFSAV Config_Save("*file*")

Options: NOERR, OK

Usage: Config_Load("myvedit.cfg",NOERR)
Config_Save("tempved.cfg")

Description: **Config_Load("***file***")** loads new configuration parameters from the file '*file*'. If '*file*' is not found in the current directory, the *VEDIT Home directory* will be searched.

Config_Load("*file***",NOERR)** suppresses the error message if '*file*' is not found; in this case **Error_Flag** is set.

Config_Save("*file***")** saves the entire configuration to the file '*file*'. If '*file*' exists, a prompt is displayed to confirm the overwrite. The '*file*' is a command macro containing all of the **Config(***name***)** parameters. The '*file*' is saved as a text file which can be easily edited.

Config_Save("*file***",OK)** skips the confirmation prompt when '*file*' already exists.

Notes: See the notes for the **Config( ) command**

See Also: Topics: "Configuration" in Chapter 8 of the VEDIT User's Manual

Examples: **Config_Save("myvedit.cfg",OK)**

Save the current configuration settings to the text file **`myvedit.cfg`**. If the file exists, it is overwritten.

Config_Load("myvedit.cfg")

Load the configuration file MYVEDIT.CFG.

CFS Config_String(*name*,"*text*")

Usage: Config_String(HOME,"c:\editors\vedit") CFS

Description: **Config_String(***name***,"***text***")** changes the configuration string '*name*' to "*text*". **CFS**, without arguments displays the current value of all configuration strings. Chapter 8 of the VEDIT User's Manual and the on-line help describe all configuration strings.

Config_String(SYN_NAME,"*file.syn***")** configures the current buffer to the color syntax highlighting file '*file.syn*'. The .SYN file will be loaded the next time the user enters Visual Mode.

Config_String(VTM_NAME,"*file.vtm***")** configures the current buffer to the template editing macro file '*file.vtm*'. The .VTM file will be loaded the next time the user enters Visual Mode.

Notes: **SYN_NAME** and **VTM_NAME** are typically set by the "File-open Configuration" feature.

See Also: Commands: Config_Display( )
Topics: Appendix D "String Arguments" of this manual
"File-open Configuration" in Chapter 5 of the VEDIT User's Manual

Examples: **Config_String(HOME,"c:\editor")**

Change the configuration string HOME (the VEDIT Home Directory) to "c:\editor".

CFT Config_Tab(*n*)

Options: ALL, LOCAL

Usage: Config_Tab(20,40,60,80,100,120) CFT(8;LOCAL)

Description: **Config_Tab(***n***)** changes the tab stops used for displaying Tab characters and, when **{CONFIG, Emulation, Expand <Tab> with spaces}** is set, for expanding the **<Tab>** key. Up to 33 explicit tab stops in the range 1 - 254 may be set. If only one number '*n*' is given, uniform tab stops at every n'th column are set.

Config_Tab with no arguments displays the tab stops.

Each edit buffer has its own tab stops. **Config_Tab(***n***)** changes the tab stops for the current edit buffer and all subsequently opened edit buffers. **Config_Tab(***n***;LOCAL)** changes the tab stops only for the

current edit buffer. **Config_Tab(*n*;ALL)** changes the tab stops for all open and all subsequently opened edit buffers.

Notes: 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 ...

In order to use command options, they ***MUST*** be preceded with a ";" (semicolon).

The tab stops can also be changed with **{CONFIG, Tab stops}.**

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

See Also: Commands: Next_Tab_Stop
Topic: Chapter 8 "Configuration" in the VEDIT User's Manual

Examples: **Config_Tab(8)** Set standard tab positions for the current and subsequently opened edit buffers.

CFV Config_Vedit("*file*") (DOS only)

Options: KEYBOARD

Usage: Config_Vedit("vedit.exe")

Description: **Config_Vedit("*file*")** saves the current configuration permanently into '*file*' (usually VEDIT.EXE).

Config_Vedit("*file*", KEYBOARD) also saves the entire current keyboard layout into '*file*'.

See Also: Topics: "Configuration" in Chapter 8 of the VEDIT User's Manual

Continue

Usage: if (Error_Match) { Continue }

Description: **Continue** stops the current iteration of the current **While**, **Do-while**, **For**, or **Repeat** loop, causing the loop to be re-tested. If there is no loop, the command is ignored.

See Also: Commands: Goto_Pos( ), Break, Return( ), Goto label
Topics: "Flow Control - Break-out Commands" in Chapter 3

Examples: See the topic "Flow Control" in Chapter 3 for examples.

CC Cur_Char Cur_Char(m)
CN Cur_Col
CL Cur_Line
CP Cur_Pos
CRN Cursor_Col

Usage: Internal Values

Description: **Cur_Char** returns the value of the character at the edit position. At the End-Of-File it returns 26 (<Ctrl-Z>). For example, with the cursor on an "A", it returns 65.

Cur_Char(*m***)** returns the value of the '*m'th*' next/previous character. **Cur_Char(0)** is the same as **Cur_Char**.

Cur_Col returns the horizontal column number for the character at the edit position. It accounts for the "width" of each character in the current display mode.

Cur_Line returns the line number in the file of the current line of the edit position. This is the "LINE:" number displayed on the status line.

Cur_Pos returns the edit position (offset) in the file. The position of the first character in the file is 0 (zero). This is probably the most used internal value.

Cursor_Col returns the column number corresponding to the cursor position in Visual Mode. This is the "COL:" number displayed on the status line. It is greater than **Cur_Col** when the cursor is positioned past the end-of-line. It can be used to specify the columns for an upcoming columnar block command.

Examples:

Block_Begin(Cur_Pos)	Set the block_begin marker at the current edit position.
Column_End(Cursor_Col)	Set the column_end marker at the same column as the Visual Mode cursor.
Cur_Char(-1)	Return the value of the previous character.
Cur_Char(BOL_Pos-Cur_Pos)	Return the value of the character at the beginning of the current line.
Cur_Char(File_Size-Cur_Pos-1)	Return the value of the last character in the file.

Date( )
Time( )

Options: BEGIN, EXTRA, NOCR, NOMSG, VALUE

Usage: Date( ) Time(EXTRA) Date(NOCR)

Description: **Date( )** displays the system date as mm-dd-yyyy preceded by the header "Date:" and followed by a "newline".

Date(BEGIN) displays the date as dd-mm-yyyy.

Date(BEGIN+VALUE,'.') displays the date as dd.mm.yyyy.

Date(NOCR) suppresses the "newline", and **Date(NOMSG)** suppresses the header.

Time( ) displays the system time preceded by the header "Time:" and followed by a "newline". **Time(EXTRA)** displays the time with 1/18 second resolution.

Time(NOCR) suppresses the "newline" and **Time(NOMSG)** suppresses the header.

See Also: Commands: Time_Tick

Time(EXTRA+NOCR) Display the system time with 1/18 second resolution and suppress the "newline".

Out_Ins Date(NOCR) Out_Ins(CLEAR)

Insert the date into the text.

DB Del_Block(*p,q*)

Options: COLUMN, COLSET, LINESET, NORESTORE, RESET

Usage: Del_Block(Block_Begin,Block_End) Del_Block(1,80)

Description: **Del_Block(***p,q***)** deletes the block of text starting with the '*p*'th character in the file up to, but not including the '*q*'th character. If the edit position is within the deleted block, it will be moved to the character following the deleted text; otherwise, the edit position is not moved.

Del_Block(*p,q***,NORESTORE)** always moves the edit position to the character following the deleted text.

Del_Block(*p,q***,RESET)** clears the **Block_Begin** and **Block_End** markers if they are set.

The command options "**COLUMN**", "**COLSET**" and "**LINESET**" specify a columnar or line-range block for deletion.

Notes: Use the **Del_Line( )** command to delete entire lines of text.

See Also: Commands: Char( ), Del_Line( ), Del_Char( ), Block_Copy( )
Topics: "Block Operations" in Chapter 3

Examples: **Del_Block(Block_Begin,Block_End,COLUMN)**

Delete the columns of text between the Block_Begin marker and the Block_End marker.

Del_Block(BB,BE,COLSET,20,30)

Delete the text between columns 20 and 30 on the lines from the Block_Begin marker (BB) to the Block_End marker (BE).

DC Del_Char(*m*)

Usage: Del_Char(12) Del_Char(-4)

Description: **Del_Char(*m*)** deletes '*m*' characters from the text. If '*m*' is positive, the '*m*' characters immediately at and following the edit position are deleted. If '*m*' is negative, the '*m*' characters preceding the edit position are deleted.

Fewer than '*m*' characters are deleted (without error) if the limits of the edit buffer/file are reached.

Notes: Use the **Del_Line( )** command to delete entire lines of text.

See Also: Commands: Char( ), Del_Line( ), Del_Block( )
Topics: "Block Operations" in Chapter 3

DL Del_Line(m)

Usage: Del_Line(4) DL(-3) Del_Line(0)

Description: **Del_Line(m)** deletes the next/previous '*m*' lines. **Del_Line(n)** deletes all text from the edit position up to and including the '*n*'th "newline". **Del_Line(-n)** deletes the previous '*n*' lines and all text preceding the edit position on the current line. **Del_Line(0)** deletes all text (if any) preceding the edit position on the current line.

Fewer than '*m*' lines are deleted (without error) if the limits of the edit buffer/file are reached.

See Also: Commands: Del_Char( ), Type( ), Reg_Copy( )
Topics: "Block Operations" in Chapter 3

Examples:

```
Repeat (ALL) {
   Search("temp line")
   Del_Line(0) Del_Line(1)
}
```

Delete all lines that contain the string "temp line".

```
Reg_Copy(6,100)
Del_Line(100)
```

Save the following 100 lines of text in text register "6" and then delete them from the edit buffer.

DTH Desktop_Height (Windows only)
DTW Desktop_Width

Usage: Internal Values

Description: **Desktop_Height** and **Desktop_Width** return the size of the screen in pixels, e.g. 600 x 800 or 768 x 1024.

See Also: Commands: App_Height, App_Width, Win_Move( )

DTAB Detab_Block(*p,q*)
RTAB Retab_Block(*p,q*)

Options: COLUMN, COLSET, LINESET, NORESTORE

Usage: Detab_Block(BB,BE) Retab_Block(BB,BE,COLUMN)

Description: **Detab_Block(***p,q***)** converts all tab characters in the block of text to spaces according to the current tab stops. The edit position is left as close to its original position as possible.

Detab_Block(*p,q***,NORESTORE)** sets the edit position past the end of the converted block.

Retab_Block(*p,q***)** converts spaces in the block of text to the optimum number of tabs and spaces according to the current tab stops. Only sequences of two or more spaces are converted; single spaces are never converted to tabs.

The command options "COLUMN", "COLSET" and "LINESET" specify a columnar or line-range block for conversion.

Notes: These commands are similar to **{BLOCK, Edit/translate, Detab}** and **{BLOCK, Edit/translate, Retab}**.

See Also: Commands: Block_Copy( ), Reg_Copy( ), Translate_Block( )
Topics: "Block Operations" in Chapter 3

Examples: **Detab_Block(0,File_Size)** Convert all tabs in the entire buffer (file) to spaces.

DI1 Dialog_Input_1("...") (Windows only)

Options: See below.

Description: **Dialog_Input_1(***r,***"***string***",***OPTIONS,x,y***)** creates a dialog box with a title, text, up to ten (push) buttons, and optional check boxes, radio buttons and string input boxes.

The dialog box is dynamically sized according to the specified text, the number of buttons, and the other specified items. The dialog box can be precisely positioned with respect to the screen or the VEDIT application window.

The command returns the number of the button selected (1 - 10) or 0 if **<Esc>** was pressed to cancel the dialog box.

The dialog box items are specified in the '*string*' one after another in their displayed order from top to bottom. Default spacing is added between each item; however, items can also be displayed side-by-side, and the vertical and horizontal spacing can be increased or decreased from the default spacing.

NOTE: Refer to the on-line help for a detailed description of the '*string*' items and their many options.

If check boxes or radio buttons are used, '*r*' is the numeric register corresponding to the first check box or radio button. The second check box or radio button corresponds to numeric register '*r*'+1, and so on. The numeric register(s) are used both for setting the initial value and returning the final value. Therefore, the numeric register(s) must be initialized before the **Dialog_Input_1( )** command.

If string input boxes are used, '*r*' is the text register corresponding to the first input box. The second input box corresponds to text register '*r*'+1, and so on. The text register(s) are used for returning the final string. They are optionally used for setting the default input string(s), in which case they must be initialized.

Dialog_Input_1(*r***,"***string***","***default-input-1***", *OPTIONS*, *x*,*y*)** is an optional way of specifying the default text for the first input string.

Command options:

SET	Set the default text for the string input boxes from the corresponding text registers. Otherwise, the default text for the first input box will be set from '*Default-input-1*', if specified.
COUNT,*n*	Limit the maximum number of characters in each string input box to '*n*' characters.
APPEND	Append the input box string(s) to the existing contents of the corresponding text register(s).
INSERT	Insert the input box string(s) at the beginning of the existing contents of the corresponding text register(s).

The default dialog box position is centered horizontally in the screen and 1/3 down from the top. However, the dialog box can be precisely positioned.

Specify one option for the frame of reference:

SCREEN	(Default) relative to the entire screen.
APP	Relative to the VEDIT application program.
WORKAREA	Relative to VEDIT, but not including the toolbar, status line and any reserved windows.

Also specify up to two non-conflicting orientations:

TOP **BOTTOM** **LEFT** **RIGHT** **CENTER**	**TOP** and **LEFT** refer to the top and left edges of the dialog box and the top and left of the reference frame. Use positive '*x*' and '*y*' values to move the box right and down. **BOTTOM** and **RIGHT** refer to the bottom and right edges of the dialog box and the reference frame. Use negative '*x*' and '*y*' values to move the box left and up. **CENTER** applies both horizontally and vertically, but is overridden vertically with **TOP** or **BOTTOM**, and horizontally with **LEFT** or **RIGHT**. It refers to the center of the dialog box and the center of the reference frame. Therefore "**CENTER,0,0**" exactly centers the dialog box.
x,y	Specifies a pixel offset from the top, bottom, left, right or center orientation. '*x*' is the horizontal offset; '*y*' is the vertical offset. Positive values move the dialog box right or down; negative values move it left or up. If **SCREEN**, **APP** or **WORKAREA** is specified, both the '*x*' and '*y*' offsets must be specified; use "**0,0**" if no offset is desired. If **SCREEN**, **APP** or **WORKAREA** is not specified, '*x*' and '*y*' must be omitted and the only allowed orientation option is **CENTER**.

Return: Returns the number of the button pushed (1 - 10) or 0 if **<Esc>** was pressed to cancel the dialog box.

See Also: Commands: Get_Input( ), Get_Key( ), Get_Num( )
The on-line help describes the dialog box items in detail.

Examples:

```
#1=Dialog_Input_1(5,"`This is the Title`,
`This is line one of the text`,
`This is line two of the text`,
`??Input:`,
`This is line three of the text`,
`[Button&1]`,`[Button&2]`,`[Button&3]`,`[Cancel]`",
"a:\",SCREEN+CENTER,0,0)
```

```
#70 = DI1(#95,^`Wildfile Wizard`,
`Enter list of filenames to process.`,
`??`,
`[]&Search subdirectories`,
`[&Ok]`,`[&More]`,`[Cance&l]`^,
SCREEN+CENTER,0,0)
```

DIR Directory("*fspec*")

Options: COUNT, NOERR, NOMSG, SHORT, SUPPRESS

Usage: Directory("b:*.txt") Dir("C:*.*",NOMSG+SUPPRESS)

Description: **Directory("***fspec***")** displays a list of files and subdirectories on any desired drive/directory. Optional wildcard characters "**?**" and "*****" may be specified to list only those files that match '*fspec*'.

Directory("*fspec* **-s")** includes files in all subdirectories. A header line is displayed for each subdirectory. Subdirectories are not included in the list of matching files.

Directory("*fspec***",COUNT,***n***)** displays the directory in '*n*' columns instead of the normal 4 columns.

Directory("*fspec***",NOERR)** suppresses the error message if '*fspec*' doesn't match any files; it sets **Error_Flag** to "2". Also suppresses the error message if pathname contains a non-existent directory; it sets **Error_Flag** to "3".

Directory("*fspec***",NOMSG)** omits the header line consisting of the current drive and directory; it also displays the filenames one per line.

Directory("*fspec***",SUPPRESS)** omits displaying subdirectory names and hidden/system files.

Directory("*fspec***",SHORT)** displays long filenames in the short 8.3 format. The filename and extension are lined up into columns.

Return: Returns the number of displayed filenames. The return value is also saved in **Return_Value**.

See Also: Commands: Chdir( ), Name_Dir( )

Examples:

Directory("b:*.txt") Display the directory of all files with extension ".txt" on drive B.

Dir("b:\letters\") Display the directory of all files in directory "letters" of drive B. Note the final "\".

Out_Ins( ) Directory("*.*",NOMSG) Out_Ins(CLEAR)

Insert the directory into the edit buffer, one file per line.

DKF Disk_Free(*drive*)
DKS Disk_Size(*drive*)

Usage: Internal Values

Description: **Disk_Free(***drive***)** returns the amount of free space on the specified logical disk drive in Megabytes. One Megabyte = 1048576 bytes.

Return_Value is set to the exact remaining free bytes. Therefore, Disk_Free * 1048576 + Return_Value is the exact number of bytes

free; however VEDIT's 32-bit math cannot calculate this number correctly past 2 Gigabytes.

Disk_Size(*drive***)** returns the total size of the specified logical disk drive in Megabytes.

Examples: **Disk_Free** Display the available disk space on the current drive.

Disk_Size('c:') Display the total size of logical drive "C:" in megabytes.

DOV Do_Visual("*vm-codes*")

Usage: Do_Visual("\CD\\CD\\CD\") DOV("\ME\EC")

Description: **Do_Visual("***vm-codes***")** executes Visual Mode functions from within a command macro. The "Edit Function Codes" in Appendix A lists the two letter codes for each edit function. They must be surrounded by "\ \" and must appear within the explicit delimiters. Normal displayable characters can also be included — they are ***NOT*** surrounded by "\ \".

Notes: **Do_Visual( )** is a very powerful and flexible command. It can be used within keystroke macros to fine-tune existing editing functions. It permits decision making — selecting which Visual Mode function to perform based on a condition, such as the type of character at the cursor, or the number of block markers set.

Examples: **Do_Visual("\CD\\CD\\CD\")**

Perform three **[CURSOR DOWN]** from within a command macro.

DOV("\ME\EL") Center a line by selecting this function from the menu system.

EOF End_Of_File( )
EOL End_Of_Line( )

Options: LOCAL

Usage: End_Of_File( ) EOF(LOCAL) End_Of_Line( )

Description: **End_Of_File( )** moves the edit position past the last character in the file. **End_Of_File(LOCAL)** only moves the edit position to the end of the portion of the file currently in memory.

End_Of_Line( ) moves the edit position to the end of the current line, typically to the "newline". If the edit position is already at the end-of-line or at the end-of-file, the command is ignored.

See Also: Commands: Begin_Of_File( ), At_EOF, At_BOF
Topics: "File Buffering in Command Mode" in Chapter 3

Examples: **End_Of_File( ) Line(-100)** Move the edit position to the 100th line before the end of the file.

EOL_Pos - See BOL_Pos

EF Error_Flag
EM Error_Match
Error_OS

Usage: Internal Values

Description: **Error_Flag** returns the current value of the error flag. The flag is reset before each command is executed. It is set (value 1) by **Search( )**, **Replace( )**, **Match( )** and **Match_Parentheses( )** by an unsuccessful search/match. It is also set by other selected commands.

Error_Match returns the value of the search/match error flag. It is set/reset only by the **Search( )**, **Replace( )**, **Compare( )**, **Match( )**, **Match_Parentheses( )**, **and Reg_Compare( )** commands. It allows testing the results of these commands many commands later.

Error_OS returns the write error flag set/reset by the last disk write operation. It returns 1 if there was a write error; 0 if no write error.

Notes: Since **Error_Flag** is reset by each command, it must be tested immediately after the command.

Examples: **if (Error_Match > 0) { Goto ERROR }**

If the last search/match/compare was unsuccessful, jump to the label "ERROR".

Escape_Mode(*vm-code*)

Usage: Escape_Mode('M'+'E'*256)

Description: **Escape_Mode(***vm-code***)** programs the **[ESCAPE]** function to perform an alternative to popping up the **{ESCAPE}** menu. Any of the edit functions listed in Appendix A of the VEDIT Macro Language Reference Manual are acceptable.

Examples: **Escape_Mode('V'+'E'*256)** Force the [ESCAPE] function to perform [VISUAL EXIT].

Exit
XALL
QALL / QALLY

Usage: XALL Exit( ) QALL

Description: **Exit** exits VEDIT, similar to **{FILE, Exit}**, prompting the user to save or abandon each altered file/buffer. If an edit buffer has been altered, but has no assigned filename, it prompts for one.

XALL exits VEDIT, saving each altered file/buffer, but without prompting for confirmation. This is a fast way to exit VEDIT, saving all files. It only saves buffers with open files.

QALL exits VEDIT, quitting (abandoning) all files. The user is prompted for confirmation. **QALLY** exits VEDIT, quitting all files, but without the confirmation prompt. (This command should be used with extreme care!)

The command forms **Exit(*n*)**, **Xall(*n*)**, **Qall(*n*)** and **Qally(*n*)** return '*n*' to the operating system as VEDIT's "return code" instead of the default value of zero. A DOS Batch file can test the return code with "ERRORLEVEL".

See Also: Commands: File_Close( ), File_Quit( ), File_Save( )

Examples:

```
vpw myfile.txt
[VISUAL EXIT]
Exit
```

The editor is invoked in the normal way to edit a file in Visual Mode. The new file is then saved on disk before exiting VEDIT.

XBUF1 Extra_Buffer_1
XBUF2 Extra_Buffer_2
XBUF3 Extra_Buffer_3
XBUF4 Extra_Buffer_4

Usage: Internal Values

Description: Besides the normal edit buffers typically used to edit files, VEDIT also has "extra" buffers which are often used by macros for processing blocks of text and other purposes that require a temporary edit buffer.

The Windows version has 26 extra buffers with an ID number of 100 - 125. The DOS version has 4 extra buffers with an ID number of 33 - 36. Since the ID numbers are different for the Windows and DOS versions, these internal values should always be used the extra buffers are referenced, typically with the **Buf_Switch( )** command. These internal values are also more mnemonic.

Notes: Many macros use **Buf_Free(EXTRA)** to dynamically determine the next free "extra" buffer.

Any text placed into an "extra" buffer, or any file opened in an "extra" buffer is not automatically saved when VEDIT exits.

See Also: Commands: Buf_Free(EXTRA), Buf_Switch( )

FA File_Attrib("*file*",*n*)

Options: RESET, SET

Usage: File_Attrib("archive.001",1,RESET)

Description: **File_Attrib("*file*")** returns the attributes (read-only, hidden, system, etc.) of '*file*'. Windows/DOS define the attributes as:

Mask 1	Read-only
Mask 2	Hidden
Mask 4	System
Mask 8	Volume label
Mask 16	Directory
Mask 32	Archive

File_Attrib("*file*",*n*) sets the attributes of '*file*' to '*n*'.

File_Attrib("*file*",*n*,SET) sets the attribute(s) (bits) '*n*'.

File_Attrib("*file*",*n*,RESET) clears the attribute(s) (bits) '*n*'.

Return: Returns the attributes of the specified file, or -1 if the file does not exist. If changed, it returns the original attributes.

Examples: The following example removes the read-only attribute before editing the file. After editing, it restores the read-only attribute.

```
File_Attrib("archive.sav",1,RESET)
File_Open("archive.sav")
File_Close()
File_Attrib("archive.sav",1,SET)
```

FCHK File_Check("*file*")

Usage: File_Check("myfile.txt")

Description: **File_Check("*file*")** checks if the specified file is currently open in one of VEDIT's edit buffers and returns the buffer's ID number.

Return: If the file is not currently open, it returns -1. If the file is open, it returns the ID number of the edit buffer in which the file is open. It also returns the result of its check in **Return_Value**.

Notes: The macro **`wildfile.vdm`** uses this command to check if the next file to be opened is already open. If it is open, it switches to the corresponding edit buffer; if it is not open, it opens it in a new buffer.

See Also: Commands: File_Exist(EXTRA), File_Open( )

Examples:

```
#10=File_Check("myfile.txt")
if (#10 > 0) {
   Buf_Switch(#10)
} else {
   Buf_Switch(1)
}
```

Check if the file "myfile.txt" is already open. If it is, it switch to the corresponding edit buffer. If not, switch to the main buffer #1.

FC File_Close( )

Options: CONFIRM, EVENT, OK, NOEVENT, NOMSG

Usage: File_Close( ) FC(NOMSG)

Description: **File_Close( )** saves and closes the file being edited in the current edit buffer, without prompting, and remains in the same buffer in preparation for editing another file. It is often followed by **File_Open( )** to edit another file. If the edit buffer has been altered, but has no assigned filename, it prompts for one.

File_Close( ) normally gives the message "Saving: *filename*"; this message can be suppressed with **File_Close(NOMSG)**.

File_Close(CONFIRM) prompts for confirmation to save or abandon the file if it has been altered. If the user selects [No-abandon], it performs a **Buf_Empty( )** instead.

If **Config(F_E_F_MACRO,1)** is enabled, the "File-close" and "File-post-close" event macros will also run. **File_Close(NOEVENT)** suppresses the File-close event macros even if enabled. **File_Close(EVENT)** executes the File-close event macros even if disabled.

Notes: **File_Close( )** is ***not*** the same as **{FILE, Close}**. **File_Close( )** saves the current file (without confirmation) and remains in the current edit buffer. **{FILE, Close}** gets confirmation to Save/Abandon the current file and also closes the current edit buffer and all attached windows; it is equivalent to **Buf_Close(CONFIRM+DELETE)**.

{FILE, Open (More), Same Buffer} is a combination of the **File_Close(CONFIRM)** and **File_Open( )** commands.

While a "File Open" event macro is commonly used, "File Close" event macros are rarely used.

See Also: Commands: Buf_Close( ), File_Quit( ), File_Truncate( ), Is_Altered, Is_Open_Write
Topics: "Event Macros" in Chapter 3

Examples:

File_Close( ) **File_Open("newfile.txt")**	The current file is saved on disk, and the file "NEWFILE.TXT" is opened in the same buffer.

FCP File_Copy("*sfile*","*dfile*")
FMV File_Move("*sfile*","*dfile*")
FREN File_Rename("*sfile*","*dfile*")

Options: NOERR, OK

Usage: File_Copy("data.001","data.sav") FREN("letter.bak","letter.txt")

Description: **File_Copy("***sfile*","*dfile*") copies the file '*sfile*' to '*dfile*'. If '*dfile*' already exists, it prompts for confirmation to overwrite the file.

File_Copy(...,OK) skips the confirmation prompt.

File_Copy(...,NOERR) suppresses the error message if '*sfile*' does not exist or the copy cannot be made.

File_Move( ) and **File_Rename( )** are different names for the same function. They move/rename the file '*sfile*' to '*dfile*'. If '*dfile*' already exists, they prompts for confirmation to overwrite the file.

File_Move(...,OK) skips the confirmation prompt.

File_Move(...,NOERR) suppresses the error message if '*sfile*' does not exist or the rename/move cannot be made.

Notes: These commands overwrite existing files without creating a backup.

A move/rename on the same logical drive is nearly instantaneous because it does not need to be copied.

FDEL File_Delete("*fspec*")

Options: NOERR, OK, OK+EXTRA

Usage: File_Delete("file.txt") FDEL("*.bak")
File_Delete("c:\text*.txt",OK+EXTRA)

Description: **File_Delete("***fspec***")** deletes the file(s) '*fspec*' from disk. Wildcard characters "?" and "*" may be used to delete more than one file. The command first displays a directory of the files to be deleted and asks for confirmation.

File_Delete("*fspec***",OK)** skips the directory display and confirmation prompt. Use just "**OK**" if you don't expect '*fspec*' to contain the "*" wildcard character.

File_Delete("*fspec***",OK+EXTRA)** skips the confirmation prompt even if '*fspec*' contains the "*" wildcard character. USE WITH CARE -- a small oversight in your macro could delete many files!

File_Delete("*fspec***",NOERR)** suppresses the "**FILE NOT FOUND...**" error message if the specified file(s) is not found.

Return: Returns the number of files successfully deleted; 0 if none. Sets **Error_Flag** if the file(s) is not found.

Notes: Never delete any "*.r*$$" or "*.r*R$" files from within VEDIT! These are the temporary files VEDIT is using. Do not attempt to delete the files begin edited.

See Also: Commands: Directory( )

Examples:

File_Delete("c:\oldfile.txt") Delete the file "oldfile.txt" from the root directory of drive C:.

FDEL("*.bak") Delete all files with a filename extension of ".bak" from the current directory.

FEXIST File_Exist("*fspec*")

Options: NOERR, SUPPRESS

Usage: File_Exist("myfile.txt") FEXIST("*.*",SUPPRESS)

Description: **File_Exist("***fspec***")** tests for the existence of the file(s) '***fspec***' and returns a value of 1 if the file exists, 0 if it does not. If wildcard characters are used in '***fspec***', the return value is the number of files matching the specification. '***fspec***' can be the name of a directory or a system/hidden file.

File_Exist("*fspec***",SUPPRESS)** omits checking for directory names and system/hidden files.

File_Exist("*fspec***",NOERR)** suppresses the error message if the pathname contains a non-existent directory. In this case the return value is "0" and **Error_Flag** is set to "3".

Return: Returns the number of filenames matching '*fspec*'. (0 if none.)

See Also: Commands: Chdir( ), File_Check( ), Return_Value, Name_Dir

Examples: **.File_Exist("c:\windows")**

Display the number of files in the "c:\windows" directory.

if (File_Exist("startup.vdm")) { Exit }

Test for the existence of the file "startup.vdm". If found, it exits the editor.

FMD File_Mkdir("*dir*")
FRD File_Rmdir("*dir*")

Options: NOERR

Usage: File_Mkdir("c:\newdir") File_Rmdir("c:\oldir",NOERR)

Description: **File_Mkdir("*dir*")** makes (creates) the new directory '*dir*'. This is equivalent to the DOS/NT "md or mkdir" commands.

File_Mkdir("*dir*",NOERR) suppresses the error message if the directory cannot be created, e.g. if it already exists.

File_Rmdir("*dir*") removes (deletes) the empty directory '*dir*'. This is equivalent to the DOS/NT "rd or rmdir" commands or the UNIX "rm -R" command. Note that the directory must be empty, i.e. not have any files, before it can be removed.

File_Rmdir("*dir*",NOERR) suppresses the error message if the directory cannot be removed, e.g. if it contains files.

Return: **File_Mkdir( )** returns TRUE (1) if the directory was created, and FALSE (0) if the directory was not created.

File_Rmdir( ) returns TRUE (1) if the directory was removed, and FALSE (0) if the directory was not removed.

Notes: The **File_Delete("*.*")** or **File_Delete("*")** command can be used to delete all files in a directory.

See Also: Commands: **File_Delete( )**

Examples: Ensure that the directory "c:\data\april" exists, creating it if necessary:

```
if (File_Exist("c:\data")==0) {
   File_Mkdir("c:\data")
}
if (File_Exist("c:\data\april")==0) {
   File_Mkdir("c:\data\april")
}
```

FMV File_Move( ) - See File_Copy( )

FO File_Open("*fspec*")

Options: ATTACH, BROWSE, CHGDIR, EVENT, FORCE, MRU, NOEVENT, NOMSG, OVERWRITE

Usage: File_Open("file.txt") File_Open("'origfile.txt" -a "newfile.txt"') File_Open("a long filename") File_Open("'file1","file2"')

Description: **File_Open("***fspec***")** opens the specified file(s) for editing and reads as much of it into memory as possible. If the file does not exist, it is created. The file is opened in the current buffer if the current buffer has no open file. Otherwise, it is opened in the first available edit buffer. If the specified file is already open in another buffer, the command simply switches to that buffer.

Long Filenames

Long filenames with embedded spaces or commas must be enclosed in double-quotes; if multiple files are specified, or the "**-a**", "**-l**" and "**-t**" options are used, the entire string must be enclosed in other delimiters, typically single-quotes.

Two or more files can be specified; a group of files can be specified using the wildcard characters "*" and "?". Each additional file is opened in an additional edit buffer. Each file may include the "**-a**", "**-l**" and "**-t**" options.

The "**-a** *asfile*" option following a filename saves the file as '*asfile*'. The "**-l** *nnn*" option sets the initial edit position on line '*nnn*'. The "**-t** *nnn*" option sets the File type to '*nnn*', i.e. it sets **Config(F_F_TYPE,***nnn*).

File_Open("*file***",ATTACH)** creates a new overlapping window for each opened file. The windows are only created if **{CONFIG, Screen display, Auto-create windows for buffers}** (or equivalent **Config(WIN_AUTO_CRE)**) is enabled (default).

File_Open("*file*",BROWSE) opens the specified file in browse-only mode; it cannot be altered.

File_Open("*file*",CHGDIR) changes the "current" directory to the directory containing the file.

File_Open("*file*",FORCE) allows a file to be opened in an "extra" buffer. Note that files opened in "extra" buffers are not automatically saved, nor are you prompted to save them. Use with care. See the **`wildfile.vdm`** macro for an example.

File_Open("*file*",MRU) adds the filename to the "Most-Recently-Used" list in the **{FILE}** menu when the file is closed. If VEDIT is exited with "Quit all" or the **Qall** command, files are not added to the MRU list.(Windows version only)

File_Open("*file*",OVERWRITE) suppresses creating a backup file, even if backup files are enabled.

The message "New file" is displayed if the file does not already exist and is therefore created; the message can be suppressed with **File_Open("*file*",NOMSG)**.

If **Config(F_E_AUTO_CFG)** is enabled, the "File-open configuration" macro is run. If **Config(F_E_F_MACRO,1)** is enabled, the "File-pre-open" and "File-open" event macros will also run. The command option "**NOEVENT**" suppresses the File-open event macros, even if enabled. The command option "**EVENT**" executes the File-open event macros, even if disabled.

Notes:

File_Open( ) is similar to the **{FILE, Open}** function.

{CONFIG, File Handling, Enable backup files}, equivalent to **Config(F_E_BACKUP)**, determines whether a backup file is maintained.

A "File-open configuration" macro is typically set up by the **`startup.vdm`** macro to configure several parameters according to the file type (filename extension). A "File Pre-Open" event macro is rarely used.

For each file that **File_Open( )** opens, it performs the following steps to determine in which edit buffer it will open the file:

1. It checks if the file is already being edited. If so, it simply switches to the associated edit buffer.
2. If the current edit buffer is empty and has no open file, it opens the file in the current buffer.
3. It switches to the next available (free) edit buffer and opens the file in the new buffer.

After **File_Open( )** opens two or more files, it returns to the edit buffer of the first file opened.

See Also: Commands: File_Write( ), File_Open_Read( ), File_Open_Write( ), Is_Saveas
Topics: "File Editing Commands" and "Event Macros" in Chapter 3

Examples:

File_Close( )
File_Open("*.txt")
The current file being edited is closed and all files in the current directory with the ".txt" file extension are opened for editing. The first file is opened in the current edit buffer, and the additional files are opened in additional edit buffers.

File_Open('"part1.txt" -l 200')
The file "part1.txt" is opened and the edit position is set to line 200.

File_Open("myfile|@(1)")
The file named "myfile" with the file extension contained in text register 1 is opened.

FOPENR File_Open_Read("*file*")

Usage: File_Open_Read("newfile.txt") FOPENR("myfile.asm")

Description: (This is a technical, rarely used command!)

File_Open_Read("*file*") opens the file '*file*' for input (reading). However, nothing is actually read into the edit buffer. The **File_Read( )** 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.

Notes: As soon as VEDIT 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.

See Also: Commands: File_Read( ), File_Open( ), File_Write( ), Name_Read, Is_Open_Read
Topics: "Explicit Read/Write Commands" in Chapter 3

Examples:

File_Open_Read("parts.inv")
File_Read(20)
The file "parts.inv" is opened for input and twenty lines from it are read into the current edit buffer.

FOPENW File_Open_Write("*file*")

Usage: File_Open_Write("newdat.inv")

Description: (This is a technical, rarely used command!)

File_Open_Write("*file*") 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 **File_Open( )** command.

If a file is already open for output, the error "FILE IS ALREADY OPEN IN THIS BUFFER" is given.

Note: The file opened is actually a temporary file with the same filename, but with an extension of ".*r*$$" where '*r*' is the ID number of the current edit buffer. The file is not made permanent and given its true name until it is "closed" with the **File_Save( )**, **File_Truncate( )**, **Buf_Close( )** or **File_Close( )** commands. 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: File_Write( ), File_Save( ), File_Truncate( ), Buf_Close( ), File_Close( ), Is_Saveas, Is_Open_Write
Topics: "Explicit Read/Write Commands" in Chapter 3

FQ File_Quit( ) - See Buf_Empty( )

FR File_Read(*n*)

Options: REVERSE

Usage: File_Read(100) File_Read(0,REVERSE)

Description: (This is a technical, rarely used command!)

File_Read(*n*) appends the next '*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. **File_Read(0)** reads all of the input file or until the edit buffer is almost full. **File_Read(*n*)** can be executed (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 **File_Open( )** and **File_Open_Read( )** commands.

File_Read(*n*,REVERSE) reads back '*n*' lines from the output file to the beginning of the edit buffer. **File_Read(0, REVERSE)** 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. These commands perform the reading necessary for backward file buffering.

Notes: No indication is given if fewer than '*n*' lines were appended.

See Also: Commands: Mem_Status, File_Write( ), File_Open( ), File_Open_Read( )
Topics: "Explicit Read/Write Commands" in Chapter 3

Examples:

File_Open_Read("text.doc") **File_Read(0)**	Open the file "text.doc" and read it all in, or until the edit buffer is almost full.

File_Read(-0)	Read back as much of the output file as will fit into the beginning of the edit buffer.

FREN File_Rename( ) - See File_Copy( )

FRD File_Rmdir( ) - See File_Mkdir( )

FS File_Save( )
FSA File_Save_As("file")

Options: ALL, BEGIN, NOMSG, OK

Usage: File_Save( ) FS(ALL) File_Save_As("newname.txt",OK)

Description: **File_Save( )** saves the file being edited in the current edit buffer to disk and keeps it open for continued editing. The edit position in the file and any text markers are maintained.

File_Save(BEGIN) saves the file, but leaves the edit position at the beginning of the file, after saving. It is equivalent to the command sequence **File_Save( )Begin_Of_File( )**, but is much faster on huge files.

File_Save(NOMSG) suppresses the normal "Saving ..." message.

File_Save(ALL) saves all modified files in all edit buffers to disk and keeps them open for continued editing.

File_Save_As("*newname.txt***")** saves the current file under the new name '*newname.txt*' and allows further editing.

File_Save_As("*newname.txt***", OK)** suppresses the confirmation prompt if '*newname.txt*' already exists.

Notes: **File_Save( )** writes the file to disk. Therefore, if you quit with **Buf_Quit( )** sometime after a **File_Save( )** command, you will only abandon those changes made after the **File_Save( )** command. Those changes made before the **File_Save( )** command will have already been saved on disk.

File_Save( ) is equivalent to **{FILE, Save}**.

File_Save(ALL) is equivalent to **{FILE, Save all}**.

File_Save_As( ) is equivalent to **{FILE, Save As}**.

See Also: Commands: Buf_Close( ), File_Close( ), Is_Altered

FSIZE File_Size

Usage: Internal Value

Description: **File_Size** returns the size of the file in the current edit buffer. (This is the size the file would have if you now saved it.)

Examples: **Detab_Block(0,File_Size)** Convert all tabs in the entire file (buffer) to spaces.

FTRUNC File_Truncate( )

Options: OK

Usage: File_Truncate( ) FTRUNC(OK)

Description: (This is a technical, rarely used command!)

File_Truncate( ) truncates and closes the output file; it only saves text that has ALREADY been written to disk. **This command DOES NOT actually write any text to disk.** The user is prompted for confirmation to close the file.

File_Truncate(OK) skips the confirmation.

WARNING: Use this command with care! You can easily erase the file you are editing! In general, **File_Truncate( )** is only used to split large files into small ones and, for this, it is preceded by **File_Open_Write( )** and **File_Write( ) commands.**

Notes: Technically, a file has to be "written" and "closed" in order to save it on disk. The commands **Buf_Close( )** and **File_Close( )** write text to disk **AND** close the file. In contrast, **File_Truncate( )** only closes the file; it writes nothing. Text can be explicitly written to disk with the **File_Write( )** command.

Since the output file is initially opened with the file extension .*r*$$, the .*r*$$ 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 ".r$$" file is renamed to the true output filename. (See the **File_Open_Write( )** command notes.)

See Also: Commands: File_Open_Write( ), Buf_Close( ), File_Close( )

Examples:
```
File_Open_Write("save.txt")
File_Write(100)
File_Truncate(OK)
```
Write the first 100 lines of the edit buffer to the file "save.txt" and close it to save it.

FW File_Write(*n*)

Options: REVERSE

Usage: File_Write(20) FW(0) File_Write(1000,REVERSE)

Description: (This is a technical, rarely used command!)

File_Write(*n*) writes '*n*' lines from the beginning of the edit buffer to the output file, and deletes them from the buffer. **File_Write(0)** writes out the entire edit buffer up to the current line.

File_Write(*n*,REVERSE) writes the last '*n*' lines in the edit buffer to a temporary ".rR$" file. **File_Write(0,REVERSE)** writes out the

end of the edit buffer beginning with the current line. These commands perform the writing necessary for backward file buffering.

If no output file is open, the error "NO ASSIGNED FILENAME" is given and no text is written. The output file can be opened with the **File_Open_Write( )** or **File_Open( )** commands.

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

See Also: Commands: File_Read( ), File_Open( ), Mem_Free, File_Open_Write( ), Buf_Close( ), Is_Open_Write
Topics: "Explicit Read/Write Commands" in Chapter 3

Examples:

```
File_Open_Write("part1.txt")
File_Write(24)
File_Truncate(OK)
File_Open_Write("part2.txt")
Buf_Close( )
```

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", closing and saving the files.

Font_Charset (Windows only)
FONTH Font_Height
FONTW Font_Width

Usage: Internal Values

Description: **Font_Charset** returns the character set for the current display font. 0 = ANSI, 255= OEM. The DOS version always returns OEM.

Font_Height and **Font_Width** return the pixel height and width of one displayed character, based on the current display font. This includes any additional "LineSpacing" set in the **`vedit.ini`** file.

Since VEDIT is designed for fixed-width fonts, the width of every character is the same.

Notes: The only way to change the display font is with **{VIEW, Font}**. There is currently no macro language equivalent.

See Also: Commands: Desktop_Height, Desktop_Width, Win_Height, Win_Width
The topic "ANSI and OEM Characters" in Chapter 4 of the User's Manual

FP Format_Para(*n*)

Usage: Format_Para( ) FP(1) Format_Para(10)

Description: **Format_Para( )** re-formats the current paragraph of text using the currently configured right margin. If **{CONFIG, Word processing, Format from beginning of paragraph}** or **Config(W_F_BGN_PARA)** is set, it re-formats from the beginning of the paragraph. Otherwise (default), it re-formats beginning with

the line containing the edit position. After formatting, the edit position is advanced to the beginning of the next paragraph.

The paragraph will, by default, keep its current indentation. However, if **{CONFIG, Word processing, Format from left margin}** or **Config(W_F_LF_MARG)** is set, the leftmost column of the paragraph will be indented to the left margin.

Format_Para(*n*) re-formats the paragraph using a left margin of '*n*'. Use **Format_Para(1)** to use a left margin of one (1), independent of the current left margin.

Notes: **Format_Para( )** is equivalent to the **{EDIT, Format Paragraph}** function.

The paragraph will also be justified if **{CONFIG, Word Processing, Justify paragraphs}** or **Config(W_JUST_PARA)** is enabled.

If left-margin > right-margin, the command is ignored and no error given.

You must use a command loop to justify multiple paragraphs.

See Also: Commands: Margin_Left( ), Margin_Right( )
Topics: "Word Processing" in Chapter 4 of the User's Manual

Examples:

Format_Para(5)	Format the paragraph with the left margin starting at column 5.
Repeat (8) { Format_Para( ) }	Format eight paragraphs, using the current margins.

GE Get_Environment(*r*,"*name*")

Usage: Get_Environment(9,"VEDPATH")

Description: **Get_Environment(*r*,"*name*")** reads the current value of environment variable '*name*' into text register '*r*'.

Return: Returns the number of characters placed into register '*r*' (the length of the environment variable), or 0 if the variable does not exist.

Examples:

```
if (Get_Environment(9,"VEDPATH")==0) {
   Message("\nError - VEDPATH is not defined!")
}
```

Read the value of environment variable "VEDPATH" into register 9; displays an error message if it is not defined.

GF Get_Filename(*r*,"*filespec*")

Usage: Get_Filename(9,"*.txt") Get_Filename(9,@10)

Description: **Get_Filename(*r*,"*filespec*")** makes a file selection dialog box available to command macros. If '*filespec*' does not contain any of the wildcard characters "*" or "?", '*filespec*' is simply copied to text

register '*r*'. However if "*" or "?" are present, a file selection dialog box is displayed. The full pathname of the file selected by the user is then placed in register '*r*'.

Return: Returns the number of bytes that were placed into register '*r*'.

Notes: The dialog box is ***ONLY*** displayed if the '*filespec*' contains "*" or "?".

The filename in register '*r*' will be enclosed in double-quotes in order to support long filenames with embedded spaces. The register can then be used with the **File_Open( )** command.

Examples:
```
Get_Input(9,"Enter filename:",NOCR)
if (Reg_Size(9)==0) { Reg_Set(9,"*") }
Get_Filename(9,@9)
File_Open(@9)
```

Prompt for the name of the file to edit; if "*" or "?" or just **<Enter>** are entered, the file can be selected using a dialog box.

GI Get_Input(*r*,"*mtext*")

Options: APPEND, COUNT, INSERT, NOCR, STATLINE, TAB8

Usage: Get_Input(10,"Enter filename: ")

Description: **Get_Input(***r***,"***mtext***")** prompts on the screen with '*mtext*' and then reads the user's response line into text register '*r*'. '*mtext*' may be multiple lines long or may include "**\n**" to indicate new lines.

Get_Input(*r***,"***text***",STATLINE)** prompts on the status line — '*mtext*' must then be a single line prompt.

Get_Input(*r***,"***mtext***",TAB8)** expands any Tab characters in '*mtext*' assuming tab stops at every 8 columns.

The user ends the input line by pressing **<Enter>** which is also stored in the text register as a "newline"; **Get_Input(***r***, "***text***",NOCR)** does not store the "newline".

Get_Input(*r***,"***mtext***",COUNT,***n***)** limits the length of the user's input line to '*n*' characters.

Get_Input(*r***,"***mtext***","***default***")** prompts with an initial (default) input line of '*default*'.

The **APPEND** and **INSERT** options operate as with the **Reg_Set( )** command.

Return: Returns the number of bytes that were placed into register '*r*'.

See Also: Commands: Dialog_Input_1( ), Get_Key( ), Get_Num( )
Topics: "Input Commands" in Chapter 3

Examples: **Get_Input(20,"Enter Filename:",NOCR)**
File_Open(@20)

Prompt for a filename and save the filename in register 20. Then open the specified file for editing.

GK Get_Key(*"mtext"*)

Options: NOCANCEL, RAW, STATLINE, TAB8

Usage: #9=Get_Key("Press [CURSOR UP] or [CURSOR DOWN]:")

Description: **Get_Key("***mtext***")** prompts on the screen with '*mtext*' and returns the value of the next key pressed (or pending "key" character). '*mtext*' may be multiple lines long or may include "**\n**" to indicate new lines.

Normal characters return value 32 - 255. Function/control keys are decoded and return their function code which are listed in Appendix A. Unassigned function/control keys return the special two-letter-code "\B4\" = 'B'+'4'*256 = 13378.

Get_Key("*mtext***",RAW)** does not decode function/control keys. Control characters return 0 - 31. Function keys return their hardware "scan code" * 256.

Get_Key("*text***",NOCANCEL)** allows even the **[CANCEL]** key to be read (value = 'C'+'A'*256 = 16707); otherwise, **[CANCEL]** performs its normal function of breaking out of any command macro.

Get_Key("*mtext***",STATLINE)** prompts on the status line — '*mtext*' must then be a single line prompt.

Get_Key("*mtext***",TAB8)** expands any Tab characters in '*mtext*' assuming tab stops at every 8 columns.

Return: Returns the value of the next key pressed, as described above.

Notes: The next "key" is not necessarily the next key typed on the keyboard. It can also be a pending character from a keystroke macro, a **[REPEAT]** or **[REPEAT LAST]** function, an unprocessed keyboard character or from the mouse. Use **Key_Purge( )** if necessary to purge all pending "key" characters.

See Also: Commands: Get_Input( ), Get_Num,( ), Message( ), Key_Purge( )
Appendix A for a list of function codes.
On-line for **Get_Key( )** has another useful example.

Examples: **#98 = Get_Key("Press [CURSOR UP] or [CURSOR DOWN]:",STATLINE)**

Give the prompt on the status line. Decode the next function key. Register 98 has value 'C'+'U'*256 = 21827 if **[CURSOR UP]** was pressed; value 'C'+'D'*256 = 17475 if **[CURSOR DOWN]** was pressed.

GN Get_Num("*mtext*")

Options: STATLINE, SUPPRESS, TAB8

Usage: #15=Get_Num("Enter Line Number: ")

Description: Get_Num("*mtext*") prompts on the screen with '*mtext*' and returns the value of the numeric expression that is entered. '*mtext*' may be multiple lines long or may include "**\n**" to indicate new lines.

The user normally ends the number (expression) with **<Enter>**, but any invalid character also ends the number. Immediately pressing **<Enter>** returns "0".

Get_Num("*mtext***",TAB8)** expands any Tab characters in '*mtext*' assuming tab stops at every 8 columns.

Get_Num("*mtext***", STATLINE)** prompts on the status line; '*mtext*' must then be a single line prompt.

Get_Num("*text***",SUPPRESS)** returns the value of the simple decimal number; e.g. if "123+456" is entered, it returns 123. The rest of the input line is ignored.

Get_Num("*text***","***default***")** prompts with an initial (default) input line of '*default*'.

Return: Returns the value of the numeric expression entered, as described above. **Chars_Matched** is set to the number of characters evaluated; e.g. for "123" **Chars_Matched** is set to 3.

Notes: The response line may be edited in the same way as a dialog string.

See Also: Commands: Chars_Matched, Dialog_Input_1( ), Get_Input( ), Get_Key( ), Message( ), Num_Eval( )

Examples:

```
#15=Get_Num("Enter Line Number:")
Goto_Line(#15)
```

Prompt for the desired line number. Then moves the edit position to that line in the file.

Goto *label*

Usage: if (Return_Value==3) Goto EndMacro

Description: **Goto *label*** jumps to the label "**label:**" or "**:label:**" in the current text register (buffer). It cannot jump into the middle of a **While**, **Do-while**, **For** or **Repeat** loop. However, it can jump out of a loop or jump within the loop. VEDIT searches from the beginning of the currently executing text register (macro) for the label.

`***label***' can be a variable by using the contents of a text register as the entire label name or just part of the label name. (See examples.)

Notes: Labels with both colons are preferred; they execute a little faster.

See Also: Commands: Goto_Pos( ), Break, Return( ), Continue
Topics: "Flow Control - Break-out Commands" in Chapter 3
"Commenting Macros"

Examples:

Goto ERROR1	Jump to the label "ERROR1:".
Goto @(9)	Use the contents of T-Reg 9 as the entire label name. E.g. if T-Reg 9 contains "END", it jumps to the label "END:".
Goto OPTION-|@(9)	Use T-Reg 9 as part of the label name. E.g. if T-Reg 9 contains "A", it jumps to the label "OPTION-A:".

GC Goto_Col(*n*)
GL Goto_Line(*n*)
GM Goto_Marker(*m*)
GP Goto_Pos(*n*)

Usage: Goto_Pos(100) Goto_Pos(#98) Goto_Marker(4)

Description: **Goto_Col(***n***)** moves the edit position as close as possible to column '*n*' on the current line. If the desired column is in the middle of a Tab or other expanded character, it moves to the next character. If the desired column is past the end-of-line, it moves to the end of the line. Characters have a "width" according to the current display mode.

Goto_Col(*n***,EXTRA)** also forces the Visual Mode cursor to column 'n' so that **Cursor_Col** = '*n*'. It can be positioned past the end of the line. It should normally be immediately followed by **Visual( )**.

Goto_Line(*n***)** moves the edit position to the beginning of line '*n*'.

Goto_Marker(*m***)** moves the edit position to the previously set text marker '*m*'.

Goto_Pos(*n***)** moves the edit position to file position '*n*'. '*n*' is usually the value of a text marker (**Marker(***n***)**), block marker (**Block_Begin**, **Block_End**), or numeric register containing a computed position. Position "0" is the first character in the file.

Notes: **Goto_Col(***n***)** is equivalent to **{GOTO, Column #}**.

Goto_Line(*n***)** is equivalent to **{GOTO, Line #}**.

Goto_Marker(*n***)** is equivalent to **{GOTO, Goto text marker}**.

Goto_Pos(*n***)** is equivalent to **{GOTO, File position}**. **Goto_Pos(0)** is equivalent to **Begin_Of_File( )**.

See Also: Commands: Char( ), Set_Marker( ), Cur_Pos

Examples:

Goto_Pos(Marker(1))	Jump to the position in the file saved in text marker "1".

H Help( )

Usage: H Help("file_open") H("command")

Description: **Help( )** starts up the on-line help system with the topic "Commands" which gives a listing of all commands.

Help("*topic***")** starts up the on-line help at the topic '*topic*'.

Notes: The DOS/QNX/Linux version's **`vphelp.hlp`** file is user changeable and expandable. See the on-line help topic "ONLINE" for details.

Examples: **Help("file_open")** Starts up the on-line help and immediately accesses the **File_Open( )** command topic.

IC Ins_Char(*n*)
II Ins_Indent(*n*)
IN Ins_Newline(*n*)

Options:

COUNT, OVERWRITE

Usage: Ins_Char(12,OVERWRITE) IN(2) II(40)

Description: **Ins_Char(*n*)** inserts the character with decimal value '*n*' at the edit position. This is useful for inserting control characters with a decimal value between 0 and 31 and extended/graphics characters with a decimal value between 128 and 255.

Ins_Char(*n*,OVERWRITE) overwrites the existing character.

Ins_Char(*n*,COUNT,*n2*) repeats the insertion '*n2*' times. The count value can be zero -- it then inserts nothing.

Ins_Newline(*n*) inserts '*n*' "newlines" (blank lines) and advances the edit position. The actual "newline" depends upon the buffer's file type and can be a single Carriage-Return (Mac), Line-Feed (Unix) or both (Windows/DOS).

Ins_Newline(0) inserts nothing; therefore, **Ins_Newline(*c*)** will insert a "newline" only if condition '*c*' is TRUE.

Ins_Indent(*n*) inserts the optimum number of tabs and spaces to reach column '*n*'. If the edit position is already past column '*n*', it does nothing. If **{CONFIG, Emulation, Expand <Tab> with spaces}** (**Config(E_EXP_TAB)**) has Mask-1 set, only spaces are inserted to reach the indent column.

See Also: Commands: Ins_Text( ), Out_Ins( ), Newline_Chars
Topics: "Entering Control Characters" in Chapter 3
[ENTER CTRL] function

Examples: **Ins_Char(#20)** Insert the character whose value is in numeric register 20.

IC(132) Insert a graphics character into the text.

IM Insert_Mode(*n*)

Options: CLEAR, SET

Usage: Insert_Mode(SET) Insert_Mode

Description: **Insert_Mode(SET)** or **Insert_Mode(1)** enables "Insert" mode for Visual Mode editing.

Insert_Mode(CLEAR) or **Insert_Mode(0)** puts Visual Mode into "overstrike" mode.

Insert_Mode, without parameters, just returns the current value.

Return: Returns a value of 1 if in insert mode; 0 if in overstrike mode.

INSF Ins_File("*file*",*n1,n2*)]

Options: BEGIN, COLUMN

Usage: Ins_file("file.txt",1,100) INSF("file.txt")

Description: **Ins_File("***file***")** inserts a specified line number range of the file '*file*' at the current edit position. If no line range is specified, the entire file is inserted. The edit position is advanced past the inserted text.

Ins_File("*file***",BEGIN)** positions the cursor at the beginning of the inserted text (the original edit position.)

Ins_File("*file***",1,ALL,COLUMN)** inserts the file as a columnar block.

VEDIT's automatic (virtual) file buffering normally allows even large (multi-megabyte) files to be inserted with **Ins_File( )**.

Notes: The line numbers of a file can be displayed with **Type_File( )**.

See Also: Commands: File_Read( ), Type_File( ), File_Open_Read( )
Topics: "Using Text Registers in Filenames" in Chapter 2

Examples: **Ins_File("library.asm",34,65)**

Lines 34 through 65 of the file LIBRARY.ASM are inserted at the current edit position.

II Ins_Indent( ) - See Ins_Char( )

IN Ins_Newline( ) - See Ins_Char( )

IT Ins_Text("*text*")

Options: COUNT, OVERWRITE

Usage: Ins_Text("a word") Ins_Text(" ",COUNT,30)
IT("overlay",OVERWRITE)

Description: **Ins_Text("***text***")** inserts '*text*' at the current edit position. '*text*' may be several lines long with **<Enter>** at the end of each line, which then inserts the "newline" character(s), depending upon the current file type. (Under Windows/DOS, the "newline" is normally two characters - Carriage-Return and Line-Feed.) The edit position is advanced past the inserted text.

Ins_Text("*text***",OVERWRITE)** overwrites the existing text with the new text.

Ins_Text("*text***", COUNT,***n*)** inserts the text '*n*' times.

Notes: The **<Tab>** key is not expanded with spaces as is optional in Visual Mode. It is often easier to insert control characters with the **Ins_Char(n)** command.

See Also: Commands: Ins_Char( ), Out_Ins( )
Topics: "Entering Control Characters" in Chapter 2

Examples: **Ins_Text("<Enter><Tab>",COUNT,200)**

Insert 200 new lines, each beginning with a Tab character. (The command is two lines long.)

Ins_Text("overwrite",OVERWRITE)

Overwrite the existing text at the current edit position with the text "overwrite".

IA Is_Altered
IAF Is_Altered_File
IAE Is_Auto_Execution
IDKO Is_Disk_Open
Is_Expired
Is_File_Selector
ILF Is_Long_Filename
Is_Mono
INF Is_New_File
IOR Is_Open_Read
IOW Is_Open_Write
IO Is_Option(*x*)
Is_OS2
Is_Quiet
ISRI Is_Redirect_Input
Is_Saveas
ISV Is_Startup_Vdm
Is_Support
Is_VSWAP
Is_Windows
Is_Win32_Version
Is_WinNT
Is_Zoomed

Usage: Internal Values

Description: **Is_Altered** returns 1 (TRUE) if the current edit buffer has been altered since the last file save.

Is_Altered_File returns 1 (TRUE) if the current file (buffer) has been altered since it was opened. Saving the file does not change this flag.

Is_Auto_Execution returns 1 (TRUE) if a "**-x**" invocation macro is currently running.

Is_Disk_Open returns 1 (TRUE) if the current edit buffer is being used for disk sector editing, i.e. a disk is open.

Is_Expired returns 1 (TRUE) if the support period for the VEDIT product is expired, of if VEDIT is running as a trial version and the trial period is expired.

Is_File_Selector returns 1 (TRUE) if the File-selector window is currently displayed. It is enabled with **{VIEW, File selector}**.

Is_Long_Filename returns 1 (TRUE) if long filenames are being used, e.g. Windows 95. Long filename support can be turned off with the "**-d**" invocation option.

Is_Mono returns 1 (TRUE) if VEDIT is using its monochrome screen attributes (colors), i.e. if **{CONFIG, Colors, Enable monochrome}** (**Config(S_MONO)**) is set. This is true if the adapter card is a monochrome, if the "**-m**" invocation option was specified, or **Screen_Mono( )** was executed.

Is_New_File returns 1 (TRUE) if the current file was created, i.e. it did not exist when it was opened.

Is_Open_Read returns 1 (TRUE) if the input file is open in the current edit buffer. Note that the input file is closed when the end of the input file has been read.

Is_Open_Write returns 1 (TRUE) if the output (write) file is open in the current edit buffer.

Is_Option(*x***)** returns 1 (TRUE) if invocation option "-*x*" was specified. This permits a macro to implement custom invocation options. For example, if VEDIT was invoked with the "**-y**" option, **Is_Option(y)** returns 1 (TRUE).

Is_OS2 returns 1 (TRUE) if VEDIT is currently running under the OS/2 operating system.

Is_Quiet returns 1 (TRUE) if VEDIT was invoked with the "**-q**" option and is therefore running quietly without any screen display. It returns 2 if the Windows version is currently minimized.

Is_Redirect_Input returns 1 (TRUE) if "keyboard" input to the macro language commands **Get_Input**, **Get_Key**, **Get_Num**, etc. is being redirected from a file.

Is_Saveas returns 1 (TRUE) if the output file is different from the input file due to opening the file with the "**-a**" option, using **{FILE, Save as}** or using **File_Open_Write( )**.

Is_Startup_Vdm returns 1 (TRUE) if the `startup.vdm` macro was loaded and executed at startup. This indicates that the configuration settings loaded from VEDIT.CFG and VEDIT.KEY may have been changed.

Is_Support returns the product support expiration date for an active licensed VEDIT; the year is returned in the lower four hex digits, the month in the next two hex digits. **Is_VSWAP** (DOS only) returns 1 (TRUE) if VEDIT detects that V-SWAP is installed in memory; otherwise it returns 0 (FALSE).

Is_Windows returns the version number of the Microsoft Windows operating system when running under Windows, e.g. 400 (Win95), 410 (Win98), 500 (Win2000) or 501 (WinXP). Otherwise, it returns 0 (FALSE).

Is_Win32_Version returns 1 (TRUE) if the 32-bit Windows version of VEDIT is running under Windows 95/98/ME or NT/2000/XP. See also **OS_Type.**

Is_WinNT returns 1 (TRUE) if the 32-bit Windows version of VEDIT is running under Windows NT/2000/XP.

Is_Zoomed returns 1 (TRUE) if the editing windows within VEDIT are zoomed (maximized).

Itoa(*n,r*)
NS Num_Str(*n,r*)

Options: EXTRA, FILL, FORCE, HEX, LEFT

Usage: Itoa(#10,11) Itoa(369+454,11)

Description: **Itoa(***n,r***)** places the ASCII conversion of numeric expression '*n*' into text register '*r*'. I.e., it performs an integer to ASCII conversion.

It is similar to **Out_Reg(***r***) Num_Type(***n***) Out_Reg(CLEAR)**; however no "newline" is appended to the end of the register.

The **LEFT**, **FILL**, **FORCE**, **EXTRA** and **HEX** options for the **Num_Type( )** apply to **Itoa( )** too. They control how the ASCII number will be formatted within register '*r*'.

Return: Returns the number of character placed into register '*r*'.

Notes: **Itoa( )** and **Num_Str( )** are two names for the same command.

See Also: Commands: Atoi(), Num_Eval( )

KA Key_Add("*key-seq*","*Edit-seq*")

Options: INSERT, OK

Usage: Key_Add("Alt-A","[VISUAL EXIT] BOL Block_Copy(1)")

Description: **Key_Add("***key-seq***","***Edit-seq***")** adds the new key assignment for '*key-seq*' to the end of the keyboard layout table.

Key_Add(...,OK) skips the confirmation to overwrite an existing assignment to 'key-seq'.

Key_Add(...,INSERT") inserts the key assignment at the beginning of the keyboard layout table. This overrides any existing assignment to '*key-seq*', without removing the existing assignment.

Notes: Use **{CONFIG, Keyboard layout, Save layout}** or the **Key_Save( )** command to make any keyboard layout changes permanent.

DOS: Use **{CONFIG, Misc, Save into VEDIT}** or the **Config_VEDIT ( )** command to save configuration changes into the executable VEDIT.EXE file.

See Also: Commands: Key_Delete( ), Key_Save( )
Topics: "Modify Keyboard Layout" in Chapter 3

Examples: **Key_Add("Alt-A","[VISUAL EXIT] BOL Block_Copy(1)")**

Add a keystroke macro which defines **<Alt-A>** to be a function which duplicates the current line of text.

Key_Cfg_Pop( )
Key_Cfg_Push( )

Usage: Key_Cfg_Push( ) Key_Cfg_Pop( )

Description: **Key_Cfg_Push( )** saves the current keyboard layout by "pushing" it onto the same text register stack as used by **Reg_Push( )**.

Key_Cfg_Pop( ) restores the previous keyboard layout by "popping" it from the text register stack.

Notes: These commands allow a macro to save the current keyboard layout, modify it for the macro, and then restore the layout when the macro is done.

These commands must be used with care, or you may crash VEDIT. In particular, these commands must be balanced with any **Reg_Push( )** and **Reg_Pop( )** commands.

In order to provide some protection again accidentally crashing VEDIT, **Key_Cfg_Pop( )** checks that the first two bytes about to be popped are "02 FF" (hex); if not it gives a "NESTING (STACK) ERROR".

These commands stop any keystroke macro that might be running.

See also: Commands: Reg_Push( ), Win_Cfg_Push( )

KD Key_Delete("*key-seq*")

Options: NOERR, REVERSE

Usage: Key_Delete("Alt-A") Key_Delete("[MENU]FS",REVERSE)

Description: **Key_Delete("***key-seq***")** deletes any keyboard assignment to '*key-seq*'. In case the keyboard layout table contains multiple assignments to this key, only the first one in the table is deleted.

Key_Delete("*key-seq***",NOERR)** suppresses the error, which is otherwise given, if '*key-seq*' is not assigned.

Key_Delete("*edit-seq***",REVERSE)** deletes the keyboard layout entry which assigns any key to the specified "edit-sequence". (If the layout contains multiple assignments to the same edit-sequence, only the first is deleted.)

Notes: See the notes for **Key_Add( )**.

See Also: Commands: Key_Add( ), Key_Save( )
Topics: "Modify Keyboard Layout" in Chapter 3

Examples: **Key_Delete("Alt-A",NOERR)**

Delete any key assignment to **<Alt-A>**.

Key_Delete("[MENU]FS",REVERSE+NOERR)

Remove any hot-key assigned to **{FILE, Save}**.

Key_Jam (DOS Only)

Usage: Key_Jam('A') Key_Jam(0x1600)

Description: **Key_Jam(** *n***)** enters (jams) the keyboard character or scan-code '*m*' into the hardware keyboard buffer. Simulates a user typing. The keyboard buffer can usually only hold 16 characters. Values are:

1 - 255	Control, ASCII and graphics characters
Scan-code * 256	Function keys.

See Also: Commands: Get_Key(), Key_Purge()
On-line help: Topic "USER" has a complete example.

Examples: **Key_Jam('A')** Simulates user typing "A".

Key_Jam(0x3B00) Simulates user typing <F1>.

Key_List( )

Usage: Key_List( )

Description: **Key_List( )** displays (lists) the entire keyboard layout, including any keystroke macros. It has the same format as the **`vedit.key`** file saved with Key_Save( ).

See Also: Commands: Key_Add( )
Topics: "Modify Keyboard Layout" in Chapter 3

Examples: **Buf_Switch(Buf_Free(EXTRA))**
Out_Ins() Key_List() Out_Ins(CLEAR)
Search("|lt-T",BEGIN+NOERR)

Insert the keyboard layout into an "extra" buffer and search for an assignment to "Alt-T".

KL Key_Load("*file*")

Options: NOERR

Usage: Key_Load("special.lay") KL("vedit.key")

Description: **Key_Load("***file***")** loads a new keyboard layout table from the file '*file*'. If the file is not found in the current directory, the *VEDIT Home Directory* will be searched.

Key_Load("*file***",NOERR)** suppresses the error message if '*file*' is not found.

The loaded keyboard layout table can be in either the "text" or "binary" format; the "binary" format loads much faster. When loading a "text" formatted table, it is checked for syntactic errors. Errors are reported as "Error in Keyboard file, Line # *nnn*".

Notes: A keyboard layout table is saved to disk with **{CONFIG, Keyboard Layout, Save to Disk}** or with the **Key_Save( )** command. It includes any added keystroke macros. Keyboard layout tables saved in "text" format can easily be edited as a normal file.

Loading a new keyboard table overwrites any existing keystroke macros. Upon entering Visual Mode the keyboard table is checked for validity. An invalid table gives the error "KEYBOARD LAYOUT CORRUPTED".

See Also: Commands: Visual( ), Key_Save( ), Reg_Load( )
{CONFIG, Keyboard Layout} menu
Topics: "Modify Keyboard Layout" in Chapter 3

Examples: **Key_Load("special.key")** The file SPECIAL.KEY replaces the current keyboard layout table including any keystroke macros.

Key_Pop(*n*)

Usage: Key_Pop(5)

Description: **Key_Pop(*n*)** pops (removes) the first '*n*' entries from the keyboard layout table. These are normally "temporary" assignments inserted with the **Key_Add(...,INSERT)** command; therefore '*n*' is equal to the number of these preceding **Key_Add( )** commands.

Notes: Nothing prevents **Key_Pop(*n*)** from removing all or part of the original keyboard layout. The command **Key_Pop(ALL)** removes all key assignments, in effect initializing the keyboard layout.

See Also: Commands: Key_Delete( )
Topics: "Modify Keyboard Layout" in Chapter 3

Examples: See the example under "Modify Keyboard Layout" in Chapter 3.

Key_Purge( )

Usage: Key_Purge( )

Description: **Key_Purge( )** purges any pending keystrokes. This includes not only unprocessed keyboard characters, but pending characters due to a keystroke macro, and the **[REPEAT]** and **[REPEAT LAST]** functions.

Notes: **Key_Purge( )** can be used before **Get_Key( )** to ensure that **Get_Key( )** reads only the next keyboard character.

Key_Purge( ) can be used after **Get_Key( )** to purge extra characters resulting from a keystroke macro.

KRM Key_Record_Mode( )

Usage: Key_Record_Mode(1) Key_Record_Mode

Description: **Key_Record_Mode(*n*)** sets the special **[REPEAT LAST]** record mode to '*n*'.The internal value **Key_Record_Mode** returns the current value. The values are:

0 Keystroke recording is turned off; **[REPEAT LAST]** is disabled.

1 (Normal) Keystroke recording is automatically reset by each Visual Mode editing operation. **[REPEAT LAST]** will replay just the last Visual Mode editing operation.

2 Keystroke recording is enabled — all keystrokes are recorded until **Key_Record_Mode(0)** is executed. **[REPEAT LAST]** will replay all the recorded keystrokes.

Return: Returns the value of the special record mode.

Notes: **[REPEAT LAST]** can record up to 128 keystrokes. The special modes are used in our "vi" editor emulation and probably have limited use elsewhere.

KSAVE Key_Save("*file*")

Options: BINARY, OK

Usage: Key_Save("SPECIAL.KSR")

Description: **Key_Save("*file*")** saves the current keyboard layout table, including any keystroke macros, in the file '*file*'. It is saved in a "text" format which is very easy to edit as a normal file. The command option **Key_Save("*file*",BINARY)** saves the layout in a "binary" format which is not easily edited, but loads much faster.

Key_Save("*file*",OK) skips the confirmation prompt when '*file*' already exists.

Notes: **Key_Save( )** is similar to **{CONFIG, Keyboard Layout, Save to a file}**.

See Also: Commands: Key_Load( )
{CONFIG, Keyboard Layout} menu
Topics: "Modify Keyboard Layout" in Chapter 3

Examples: **Key_Save("VEDIT.KEY")** Save the current keyboard layout and keystroke macros to disk.

KSTAT Key_Status
KT Key_Total

Usage: Internal Values

Description: **Key_Status** returns 1 if the keyboard has a character ready. Otherwise, it returns 0. Note that "keystrokes" can also come from keystroke macros, and the **[REPEAT]** and **[REPEAT LAST]** functions.

Key_Total returns the total number of keyboard characters typed since starting VEDIT.

See Also: Commands: Key_Purge( )

LSP Last_Search_Pos

Usage: Internal Value

Description: **Last_Search_Pos** returns the file position at the time of the last Search/Replace. **Go_Pos(Last_Search_Pos)** returns to that position.

L Line(*m*)

Options: ERRBREAK, NOERR

Usage: Line(120) Line(-14) Line(0)

Description: **Line(***n***)** moves the edit position by '*n*' lines, to the beginning of the '*n*'th following line.

Line(-*n***)** moves the edit position to the beginning of the '*n*'th preceding line.

Line(0) moves the edit position to the beginning of the current line.

Attempting to move past either end of file leaves the edit position at the respective end and gives the error "END OF BUFFER REACHED". **Line(***m***,NOERR)** suppresses the error and continues with the next command.

Line(*m***,ERRBREAK)** suppresses the error message and performs a **Break** to exit any **While**, **Do-while**, **For** for **Repeat** loop.

Return: Always returns a value of 1. Sets **Error_Flag** if it attempts to move past either end of file.

See Also: Commands: Char( ), Type( ), Cur_Line
Topics: "Processing End-Of-File Condition" in Chapter 3

Examples:

Line(2200)	Move down 2200 lines in the file. E.g., if the edit position was at the beginning of the file, this moves to the beginning of line 2201.
Line(100-Cur_Line)	Move to the beginning of line 100 from anywhere in the file.

MN Macro_Num

Usage: Internal Value

Description: **Macro_Num** returns the ID number of the text register currently executing as a command macro. It returns buffer number +128 (+BUFFER) if an edit buffer is executing. It returns 127 if it is executing from the "COMMAND" prompt. It returns 255 if it is executing from a keystroke macro.

ML Margin_Left(*n*)
MR Margin_Right(*n*)

Usage: Margin_Left(10) ML(1) Margin_Right(70) MR(0)

Description: **Margin_Left(***n***)** sets the left margin to '*n*' for just the current editor buffer. **Margin_Left(1)** restores the default left margin of 1.

Margin_Right(*n***)** sets the right margin to '*n*'. **Margin_Right(0)** turns off the right margin. Setting the right margin also enables word wrap.

Return: **Margin_Left** returns the value of the left margin setting. **Margin_Right** returns the value of the right margin setting.

Notes: **Margin_Left(***n***)** and **Margin_Right(***n***)** are more convenient than the equivalent commands **Config(W_RT_MARG,*n*,LOCAL)** and **Config(W_LF_MARG, *n*,LOCAL)**. However, to change margins for all edit buffers, you must use the **Config( )** command.

See Also: Commands: Config( )
Topics: "Word Processing Functions" in Chapter 4 of the VEDIT User's Manual

Examples:

Margin_Right(70)	Set the right margin at 70 (which also enables word wrap).

Marker(*m*)

Usage: Marker(9)

Description: **Marker(***m***)** returns the position of text marker '*m*' in the file. It returns -1 if the marker is not set. This can be set in Visual Mode or with **Set_Marker( )**.

MW Mark_Word

Usage: Mark_Word()

Description: **Mark_Word( )** marks the word at the edit position as a block of text, setting both **Block_Begin** and **Block_End**. If the edit position is on whitespace past a word, the block will include the whitespace, otherwise whitespace after the word is not included.

Notes: This command is equivalent to **{BLOCK, Select word}**.

Match("*ss*")

Options: Listed below

Usage: Match("align") Match("|<begin") Match("|W",ALL)

Description: **Match("***ss***")** compares (matches) the text at the edit position with the search string '*ss*' and returns the results of the match. The match is only successful if the entire '*ss*' matches. '*ss*' may contain pattern matching codes or regular expressions.

Match(@*r*) matches the text at the edit position with the search string in text register '*r*'. Notes: '*r*' must not be empty or VEDIT will give a "BAD PARAMETER" error. Unless '*r*' contains pattern matching codes, **Compare(*r*)** is usually preferable.

Match Command options:

ADVANCE	Advances the edit position past the matching characters, but only if the entire match is successful.
CASE	Match is case sensitive.
WORD	Match is restricted to entire words (surrounded by separators.
COUNT, *n*	Match for the '*n*'th occurrence of '*ss*'.
REGEXP	Match using regular expressions, using "minimal" matching.
REGEXP+MAX	Match using regular expressions, using maximized matching.
ALL	Matches as many consecutive occurrences of '*ss*' as possible. It is usually used together with the **ADVANCE** option.

Return: The results of the comparison are returned and saved in **Return_Value**:

0 The comparison is successful, the entire '*ss*' matches.
1 The text is lexically "greater than" '*ss*'.
2 The text is lexically "less than" '*ss*'.
3 The mismatch occurred on a pattern matching code in '*ss*'.

If the comparison is successful, **Chars_Matched** is set to the number of matching characters. If not successful, **Error_Flag** and **Error_Match** are set to 1 (TRUE).

Notes: **Match("***ss***")** is different from **Search("***ss***")** in that **Search( )** will search for the string, while **Match( )** fails if it does not completely match the text at the current edit position. Also, no error message is given if **Match( )** is unsuccessful. Use the command form **Match(@*r*)** when the search/match string is contained in text register '*r*'.

See Also: Commands: Compare( ), Search( ), Reg_Compare( ), Return_Value, Chars_Matched, Error_Flag
Topics: "Match and Compare" in Chapter 3

Examples: **Match("|<Begin")** Check if the edit position is at the beginning of a line beginning with the word "begin".

MI Match_Item

Usage: Internal Value

Description: **Match_Item** returns the item number that matched in a search using the "**|{...}**" pattern matching code. See Pattern Matching in the VEDIT Users Manual. For example, if the command **Search("|{cat,dog, lion,mouse}")** finds "lion", **Match_Item** returns 3.

MP Match_Paren( )

Options: ERRBREAK, NOERR

Usage: Match_Paren( ) MP(NOERR)

Description: **Match_Paren( )** searches (forwards) for the next grouping character — "{", "}", "[", "]", ">", "<", "(", ")". If the edit position is already at one of these characters, it searches (forwards or backwards) for its "matching" character. It supports nested "parentheses". The command is primarily useful for checking the syntax of structured programming languages such as "C" and VEDIT macros.

Match_Paren(NOERR) suppresses the error message if no initial or matching "parenthesis" can be found; in this case **Error_Flag** and **Error_Match** are set.

Match_Paren(ERRBREAK) also suppresses the error message, but performs a **Break** out of any command loop.

Notes: **Match_Paren( )** is equivalent to**{GOTO, Match Parentheses}**.

Max(*n,m*)
Min(*n,m*)

Usage: Max(Block_Begin,Block_End) Min(Block_Begin,Block_End)

Description: **Max(***n,m***)** returns the greater of the two numeric values '*n*' and '*m*'.

Min(*n,m***)** returns the lesser of the two numeric values '*n*' and '*m*'.

Notes: VEDIT can distinguish between the **Max( )** command and the **MAX** numeric value by its context.

Examples: **Min(Block_Begin,Block_End)**

Return the position of the first character in the highlighted block. This handles the case where a block is highlighted in the reverse direction, i.e. where **Block_Begin** is the end of the block.

MF Mem_Free(*n*)

Usage: Mem_Free(40000) MF(1) Mem_Free

Description: **Mem_Free(*n*)** makes '*n*' bytes of memory free in the edit buffer, if possible, by buffering some of the file to disk. The command **Mem_Free(1)** tries to squeeze the edit buffer down to approximately 8 Kbytes in size.

Mem_Free(*n*) does not buffer out any text which is within 2000 bytes of the edit position. If the desired amount cannot be written to disk with just forward file buffering, it will also use backward file buffering. If '*n*' is too large, it will make as much memory free as possible.

Mem_Free, without parameters, only returns the number of bytes free in the edit buffer.

Return: Returns the (new) number of bytes free in the current edit buffer, after any buffering was performed.

Notes: No indication is given if the desired amount of memory could not be made available. Use the returned value, to confirm how much is free.

The primary purpose for **Mem_Free(*n*)** is to maintain compatibility with macros written for older versions of VEDIT. It was typically needed to make a large block of memory free before certain edit operations. However, the current VEDIT will generally perform the equivalent file buffering automatically as needed.

See Also: Commands: Mem_Status, Buf_Switch( )
Topics: "Explicit Read/Write Commands" in Chapter 3

Examples: **Mem_Free(40000)** Make 40000 bytes free in the edit buffer.

MSTAT Mem_Status

Usage: Mem_Status MSTAT

Description: **Mem_Status** displays the number of memory bytes free in the current edit buffer, followed by the number of bytes in the edit buffer, followed by the combined number of bytes in all the text registers.

See Also: Commands: Reg_Status, Mem_Free

Examples:

File_Read(100) **Mem_Status**	An additional 100 lines from the input file are appended and the remaining number of free bytes displayed.

M Message("*mtext*")
STATM Statline_Message("*itext*")

Options: EXTRA, STATLINE, TAB8

Usage: Message("Part 1 is done\n")
Message("Press \<CA> to Cancel",EXTRA)
Statline_Message("Processing is done.")

Description: **Message("***mtext***")** displays '*mtext*' on the screen at the current window position. '*mtext*' may be several lines long with **<Enter>** at the end of each line or may contain "**\n**" to indicated new lines. The command is often used in command macros to display messages, menus, forms and prompts.

Message("...\<*vm*>**...",EXTRA)** displays and highlights the name of the key assigned to the edit function '*vm*'. This allows a prompt to reflect the current keyboard layout. Appendix A lists the vm-codes.

Message("*mtext***",STATLINE)** displays the text on the status line — '*mtext*' must then be a single line prompt.

Message("*mtext***",TAB8)** expands any Tab characters in '*mtext*' assuming tab stops at every 8 columns.

Statline_Message("*itext***")** displays '*itext*' on the status line; it will remain until the next keystroke. '*itext*' can contain "**|@(*r*)**" to use the contents of text register '*r*'.

Notes: The **Win_Hor( )** and **Win_Vert( )** cursor positioning commands can be used to display the text at any position in the window.

See Also: Commands: Get_Input( ), Get_Key( ), Get_Num( ), Out_Print( )
Topics: "Interactive Input and Output" in Chapter 3

Examples: **Out_Print Message("Chapter 1\n")**

The header line "Chapter 1" is printed and the printer is advanced to a new line.

Min( ) - See Max( )

Mouse_Active (DOS Only)

Usage: Internal Value

Description: **Mouse_Active** returns a non-zero value if the mouse is active. It returns 0 if the mouse is not active, either because the mouse driver is not installed or because **{CONFIG, Misc, Mouse cursor}** is set to "0".

ND Name_Dir( )

Options: NOMSG, NOCR

Usage: Name_Dir(NOMSG) ND

Description: **Name_Dir( )** displays the current drive and directory. The command option **Name_Dir(NOMSG)** omits the "**Directory: "** header, and the option **Name_Dir(NOCR)** omits the following "newline".

Notes: The current directory can also be displayed with **Chdir**.

Examples: **Out_Reg(20) Name_Dir(NOMSG+NOCR) Out_Ins(CLEAR)**

Insert the current drive and directory into text register 20.

NF Name_File( )

Options: EXTRA

Usage: Name_File( )

Description: **Name_File( )** displays the names of both the input and output files. **Name_File(EXTRA)** includes the complete drive and path of the input and output files.

See Also: Commands: Name_Dir( )

NR Name_Read( )
NW Name_Write( )

Options: EXTRA, NOCR, NOMSG

Usage: Name_Read( ) NR(NOCR+NOMSG) NW(EXTRA)

Description: **Name_Read( )** displays the current buffer's input filename.

Name_Write( ) displays the current buffer's output filename. The command option "EXTRA" also displays the drive and path of the file. The command option "NOMSG" omits the "**Input/Output file:**" header, and the option "NOCR" omits the following "newline".

See Also: Commands: File_Open( ), Is_Open_Read
Topics: "Explicit Read/Write Commands" in Chapter 3

Examples: **Out_Reg(50) Name_Read(EXTRA+NOMSG+NOCR)**
Out_Reg(CLEAR)
Out_Reg(51) Name_Write(EXTRA+NOMSG+NOCR)
Out_Reg(CLEAR)

Insert the full path and filename of the current input file into register 50, and the current output file into register 51.

NC Newline_Chars
NTS Next_Tab_Stop
N_Option
Num_All_Bufs
Num_Edit_Bufs

Usage: Internal Values

Description: **Newline_Chars** returns the number of characters expected in the "newline" sequence at the end of each text line. It returns 1 when the file type is set for UNIX or Macintosh; 2 when configured for Windows/DOS; 0 for fixed-length records, e.g. record/binary mode. The file type is determined by **{CONFIG, File handling, File type}** (same as **Config(F_F_TYPE)**).

Next_Tab_Stop returns the column position of the next tab stop based on the current value of **Cur_Col**. It returns 0 if no more tab stops are defined.

N_Option returns the value of the number following the "**-n**" invocation option. If "**-n**" was not specified, it returns -1. This is an easy way to pass a numeric value to a macro when starting up VEDIT.

N_Option(*n***)** forces the value of **N_Option** to '*n*'. This is primarily used when the macro called by **Call_File( )** expects a parameter to normally be set with the "**-n**" invocation option. E.g.

```
N_Option(80)                    //Record length is 80
Call_File(100,"convert.vdm")    //Convert fixed length
                                //records to CR+LF
```

Num_All_Buffers returns the maximum number of all edit buffers in VEDIT including "extra" buffers. It is 125 for the Windows version and 36 for the DOS version. Permits a macro to account for future versions of VEDIT which may have more buffers.

Num_Edit_Buffers returns the maximum number of normal edit buffers in VEDIT that can be used for file editing. It is 99 for the Windows version and 32 for the DOS version.

See Also: Commands: Config_Tab( ), Previous_Tab_Stop( )

NDS Num_Display(*x,y*)

Options: ALL, COUNT, NOMSG

Usage: Num_Display(0,109,ALL+COUNT,10)

Description: **Num_Display(***x*,*y***)** displays the non-zero numeric registers '*x*' through '*y*' in the format "#xx = value", one register per line.

Num_Display(*x*,*y***,ALL)** displays the registers even if their value is zero. The command option "NOMSG" is the same as "ALL", but omits the heading "#xx = ".

Num_Display(*x*,*y*,**COUNT,***n***)** displays the registers in '*n*' columns.

See Also: Commands: Num_Type( )

Examples: **Num_Display(0,127,ALL+COUNT,10)**

Display the values of all numeric registers in ten columns.

NE Num_Eval( )

Options: ADVANCE, SUPPRESS

Usage: Num_Eval(ADVANCE)

Description: **Num_Eval( )** evaluates the numeric expression at the edit position and returns its value. For example, with the text "123+45/5", **Num_Eval( )** will return 132.

Num_Eval(ADVANCE) advances the edit position past the numeric expression.

Num_Eval(SUPPRESS) limits evaluation to a simple number. That is, with the text "123+45/5", it will return 123.

Return: Returns the value of the numeric expression evaluated. **Chars_Matched** is set to the number of characters in the expression.

See Also: Commands: Atoi(), Get_Num() Num_Eval_Reg()

NED Num_Eval_Date( ) (Windows Only)
NID Num_Ins_Date(*n*)

Options: ADVANCE, BEGIN, EXTRA, NOCR, VALUE

Usage: Num_Eval_Date( )

Description: **Num_Eval_Date( )** evaluates a date mm/dd/yyyy or mm-dd-yyyy as the number of days since 01-01-0001, assuming the Gregorian calendar and leap years, and returns the number.

Num_Eval_Date(BEGIN) evaluates a date dd/mm/yyyy or dd-mm-yyyy.

Num_Eval_Date(EXTRA) evaluates mm/dd/yy as mm/dd/19yy.

Num_Eval_Date(ADVANCE) advances the edit position past the date.

Num_Ins_Date(*n***)** inserts '*n*' as a date mm-dd-yyyy into the edit buffer and advance the edit position, where '*n*' is the number of days since 01-01-0001 assuming the Gregorian calendar and leap years.

Num_Ins_Date(*n***,BEGIN)** inserts the date as dd-mm-yyyy.

Num_Ins_Date(*n***,NOCR)** suppresses the "newline" following the inserted date.

Num_Ins_Date(*n*,VALUE,'/') inserts the date as mm/dd/yyyy. Any desired separator character can be specified.

Return: **Num_Eval_Date( )** returns the evaluated date as an integer. **Chars_Matched** is set to the number of characters in the date.

Notes: Every 4th year is a leap year except when evenly divisible by 100; however years evenly divisible by 400 are leap years. Therefore 1900 was not a leap year, but 2000 was.

Historical dates before 1582 may be off by 11 days due to the switch in 1582 from the Julian to Gregorian calendars.

Subtract 693596 to determine the number of days since Jan. 01, 1900.

See Also: Commands: Date( ), Num_Eval( ), Num_Ins( )

Examples: The following macro searches the current file for every date, assuming the mm/dd/yyyy format, and advances the date by seven days.

```
BOF()
while( ! At_EOF ) {
   Search("|D|D/|D|D/|D|D|D|D",ERRBREAK)
   #1 = Num_Eval_Date()
   Del_Char(Chars_Matched)
   Num_Ins_Date(#1+7)
}
```

NER Num_Eval_Reg( ) - See Atoi( )

NI Num_Ins(*n*)

Options: EXTRA, FILL, FORCE, LEFT, NOCR - See **Num_Type( )**

Usage: Num_Ins(#5) NI(#90,LEFT)

Description: **Num_Ins(*n*)** inserts numeric value '*n*' into the text as a decimal ASCII number followed by a "newline".

Num_Ins(*n*,*OPTIONS*) takes the same command options as **Num_Type( )** which control how the inserted ASCII number is formatted and padded.

Notes: **Num_Ins(*n*)** is equivalent to:
Out_Ins Num_Type(*n*) Out_Ins(CLEAR).

The command **#*x* = Num_Eval** sets numeric register '*x*' from the numeric expression (number) at the edit position. This is in many ways the opposite of the **Num_Ins( )** command.

See Also: Commands: Itoa( ), Num_Type( ), Num_Eval( )

Examples:

```
#10 = Get_Num("Enter number to insert: ")
Num_Ins(#10)
```

Prompt for a number and insert it into the text.

NID Num_Ins_Date( ) - See Num_Eval_Date( )

Num_Push(*x*,*y*)
Num_Pop(*x*,*y*)

Usage: Num_Push(10,19) Num_Push(50,50) Num_Pop(10,19)

Description: **Num_Push(***x*,*y***)** saves (pushes) the contents of numeric registers '*x*' through '*y*' onto the register stack; the registers are not changed. Registers are restored from the stack with the **Num_Pop(***x*,*y***)** command. Up to 254 registers can be saved on the first-in last-out stack.

Num_Pop and **Num_Push**, without arguments, return the number of numeric registers that are currently pushed on the stack.

Num_Pop(*x*,*y***)** restores (pops) the contents of numeric registers '*x*' through '*y*' from the numeric register stack. Registers are saved on the stack with the **Num_Push( )** command. If the register stack is empty, **Num_Pop(***x*,*y***)** has no effect and no error is given.

Notes: Unlike the **Reg_Push( )** command, pushing the numeric registers on the stack does not clear the registers.

The **Num_Push(***x*,*y***)** and **Num_Pop(***x*,*y***)** commands permit a command macro to use numeric registers for internal purposes and restore the original contents when the macro is done.

See Also: Commands: Reg_Pop( ), Reg_Push( ), Num_Push( )
Topics: "Numeric Register Stack" in Chapter 3

Examples:

```
Num_Push(15,15)
#15 = Get_Num("Enter Line Number: ")
Goto_Line(#15)
Num_Pop(15,15)
```

Prompt for the desired line number. Then move the edit position to that line in the file. Save the current contents of register 15.

NS Num_Str( ) - See Itoa( )

NT Num_Type(*n*)

Options: EXTRA, FILL, FORCE, HEX, LEFT, NOCR, NOMSG

Usage: Num_Type(#4) NT(#4,LEFT)

Description: **Num_Type(***n***)** displays (types) numeric value '*n*' in decimal followed by a "newline". It is right justified and padded with spaces.

Numbers in the range 0 - 65,535 are right justified in a field of 5 columns. Numbers 65,536 - 2,147,483,647 use a field of 10 columns. Negative numbers use fields one column larger (6 or 11).

Num_Type(*n***,FILL)** uses "0" for padding instead of spaces.

Num_Type(*n*,EXTRA) uses extra padding for positive numbers to have the same width (6 or 11) as negative numbers.

Num_Type(*n*,FORCE) uses a field width of 10 for all positive numbers and a width of 11 for negative numbers.

Num_Type(*n*,FORCE+EXTRA) uses a field width of 11 for all positive and negative numbers.

Num_Type(*n*,LEFT) displays the number left justified without any padding.

Num_Type(*n*,NOCR) suppresses the following "newline".

Num_Type(*n*,HEX) displays the number in hexadecimal using the format 0Xhh, 0Xhhhh, 0Xhh:hhhh or 0Xhhhh:hhhh, depending upon the number of significant digits. It is always left justified.

Num_Type(*n*,HEX+NOMSG) displays the number in hexadecimal in the format "hhhhhhhh" with as many 'hh' hex digits as needed, left justified; the "0X" and ":" are suppressed.

See Also: Commands: Num_Display( ), Num_Ins( )

Examples:

```
#2=10  #2=#2+12
Num_Type(#2)
```

Type out numeric register 2, displaying the value 22.

```
Out_Print  Num_Type(#9)  Out_Print(CLEAR)
```

The value of numeric register 9 is printed followed by a "newline".

O_Option
OS OS_Type

Usage: Internal Values

Description: **O_Option** returns the value of the number following the "**-O**" invocation option. For example, if VEDIT was invoked with "**vedit -o4**", **O_Option** returns 4.

OS_Type returns the operating system type. 1 = Windows, 2 = DOS, 4 = UNIX/XENIX, 5 = QNX, 6=Linux. This permits a macro to determine which OS dependent commands are available.

See Also: Commands: Is_Win32_Version, Is_WinNT

OF Out_File("*file*")

Options: CLEAR

Usage: Out_File("savefile") OF(CLEAR)

Description: **Out_File("*file*")** re-routes any Command Mode console output to the file '*file*'. **Out_File(CLEAR)** stops the re-routing, closes the file and resumes normal console output. The "COMMAND:" prompt also stops re-routing.

See Also: Commands: Block_Save_As( ), Ins_File( ), Out_Ins( )
Topics: "Re-routing Console Output" in Chapter 2

Examples: **Out_File("savedir") Dir( ) Out_File(CLEAR)**

Save a display of the currently directory in the file "savedir".

OI Out_Ins( )

Options: CLEAR

Usage: Out_Ins( ) OI(CLEAR)

Description: **Out_Ins( )** re-routes (inserts) any Command Mode console output into the edit buffer at the current edit position.

Out_Ins(CLEAR) stops the re-routing and resumes normal console output. The "COMMAND:" prompt also stops any re-routing.

Notes: The **Out_Ins( )**, **Out_OS( )**, **Out_File( )**, **Out_Print( )** and **Out_Reg( )** commands can be nested.

See Also: Commands: Out_OS( ), Out_File( ), Out_Print( ), Out_Reg( )
Topics: "Re-routing Console Output" in Chapter 2

Examples: **Out_Ins( ) Date(NOCR) Type_Space(2) Time(NOCR) Out_Ins(CLEAR)**

Insert the current date and time.

Out_Ins Dir("*.*",NOMSG) Out_Ins(CLEAR)

Insert the disk directory into the edit buffer, one filename per line.

Out_OS( ) (DOS Only)

Options: CLEAR

Usage: Out_OS Out_OS(CLEAR)

Description: **Out_OS( )** re-routes any Command Mode console output directly to DOS, bypassing VEDIT's screen and window handling.

Out_OS(CLEAR) stops the re-routing and resumes normal console output. The "COMMAND:" prompt also stops re-routing.

Notes: This command can be used to initialize a CRT terminal or send a control sequence to the ANSI.SYS driver. The characters to be sent would normally be stored in a text register.

This command is rarely needed and must be used with care to avoid disturbing the screen display.

See Also: Commands: Reg_Type( ), Char_Dump( ), Out_Ins( )
Topics: "Re-routing Console Output" in Chapter 2

Examples: **Out_OS Reg_Type(9,0)** Dump the contents of text register 9 directly to DOS.

OP Out_Print( )

Options: CLEAR

Usage: Out_Print( ) OP(CLEAR)

Description: **Out_Print( )** re-routes the following Command Mode console output to the printer. This command can be used in conjunction with the commands **Type( )**, **Directory( )**, **Type_File( )**, **Reg_Type( )**, **Num_Type( )**, etc., to print text.

Out_Print(CLEAR) stops the re-routing and resumes normal console output. The "COMMAND:" prompt also stops re-routing.

Notes: There is a subtle difference between **Out_Print( ) Type( )** and **Print( )**. The **Print( )** command, do not expand control characters so that special printer features can be accessed. However, **Out_Print( ) Type( )** prints text exactly as it is displayed on the screen according to the current display mode. For example, in display mode "1" where control characters are expanded, instead of sending a <Ctrl-H> to the printer, it would send the two characters "**^H**".

See Also: Commands: Out_Ins( ), Type( ), Reg_Type( ), Num_Type( )
Topics: "Re-routing Console Output" in Chapter 2

Examples: **Out_Print( ) Directory( )** Print the directory on the printer.

OP( ) Reg_Type(6) Print the contents of register 6 on the printer. (Similar to **Reg_Print(6)**.)

OR Out_Reg(*r*)

Options: CLEAR

Usage: Out_Reg(8) OR(CLEAR)

Description: **Out_Reg(***r***)** re-routes (appends) the following Command Mode console output to text register '*r*'. The re-routing continues until the command **Out_Reg(CLEAR)** or the next "COMMAND:" prompt.

See Also: Commands: Out_Ins( )
Topics: "Re-routing Console Output" in Chapter 2

Examples:
```
Out_Reg(11)
Reg_Type_Block(10,1,5)
Out_Reg(CLEAR)
```
Copy the first five characters of register 10 into register 11.

OM Overwrite_Mode(*n*)

Usage: Overwrite_Mode(2) Overwrite_Mode

Description: **Overwrite_Mode(*n*)** sets the overwrite-only mode in the current edit buffer to '*n*'. It is identical to **Config(F_OVER_MODE, *n*,LOCAL)**. The valid values are:

0 Overwrite-only mode is disabled. (However, it is always enabled for disk sector editing.)

1 Record mode. Overwrite-only mode is enabled if the "File type" is set to 8 or greater for fixed-length-record data/binary files.

2 Overwrite-only mode is enabled for all file types.

Overwrite_Mode, without parameters, only returns the value of the overwrite-only mode setting for the current edit buffer. This is identical to **Config(F_OVER_MODE)**.

Return: Returns the (changed) value of the overwrite-only mode for the current edit buffer.

See Also: Topics: "Overwrite-Only Mode" in Chapter 4 of the VEDIT User's Manual.

PK Previous_Key(*n*)

Usage: Internal Value

Description: **Previous_Key(*n*)** returns the '*n*'th most recent keyboard char/function-code. **Previous_Key(0)** returns the most recent keystroke. Simple keys have value 00 - 255. Function-codes have value > 255. The last 10 keystrokes are available.

Previous_Key(*n*,RAW) returns the '*n*'th most recent raw keystroke. Function keys return their hardware "Scan-code" multiplied by 256. The last 128 raw keystrokes are available.

Notes: This command is useful for testing if the same key was pressed two or more times in a row.

Appendix A lists the function-code values. These are the same values as returned by the **Get_Key( )** command.

See Also: Commands: Get_Key( )

PTS Previous_Tab_Stop

Usage: Internal Value

Description: **Previous_Tab_Stop** returns the column position of the previous tab stop based on the current value of **Cur_Col**. It returns 0 when the edit position is at the beginning of a line.

See Also: Commands: Config_Tab( ), Next_Tab_Stop( )

PR Print(*m*)
PB Print_Block(*p,q*)

Options: COLUMN, COLSET, EVENT, LINESET, NOEVENT, NORESTORE, RAW, RESET

Usage: Print(40) Print(-ALL) Print_Block(BB,BE)

Description: **Print(*m*)** prints '*m*' lines using the same range and syntax as the **Type( )** command. If there are fewer than '*m*' lines to print, as many lines as possible are printed and no error is given. Margins and printing parameters are determined by **{CONFIG, Printer}** menu or equivalent **Config(P_..)** commands.

Control and other characters are printed according to the current Print mode, set by **{CONFIG, Printer, Print mode}**. Typically, tabs are expanded to spaces and other control characters are sent as-is to the printer without expansion. The Print mode can also be set to print in hexadecimal or print an EBCDIC file on an ASCII printer.

Print(*m*,RAW) prints in "Raw" mode without margins and prints all control characters as-is, without conversion. It ignores all printing configuration parameters.

If a print-job is not yet open, the "Printer Start String" is sent, if enabled. **Print(*m*,NOEVENT)** suppresses the print-job start (init) string, even if enabled.

Print(*m*,EVENT) sends the print-job start string, even if disabled.

Print(*m*,NORESTORE) moves the edit position just past the last character printed. If '*m*' is negative, there is no apparent change.

Print_Block(*p,q*) prints the block of text between file positions '*p*' and '*q*'.

Print_Block( ) has the same options EVENT, NOEVENT, RAW and NORESTORE as **Print( )**; it has the same options COLUMN, COLSET and LINESET as the **Type_Block( )** command.

Notes: Press **[CANCEL]** (**<Ctrl-C>**) or **<Ctrl-Break>** to stop the printing.

Since **Print(*m*)** does not move the edit position, it is often followed by a **Line( )** command to advance the edit position to the next block of text to print.

See Also: Commands: Line( ), Type( ), Reg_Print( ), Out_Print( )
{PRINT} menu

Examples:

Begin_Of_File **Print(ALL)**	Moves the edit position to the beginning of the file and prints the entire file.

PE Print_Eject( )
PF Print_Finish( )

Options: EVENT, NOEVENT, SUPPRESS

Usage: Print_Eject(0) Print_Finish

Description: **Print_Eject( )** advances the printer to the start of a new page. This is often called a "Page Eject" or "Form Feed" and causes the following printed text to start on a new page. Depending upon the setting of **{PRINT, Config, Enable Form Feed}** (**Config(P_E_FF)**), it sends either a single "Form-Feed" or multiple "Line-Feed" characters to the printer.

Print_Eject(0) resets the internal line counter without sending anything to the printer. This re-synchronizes VEDIT with the printer following a manual paper advance. It may also be needed inside macros which change the **Config(P_..)** parameters.

Print_Eject(1,NOEVENT) suppresses the print-job start (init) string, even if enabled. **Print_Eject(1,EVENT)** sends the print-job start string, even if disabled. These options are only applicable if the print-job began with a **Print_Eject( )**, which is very rare.

Print_Finish( ) finishes the current print job. Assuming **{PRINT, Config, Page eject on finish}** is enabled, it first performs a **Print_Eject**. If **{CONFIG, Printer, Enable printer setup strings}** is set to "2" or "3", sends the "Printer Finish string". It then closes the print-job; when printing to a file, this also closes the file.

Print_Finish(SUPPRESS) suppresses the page eject, even if enabled. **Print_Finish(NOEVENT)** suppresses the print-job finish (reset) string, even if enabled. **Print_Finish(EVENT)** sends the print-job finish string, even if disabled.

Notes: On most systems the output to the printer is spooled and nothing will actually print until the print-job is closed.

See Also: Commands: Config( ), Print( )
{FILE, Print} menu

Examples:

```
Begin_Of_File( )
While (!At_EOF) {
   Print(40) Line(40)
   Page_Eject
}
Print_Finish( )
```

Prints the entire file, but with only 40 lines of text per page.

```
Print(-ALL)
Print(ALL)
Print_Finish( )
```

Prints the entire file using the current printing margins. The edit position is not changed. Equivalent to **{FILE, Print}** and selecting "All - entire file".

PCPL Printer_CPL
PLN Printer_Line
PLPP Printer_LPP

Usage: Internal Values

Description: **Printer_CPL** returns the printer's number of "Characters Per Line". This value depends upon the font and size selected with **{CONFIG, Printer font}**, the paper size in the printer and its portrait/landscape orientation. (Windows only)

Printer_LPP returns the printer's number of "Lines Per Page".

(Windows) If **Config(PW_PAPER_L)** is set to "0=Auto", this value depends upon the font and size selected with **{CONFIG, Printer font}**, the paper size in the printer and its portrait/landscape orientation. If **Config(PW_PAPER_L)** is set to any other value, **Printer_LPP** simply returns this value.

(DOS, QNX, UNIX, Linux) **Printer_LPP** simply returns the value of **Config(P_PAPER_L)**.

Printer_Line returns the line number on the printed page that the next **Print( )** command would print to, e.g. 1 - 66. It returns 0 (zero) if nothing has been printed on the current page, e.g. that the top margin has not yet been printed.

Notes: The expression **Printer_LPP - Printer_Line** can be used by a printing macro to determine how many lines are left on the page, e.g. to ensure that a 10 line table will be printed on one page.

Examples:

```
if ((Printer_LPP - Printer_Line) < 10) {
   Print_Eject() }
```

If less than 10 lines remain on the printed page, start a new page.

Process_ID (Windows Only)

Usage: Internal Values

Description: **Process_ID** returns the process ID number assigned by Windows to the current instance of VEDIT. It can be used to create a unique temporary filename that won't conflict with other copies of VEDIT that may be running (as other processes).

Notes: The predefined string variable "**PID**" is the process ID number converted into a 5-digit string.

QALL - See Exit( )

QALLY - See Exit( )

RINP Redirect_Input("*file*")

Options: CLEAR

Usage: Redirect_Input("wildfile.inp") Redirect_Input(CLEAR)

Description: **Redirect_Input("*file*")** uses '*file*' as the source of the "keyboard" input characters for the **Get_Input( )**, **Get_Key( )** and **Get_Num( )** commands and the "COMMAND:" prompt. The input redirection automatically ends when the end of the file is reached.

Redirect_Input(CLEAR) ends the input redirection, restoring normal input from the keyboard.

Notes: Input redirection can be used with the **`wildfile.vdm`** macro to fully automate the processing of many files.

It is currently not possible to redirect input to the **Dialog_Input_1( )** command from a file.

See Also: Commands: Is_Redirect_Input
Topics: "Input Commands" in Chapter 3

Examples: The following command starts the Windows version of VEDIT and uses the input redirection file **`wildfile.inp`** as the input to the WILDFILE macro:

```
vpw -c'rinp("wildfile.inp") callf(100,"wildfile")'
```

RCOMP Reg_Compare(*r*,"*text*")

Options: CASE

Usage: Reg_Compare(2,"VEDIT")

Description: **Reg_Compare(*r*,"*text*")** compares the contents of text register '*r*' to '*text*'. This is a character-by-character comparison without Pattern matching or Regular expressions. The comparison is not case sensitive unless the command option **Reg_Compare(*r*, "*text*",CASE)** is used.

Two text registers '*r1*' and '*r2*' can be compared by using the syntax **Reg_Compare(*r1*,@(*r2*))**. However, '*r*' cannot be an edit buffer.

Return: The results of the comparison are returned and saved in **Return_Value**:

0 The comparison is successful, '*r*' is "equal to" '*text*'.
1 '*r*' is lexically "greater than" '*text*'.
2 '*r*' is lexically "less than" '*text*' or '*r*' is empty.

Chars_Matched is set to the number of matching characters; is set to 0 (zero) if the very first character does not match.

If '*r*' is empty, **Error_Flag** is set. If '*text*' is Null, e.g. '*r2*' is also empty, the return value is "0", else the return value is "2".

See Also: Commands: Compare( ), Match( )

RC Reg_Copy(*r,m*)
RCB Reg_Copy_Block(*r,p,q*)

Options: APPEND, CLIPBOARD, COLSET, COLUMN, DELETE, FILL, INSERT, LINESET, NORESTORE, RESET

Usage: Reg_Copy(1,40) RC(2,-20,APPEND)
Reg_Copy_Block(9,#10,#11,COLUMN)

Description: **Reg_Copy(***r*,*m***)** copies all text lines from the edit position up to and including the '*m*'th "newline" to text register '*r*'. The previous contents of the register are overwritten, unless the command options "**APPEND**" or "**INSERT**" are used. The text in the edit buffer is unchanged.

Reg_Copy_Block(*r*,*p*,*q***)** copies the block of text starting with the '*p*'th character in the file up to, but not including, the '*q*'th character to text register '*r*'. The edit position is normally unchanged, however the command option "NORESTORE" sets the edit position past the end of the copied block.

Reg_Copy_Block(*r*,*p*,*q*,**DELETE)** moves the block to register '*r*' by deleting the original block after copying it. The option "**DELETE+FILL**" copies the block and then replaces (fills) the original block with spaces. The "fill" character is set with **{CONFIG, Tab/Fill, Block fill character}**. It defaults to spaces.

Reg_Copy_Block(*r*,*p*,*q*,**COLUMN)** copies a columnar block to the register. File positions '*p*' and '*q*' define the corners of the columnar block. With **Reg_Copy_Block(***r*,*p*,*q*, **COLSET,***c1*,*c2***)**, '*c1*' and '*c2*' specify the columns of the block and file positions '*p*' and '*q*' (regardless of their column position) specify the first and last lines of the columnar block. During the copy, Tab characters are expanded to spaces and short lines are padded with spaces — all lines in the register are of equal length.

Reg_Copy_Block(*r*,*l1*,*l2*,**LINESET)** copies a line-range block to the register. All characters on lines (or records) '*l1*' through '*l2*' (inclusive) are copied to the register; this includes the "newline" at the end of line '*l1*'.

Reg_Copy_Block(*r*,*p*,*q*,**CLIPBOARD)** copies the block to register '*r*' and then copies '*r*' to the Windows clipboard. Special internal register "120" is a good choice for '*r*'; it is automatically emptied when entering Visual Mode. The maximum block size is 64K. The 32-bit Windows version has the preferred **Clip_Copy_Block( )** command which supports huge blocks.

Notes: If VEDIT cannot make sufficient memory space available for the text copy, the text register is only emptied, nothing is copied to it and the error "BLOCK IS TOO LARGE FOR TEXT REGISTER" is given.

The error "CANNOT MODIFY EXECUTING MACRO" results if a macro attempts to change a text register which contains an executing command macro.

Use **Reg_Empty( )** to empty text registers that are no longer needed.

See Also: Commands: Block_Copy( ), Clip_Copy_Block( ), Out_Reg( ), Reg_Ins( ), Type( )
Topics: "Text Registers" in Chapter 3
"Columnar Blocks" in Chapter 4 of the VEDIT User's Manual

Examples: **Reg_Copy(1,120) Del_Line(120)**

Save 120 lines in text register 1 and then deletes them from the edit buffer.

Reg_Copy_Block(6,#10,#11,APPEND+COLUMN)

Append the columnar block between the file positions stored in numeric registers 10 and 11 to text register 6.

RE Reg_Empty(*r*)

Options: EXTRA

Usage: Reg_Empty(4) RE(100,EXTRA)

Description: **Reg_Empty(***r***)** empties text register '*r*'. '*r*' cannot be emptied if it is currently executing as a command macro unless the command option **Reg_Empty(***r***,EXTRA)** is used. **Reg_Empty(***r***, EXTRA)** must be used with care, but is useful for saving the memory space of a command macro which VEDIT thinks is still executing, but is no longer really needed.

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

See Also: Commands: Reg_Copy( ), Chain( )
Topics: "Command Macros" in Chapter 3

Examples: **RE(4)** Empty text register 4.

RF Reg_Free

Usage: Internal Value

Description: **Reg_Free** returns the ID number of the next free (empty) text register in the range 10 - 99.

Notes: Using **Reg_Free** is the ideal way for a macro, especially a "subroutine" macro, to determine which text registers can temporarily be used for its own purposes.

Examine **`wildfile.vdm`** to see how **Reg_Free** can be used.

RI Reg_Ins(*r*)

Options: BEGIN, CLIPBOARD, COLUMN, FORCE, LINEBLOCK, OVERWRITE, RAW

Usage: Reg_Ins(4) RI(0,COLUMN)

Description: **Reg_Ins(*r*)** inserts the contents of text register '*r*' at the edit position. If the register is empty, nothing is inserted. The contents of the register are not affected. The edit position is advanced past the inserted text.

Reg_Ins(*r*,BEGIN) leaves the edit position at the beginning of the inserted text.

Reg_Ins(*r*,OVERWRITE) overwrites the existing text at the edit position. If the buffer is in overwrite-only mode, the "**OVERWRITE**" option is automatically selected.

Reg_Ins(*r*,CLIPBOARD) copies the Windows clipboard to register '*r*' and then inserts '*r*' into the edit buffer. Reserved register "120" is a good choice for '*r*'; it is automatically emptied when entering Visual Mode. The maximum block size is 64K. The 32-bit Windows version has the preferred **Clip_Ins( )** command which supports huge blocks.

Reg_Ins(*r*,FORCE) pads the end of the line with spaces to reach the cursor column when inserting a columnar block with the cursor past the end-of-line. It is a special option used internally by **{BLOCK, Insert register}** and in BRIEF.KEY. It only works in keystroke macros run from visual mode.

The register is normally inserted as the same type of block, i.e. stream, columnar or line-range, as it was saved. However, the command options "**RAW**", "**COLUMN**" and "**LINEBLOCK**" force the register to be inserted as a stream, columnar or line-range block regardless of how it was saved.

When inserting a columnar block, each line in the register is inserted into subsequent lines in the text, each time beginning at the original edit position's column. If the right edge of register '*r*' contains jagged lines, no justification of the block is performed. If **{CONFIG, Emulation, Expand <Tab> with spaces}** (**Config(E_EXP_TAB)**) has Mask-2 reset (e.g. value "0"), any spaces in the inserted text and surrounding areas are converted to Tab characters.

When inserting a line-range block, the entire block is inserted at the beginning of the current line.

Notes: With the command form **Reg_Ins(*r*+BUFFER)**, '*r*' can be an edit buffer, in which case the contents of that edit buffer (but not necessarily the entire file) are inserted.

Reg_Ins( ) is similar to the **{BLOCK, Insert register}** function.

See Also: Commands: Clip_Ins( ), Reg_Copy( ), Reg_Size( )
Topics: "Text Registers" in Chapter 2, "Block Operations" in Chapter 3, "Columnar Blocks" in Chapter 4 of the VEDIT User's Manual

Examples:

Begin_Of_File( ) **Reg_Ins(9)**	Insert the contents of text register 9 at the beginning of the edit buffer.
Repeat(12) { Reg_Ins(2) }	Insert twelve (12) copies of text register 2 at the edit position.
Reg_Copy(3,132) **Del_Line(132)** **Begin_Of_File** **Line(10) Reg_Ins(3)**	Move 132 lines of text, by saving it in text register 3, deleting the original lines and inserting the text after the tenth line of the file.

RL Reg_Load(*r*,"*file*")
RLP Reg_Load_Part(*r*,"*file*",*offset,length*)

Options: APPEND, EXTRA, HOMEDIR, INSERT, MACRODIR, NOERR, USERCFGDIR, USERMACRODIR

Usage: RL(4,"part2.txt") Reg_Load(100,"macro1.vdm",EXTRA)

Description: **Reg_Load(***r***,"***file***")** loads the entire '*file*' into text register '*r*'. The file is not altered. The command option "**APPEND**" appends the file to any existing contents in register '*r*'. The command option "**INSERT**" inserts the file at the beginning of the register.

Reg_Load(*r***,"***file***",HOMEDIR)** searches the *VEDIT Home Directory* for the file if it is not found in the current directory.

Reg_Load(*r***,"***file***",MACRODIR)** searches the *VEDIT Macro Directory* for the file if it is not found in the current directory.

Reg_Load(*r***,"***file***",USERCFGDIR)** searches the *User Config Directory* for the file if it is not found in the current directory.

Reg_Load(*r***,"***file***",USERMACRODIR)** searches the *User Macro Directory* for the file if it is not found in the current directory.

Reg_Load(*r***,"***file***",EXTRA)** searches for the file first in the current directory, then in the *User Macro Directory*, then in the *VEDIT Macro Directory*, and last in the *VEDIT Home Directory*. "EXTRA" is equivalent to "USERMACRODIR+MACRODIR+HOMEDIR".

Reg_Load(*r***,"***file***",***offset*,*length*)** loads just a portion of '*file*' into '*r*'. The portion begins with the '*offset*' character in the file (counting starts with 0) and consists of '*length*' bytes.

Return: Always returns a value of 1. Sets **Error_Flag** if the file was not found.

Notes: **Reg_Load_Part( )** is used by the edit-session-restore feature to restore the text registers.

See Also: Commands: Reg_Save( ), Ins_File( ), Call_File( )

Examples:

Reg_Load(4,"macro.vdm")	Load the file **`macro.vdm`** into text register 4.

RLM Reg_Lock_Macro(*r*)

Options: CLEAR, EXTRA

Usage: Reg_Lock_Macro(10) RLM(0)

Description: **Reg_Lock_Macro(*r*)** sets up the macro in text register '*r*' to be executed in place of the normal "COMMAND:" prompt. It is primarily used to display a main menu of operations, as in the WILDFILE and COMPARE macros. '*r*' may be any register ***except*** "0".

Reg_Lock_Macro(CLEAR) or **Reg_Lock_Macro(0)** disable any "locked-in" macro, as does any invalid '*r*'.

VEDIT automatically turns off auto-execution when any syntax or logical error is encountered in a command macro; **Reg_Lock_Macro(*r*,EXTRA)** keeps the locked-in execution enabled. Use this with great care; it can lead to infinite loops that require re-booting of the computer.

Reg_Lock_Macro, without parameters, only returns the ID number of the currently locked-in macro, or 0 (zero) if there is none.

Return: Returns the ID number of the currently locked-in macro, or 0 (zero) if there is none.

Notes: Use this command with care! Since pressing **[CANCEL]** (**<Ctrl-C>**) normally returns to the "COMMAND:" prompt, it re-executes the register instead. Debug any macros thoroughly before and after adding the **Reg_Lock_Macro(*r*)** command. Be sure to allow a way for the user to exit VEDIT (it can be done with the **{FILE}** or **{ESCAPE}** menus).

See Also: Commands: Call( ), Reg_Empty( ), Chain( )
Topics: "Locked-in Macros" in Chapter 3

Examples:

Reg_Lock_Macro(101)	Set up to execute register 101 in place of the normal command prompt.
Reg_Lock_Macro(CLEAR)	Turn off the locked-in macro execution.

RMF Reg_Mem_Free
RSIZE Reg_Size(*r*)

Usage: Internal Value

Description: **Reg_Mem_Free** returns the number of bytes free (available) for text register usage.

Reg_Size(*r*) returns the number of bytes used by text register '*r*'. It is often used to test if a text register is empty.

Examples:
```
Get_Environment(9,"VEDPATH")
if (Reg_Size(9)==0) {
   Message("\nError - VEDPATH is not defined!")
}
```

Read the value of environment variable "VEDPATH" into register 9; display an error message if it is not defined.

Reg_Pop( ) - See Reg_Push( )

RP Reg_Print(*r*)

Options: EVENT, NOEVENT, RAW

Usage: Reg_Print(3) Reg_Print(3,0,RAW)

Description: **Reg_Print(*r*)** prints the contents of text register '*r*'. Margins and printing parameters are determined by **{CONFIG, Printer}** menu or equivalent **Config(P_..)** commands.

Control and other characters are printed according to the current Print mode, set by **{CONFIG, Printer, Print mode}**. Typically, tabs are expanded to spaces and other control characters are sent as-is to the printer without expansion.

Reg_Print(*r*,*n*) prints the contents of text register '*r*' with a print mode of '*n*', which can printed expanded control characters, print in hexadecimal or print EBCDIC characters. See **Reg_Type( )** for the values of '*n*'. (Masks 1024, 128, 64 and 04 do not apply to **Reg_Print( )**.)

Reg_Print(*r*,0,RAW) prints in "Raw" mode without margins and prints all control characters as-is, without conversion. It ignores all printing configuration parameters.

Reg_Print(*r*,*n*,NOEVENT) suppresses the print-job start (init) string, even if enabled. Usually used in conjunction with the "**RAW**" option to select printer fonts and other features.

Reg_Print(*r*,*n*,EVENT) sends the print-job start (init) string, even if disabled.

See Also: Commands: Print( ), Out_Print( ), Reg_Type( )
Topics: "Printing Text" in Chapter 2

Examples:

Reg_Print(5) Print the contents of register 5.

Reg_Print(9,0) Print the contents of register 9. No characters are expanded - all characters are sent as-is.

Reg_Print(9,256+16+8+2)

All control characters, except CR and LF are expanded to "^x" format. Esc is expanded to "<ESC>". Tabs are expanded with spaces.

Reg_Prot(*r*,*s*,*n*)

Usage: Reg_Prot(9,9) Reg_Prot(10,20,2)

Description: **Reg_Prot(***r*,*s*,*n***)** sets the write-protection of text registers '*r*' through '*s*' to '*n*'. The protection levels are:

0 Write protection is off.

1 Visual Mode and keystroke macros cannot alter registers. However, registers can be altered from command macros. Technically, registers are write-protected when **Visual_Macro( )** has Mask-8 set, e.g. the macro originated from Visual Mode.

2 Registers cannot be altered, loaded or emptied.

Reg_Prot(*r***)**, without options, only returns the protection level of register '*r*'.

Return: Returns the protection level of text register '*r*'.

Notes: **Reg_Prot(***r*,*s*,*n***)** permits macros to protect their registers from inadvertent alteration while the user is editing. Any macros which depend upon the contents of any text register must use **Reg_Prot( )** if they are to be "bullet-proof".

Examples: **Reg_Prot(10,20,1)** Protect registers 10 thru 20 from alteration from the Visual Mode.

Reg_Push(*r*,*s*)
Reg_Pop(*r*,*s*)

Options: SET

Usage: Reg_Push(1,9) Reg_Push(100,100)
Reg_Pop(1,9) Reg_Pop(100,100)

Description: **Reg_Push(***r*,*s***)** saves (pushes) the contents of text registers '*r*' through '*s*' onto the register stack and then empties (clears) the register contents. Registers are restored from the stack with the **Reg_Pop(***r*,*s***)** command. Up to 128 registers can be saved on the first-in last-out stack.

Reg_Push(*r*,*s***,SET)** does not empty (clear) the registers that are pushed on the stack.

Reg_Pop and **Reg_Push**, without arguments, return the number of text registers that are currently pushed on the stack due to **Reg_Push( )**, **Key_Cfg_Push( )** and **Win_Cfg_Push( )**.

Reg_Pop(*r*,*s***)** restores (pops) the contents of text registers '*r*' through '*s*' from the register stack that were saved with the **Reg_Push(***r*,*s***)** command. If the register stack is empty, **Reg_Pop(***r*,*s***)** has no effect and no error is given.

Notes: The **Reg_Push( )** and **Reg_Pop( )** commands permit a command macro to use text registers for internal purposes and restore the original contents when the macro is done.

With the "**SET**" option, you can also push registers 123 and 124, which are the **{USER}** and **{TOOLS}** menus.

It is up to the user to push and pop the registers in the correct order. The stack operates "first-in last-out". Therefore, multiple **Reg_Pop(***r*,*s***)** commands should appear in the reverse order from the original **Reg_Push(***r*,*s***)** commands. For example, **Reg_Push(1,4)**, **Reg_Push(7,9) Reg_Pop(7,9)** and **Reg_Pop(1,4)** correctly save and restore the registers 1 through 4 and 7 through 9.

In detail, **Reg_Push(1,4)** pushes (saves) register 1 first, then 2, then 3 and 4 last. **Reg_Pop(1,4)** pops (restores) register 4 first, then 3, then 2 and 1 last.

See Also: Commands: Num_Push( ), Num_Pop( )
Topics: "Text Register Stack" in Chapter 3
The on-line help has an example using the **{USER}** and **{TOOLS}** menus.

Examples:

Reg_Push(1,1) **Reg_Load(1,"oldfile.txt")** **Reg_Save(1,"newfile.txt")** **Reg_Pop(1,1)**	T-Reg 1 is used to copy the file "oldfile.txt" to "newfile.txt"; however, the original contents of register 1 are not changed.

RSAV Reg_Save(*r*,"*file*")

Options: OK

Usage: Reg_Save(4,"macro1.vdm") RSAV(2,"part2.txt",OK)

Description: **Reg_Save(***r*,"*file*"**)** saves the contents of text register '*r*' in the file '*file*'. The register contents are not affected. If an existing '*file*' already exists, the user is prompted for confirmation to overwrite it; **Reg_Save(***r*,"*file*"**,OK)** suppresses the confirmation prompt.

Notes: 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.

See Also: Commands: Reg_Load( )

Examples:

Reg_Save(4,"macro.vdm")	Save the contents of text register 4 in the file **`macro.vdm`**.

RS Reg_Set(*r*,"*text*")

Options: APPEND, INSERT

Usage: Reg_Set(3,"Begin_Of_File Print(ALL)")

Description: **Reg_Set(***r*,"*text*"**)** places '*text*' into text register '*r*'. **Reg_Set(***r*, "*text*"**,APPEND)** appends the text to any existing contents in the

register. **Reg_Set(***r***, "***text***", INSERT)** inserts the text at the beginning of any existing contents. **Reg_Set(***s***,@***r***)** copies text register '*r*' to text register '*s*'.

Notes: '*text*' may contain the **<Enter>** key, which inserts the "newline" character(s).

If insufficient memory space exists, the error "NOT ENOUGH MEMORY FOR OPERATION" is given and only part of the '*text*' will be inserted.

See Also: Topics: "Loading Macros into Text Registers" in Chapter 3

Examples: **Reg_Set(3,"Begin_Of_File Print(ALL)")**

The command sequence "Begin_Of_File Print(ALL)" is placed into text register 3.

RSIZE Reg_Size( ) - See Reg_Mem_Free

RSTAT Reg_Status

Usage: Reg_Status

Description: **Reg_Status** displays the number of characters held in each text register. **Reg_Status** is commonly used to see which registers are being used and how many characters they hold.

See Also: Commands: Mem_Status, Reg_Size, Buf_Stat

Examples: **Reg_Status** Display the sizes of the text registers.

RT Reg_Type(*r*,*n*)
RTB Reg_Type_Block(*r*,*p*,*q*)

Usage: Reg_Type(3) RT(3,2048) Reg_Type_Block(5,10,20)

Description: **Reg_Type(*r*)** displays the contents of text register '*r*'. It is commonly used to view the contents of a register. Control and graphics characters are expanded according to the current display mode.

The special command form **Reg_Type(*r*,0)** displays (dumps) the contents of register '*r*' without any expansion of control or graphics characters.

Reg_Type(*r*,*n*) expands control, tab, graphics and other characters, and lets you control exactly how they are expanded. The expansion is controlled by adding together the desired "mask" values:

Mask 32768	Display all characters in EBCDIC or via the current translation table
Mask 16384	Display after translating between ANSI/OEM.
Mask 8192	Display all characters in bit-wise mode.
Mask 4096	Display all characters in octal mode.
Mask 2048	Display all characters in hexadecimal mode.
Mask 1024	Display <CR>/<LF> as literal characters.
Mask 512	Display graphics characters in the format "nnn".
Mask 256	Display control characters in the format ^x.
Mask 128	Truncate long lines which extend past the right edge of the window.
Mask 64	Clear window when a <Ctrl-L> is encountered.
Mask 32	Display <CR> and <LF> as "<CR>" and "<LF>".
Mask 16	Display <Ctrl-H> (Backspace) as ^H. (Depends upon Mask 256.)
Mask 08	Display <Esc> as "<Esc>" instead of a normal control character.
Mask 04	Pause when a <Ctrl-S> is encountered.
Mask 02	Expand Tabs with spaces.
Mask 01	Use tab stops at every 8.

Reg_Type_Block(*r*,*p*,*q*) displays the block of characters between positions '*p*' and '*q*' in text register '*r*'. This permits performing substring operations with text registers, using text registers as fixed length string arrays and much more. **Reg_Type_Block(*r*,*p*,*q*)** can be used in conjunction with **Out_Ins( )** to insert a substring of a text register.

Notes: Press **[CANCEL]** (**<Ctrl-C>**) to stop the **Reg_Type( )** command.

See Also: Commands: Out_Print( ), Out_File( ), Out_Reg( ), Reg_Print( )
Topics: "Screen Display Modes" in Chapter 4 of the User's Manual

Examples: **Out_Print Reg_Type(5,0) Out_Print(CLEAR)**

The contents of text register 5 are dumped to the printer.

Registry_Delete_Item(...)
Registry_Delete_Key(...)
Registry_Get_Item(...)
Registry_Get_Number(...)
Registry_Delete_Key(...)
Registry_Set_Item(...)

Usage: Registry_Get_Item(106,"HKEY_CLASSES_ROOT\http\shell\open\command\")

Description: These very technical commands allow the VEDIT macro language to access and change the Windows registry.

The commands are fully described in the Windows version on-line help. For examples, refer to the **`regvedit.vdm`** macro.

Notes: Only users that understand the internal structure of the Windows registry should use these commands.

USE THESE COMMANDS WITH EXTREME CAUTION!!!

Improper use of these commands or "experimenting" could cause your computer to become unbootable and require a reinstallation of Windows. Some or all data on your hard disk could become lost!

Remainder

Usage: Internal Value

Description: **Remainder** returns the absolute value of the remainder from the last division. For example, following "20/7", **Remainder** returns the value 6.

Repeat_Count
Repeat_Flag

Usage: Internal Values (Very technical and rarely used)

Description: **Repeat_Count** returns the pending **[REPEAT]** count, or 1 if there is no repeat count.

Repeat_Flag returns 1 (TRUE) if **[REPEAT]** is pending. Otherwise, it returns 0.

R Replace( ) - See Search( )

RB Replace_Block( ) - See Search( )

RPOS Restore_Pos( )
SPOS Save_Pos( )

Options: RESET

Usage: Save_Pos Restore_Pos(RESET)

Description: **Save_Pos( )** saves the current edit position on a special stack of "text markers". You can save up to 5 positions on the stack. **Restore_Pos( )** restores the edit position from the most recent position saved with **Save_Pos( )**. If the stack is empty, the command has no effect. The command option **Restore_Pos(RESET)** empties (resets) the edit position stack.

RTAB Retab_Block(*p,q*) - See Detab_Block( )

Return(*n*)

Usage: if (At_EOF) { Return }

Description: **Return(*n*)** stops the currently executing macro. Execution returns to the parent, or calling macro if there is one, otherwise it returns to the "COMMAND:" prompt or any "locked-in" macro. Inside a keystroke macro, it returns to Visual Mode. **Return_Value** is set to '*n*' for subsequent testing.

Return: This command does not return to the current macro. **Return_Value** is set to '*n*'.

See Also: Commands: Goto_Pos( ), Break, Continue, Goto label
Topics: "Flow Control - Break-out Commands" in Chapter 3
"Commenting Macros"

Examples: See the topic "Flow Control" in Chapter 3 for examples.

RV Return_Value

Usage: Internal Value

Description: **Return_Value** returns the value of the last **Return(*n*)** executed due to a **Call( )**. Selected commands also save their normal return value in this internal value.

REVV Reverse_Video(*n*)

Usage: Internal Value

Description: **Reverse_Video(*n*)** returns the reverse video of screen attribute '*n*'.

SPOS Save_Pos( ) - See Restore_Pos( )

SCOL Screen_Cols
SL Screen_Lines

Usage: Internal Values

Description: **Screen_Cols** returns the total number of columns currently displayed on the "screen". For Windows, this is the number of text columns in a full-sized window; it depends upon the size of the VEDIT application window and the display font. For DOS, it is typically 80.

Screen_Line returns the total number of "screen" lines. For Windows, this is the number of text lines in a full-sized window; it depends upon the size of the VEDIT application window and the display font. In the DOS version, this is the number of screen lines displayed in the current hardware mode; it is typically 25.

Notes: Use **Screen_Size( )** to change the number of screen lines and/or columns.

If the DOS screen size is changed with hardware access commands, **Screen_Reset( )** is needed before VEDIT recognizes the new size.

SIBM Screen_IBM
SIBMM Screen_IBM_Mode

Usage: Internal Values

Description: **Screen_IBM** returns the IBM PC screen display type:
0 = Not IBM PC, 1 = Monochrome (Hercules)
2 = CGA, 4 = EGA, 5 = VGA

(DOS) It is often used to test whether **Screen_Size( )** can put an VGA/EGA into 50/43 lines mode.

Screen_IBM_Mode returns the IBM PC's hardware video mode. Typically: 3 = color, 7 = monochrome. It can also have other values for special VGA modes.

SI Screen_Init( ) - See Win_Delete( )

SL Screen_Lines( ) - See Screen_Cols( )

Screen_Mono( )

Usage: Screen_Mono( )

Description: **Screen_Mono( )** forces VEDIT to use the monochrome set of screen attributes. VEDIT always uses monochrome attributes with a monochrome adapter. To fully take effect, **Screen_Mono( )** should be followed with **Screen_Init( )**.

Notes: **Screen_Mono( )** is equivalent to setting **{CONFIG, Colors, Enable monochrome}**.

SR Screen_Reset( )

Usage: Screen_Reset( )

Description: **Screen_Reset( )** resets VEDIT to the current screen size and mode. On an IBM PC, it queries the screen hardware and BIOS to determine the current screen size. It then rewrites the entire screen, resizing windows as needed.

Screen_Reset( ) should be used after any DOS hardware access commands that change the video mode or screen size.

SS Screen_Size(*l,c*)

Options: TOGGLE

Usage: Screen_Size(50) Screen_Size(30,80) SS(TOGGLE)

Description: (Windows) **Screen_Size(***l*,*c***)** changes the VEDIT program window size so that a full-size editing window with scroll bars has '*l*' lines and '*c*' columns.

(DOS) **Screen_Size(***n***)** attempts to change the screen size to '*n*' lines. On an IBM PC, it will switch to a VGA mode with at least '*n*' lines. The command **Screen_Size(TOGGLE)** toggles a VGA between 25, 28 and 50 line modes. It is equivalent to **{VIEW, VGA/EGA toggle}**.

Return: Returns the (new) current number of screen lines.

Notes: The internal values **Screen_IBM** and **Screen_IBM_Mode** let a macro determine the type and video mode of the IBM PC display adapter on which it is running.

See Also: Commands: Screen_Cols, Screen_Lines, Screen_Type, Screen_IBM, Screen_IBM_Mode

Examples:

Screen_Size(50)	(DOS) Switch to VGA 50 line mode.
Screen_Size(30,80)	(Windows) Resize the VEDIT program for a 30x80 full-size editing windows.

ST Screen_Type

Usage: Internal Value

Description: **Screen_Type** returns the screen display type:

0 = Non-IBM PC (CRT terminal) Monochrome
1 = Non-IBM PC (CRT terminal) Color
2 = IBM PC Monochrome
3 = IBM PC Color (CGA, EGA, VGA)

Screen_Type can be used by a macro to determine which screen attributes (colors) are usable.

S Search("*ss*")
SB Search_Block("*ss*",*p,q*)
R Replace("*ss*","*rs*")
RB Replace_Block("*ss*","*rs*",*p,q*)

Options: Listed below (Not to be confused with **Search_Options**)

Usage: Search("mispell") Search("smith",REVERSE)
Search_Block("|Sword|S",BB,BE,BEGIN+COLUMN)
Replace("mispell","spell")
RB("the","these",BB,BE,COLSET+WORD+REVERSE,1,20)

Description: **Search("***ss***")** searches, starting from the edit position, for the text '*ss*'. The edit position is placed at the first character of the matched text. If '*ss*'is not found, the error "CANNOT FIND *string*" is given (unless suppressed) and the edit position remains unchanged.

Search("") searches for the '*ss*' previously specified with the "**SET**" option or with the **[SEARCH]** function.

Search_Block("*ss***",***p,q***)** also searches starting from the edit position, but the search is only successful if the matching text is entirely between file positions '*p*' and '*q*'. To search the entire block, be sure to first set the edit position at the beginning of the block, perhaps with **Goto_Pos('***p***')**.

Replace("*ss***","***rs***")** searches for the next occurrence of '*ss*' and changes it to '*rs*'. The edit position is placed after '*rs*' if '*ss*' is found, or else is left at its previous position if not found.

Search/Replace Command Options:

COLUMN	**Search_Block( )** and **Replace_Block( )** only match text that is entirely within a columnar block. '*p*' and '*q*' specify the corners of the block.
COLSET, *c1***,** *c2*	**Search_Block( )** and **Replace_Block( )** only match text that is entirely within a columnar block. '*p*' and '*q*' specify the lines of the block, and '*c1*' and '*c2*' specify the columns of the block.
LINESET	**Search_Block( )** and **Replace_Block( )** only match text that is entirely within a line-range block. '*l1*' and '*l2*' specify the first and last lines.
LOCAL	The Search/Replace is limited to the portion of the file currently in memory.
CASE	The Search/Replace is case sensitive; otherwise it is not case sensitive.
WORD	The Search/Replace is restricted to distinct words.
COUNT, *n*	**Search( )** searches for the '*n*'th occurrence of '*ss*'. **Replace( )** replaces the next '*n*'th occurrences of '*ss*'.

SIMPLE	Search/Replace for literal (simple) strings without using pattern matching or regular expressions.
REGEXP	Search/Replace using regular expressions, using "minimal" matching.
REGEXP+MAX	Search/Replace using regular expressions, using maximized matching.
EBCDIC	The search/replace strings are internally translated from ASCII to EBCDIC. Permits searching for text in an EBCDIC file.
HEX	The search/replace strings consist of hex values "00" thru "ff" separated by spaces. Hex words, double-words and quad-words are also supported.
REVERSE	Search/Replace is in reverse direction, from the edit position toward the beginning of the file.
ADVANCE	**Search( )** only: advances the edit position past the matched text.
SET	Sets the search string for **{SEARCH, Next}** or **Search("")**.
BEGIN	Starts the Search/Replace from the beginning of the file. Combined with "LOCAL", starts from the beginning of the portion of the file currently in memory.
ALL	**Search( )** searches for the last occurrence of '*ss*'. **Replace( )** replaces all occurrences of '*ss*'.
NORESTORE	Does not restore the edit position following an unsuccessful search, but leaves it at the end/beginning of the file. This can save time in large files.
CONFIRM	**Search( )** sets temporary block markers to the matched text so that it highlights in Visual Mode, same as **[SEARCH]**. **Replace( )** prompts for confirmation before replacing text, same as **[REPLACE]**
NOERR	Suppresses the error message if the search is unsuccessful. Execution continues with the next command.
ERRBREAK	Suppresses the error message if the search is unsuccessful, but performs a **Break** out of any command loop.

Return: Returns the number of occurrences found (or replaced), (0 if none). **Chars_Matched** is set to the number of characters matched by '*ss*'. **Error_Flag** and **Error_Match** are set if the search is not successful.

The returned value is also saved in **Return_Value**.

Notes: '*ss*' and '*rs*' may be up to 260 characters long.

See Also: Commands: Compare( ), Match( ), Match_Paren( ), SR_Set( ), Chars_Matched, Error_Flag, Error_Match, Return_Value
Topics: "Search and Replace" in Chapter 2
"Pattern Matching and Regular Expressions" in Chapter 4 of the VEDIT User's Manual

Examples:

```
Repeat (ALL) {
   Search("first",COUNT,3)
   Ins_Text("third",OVERWRITE)
}
```

Change every third occurrence of the word "first" to "third".

```
Goto_Pos(BB)
Replace_Block("fix up","improve",BB,BE,ALL)
```

Find all occurrences of the string "fix up" within the block of text defined by the block-begin and block-end markers, and replace them with the string "improve".

```
Search("|<First",REVERSE+NOERR+LOCAL)
```

Searches backwards for "First" occurring at the beginning of a line, and limits the search to the text currently in memory. No error message is displayed if not found.

```
Search("[A-Z][a-z]*",REGEXP)
```

Searches for the next capitalized word. It uses regular expressions.

Search_Options

Usage: Search_Options Search_Options(CASE)

Description: **Search_Options** returns the search options selected in the last Search/Replace dialog-box. This includes the search mode values SIMPLE, REGEXP, MAX, HEX, EBCDIC, and the option values CASE, WORD, BEGIN, LOCAL, EXTRA. The value "EXTRA" corresponds to the "[ ] Block" option.

Search_Options(*n*) sets the saved search options to '*n*'. This determines the search mode (e.g. SIMPLE or REGEXP) and options (e.g. WORD or BLOCK) for the next **{SEARCH, Next}** or **{SEARCH, Previous}** function.

See also: On-line help for **{SEARCH, Search}**.

Search_Status

Usage: Internal Value

Description: **Search_Status** returns the **[SEARCH]/[REPLACE]** status:

0 Search/replace is cancelled
1 **[SEARCH]** is set up
2 **[REPLACE]** is set up

Search_Status permits a macro to predict what operation **{SEARCH, Next}** will perform.

Set_Altered_Flag

Usage: Set_Altered_Flag(1) Set_Altered_Flag(0)

Description: **Set_Altered_Flag(1)** sets an internal flag so that the buffer appears to have been altered, even if it has not. The value of this internal flag is returned by **Is_Altered**.

Set_Altered_Flag(0) clears an internal flag so that the buffer appears not to have been altered, even if it has.

Notes: This command has only very limited and specialized use. Note that clearing this flag may cause an altered file not to be saved.

SM Set_Marker(*m*,*n*)

Usage: Set_Marker(4,123) Set_Marker(0,Cur_Pos)

Description: **Set_Marker(***m*,*n***)** sets text marker '*m*' to file position '*n*'. **Set_Marker(***m***,CLEAR)** clears text marker '*m*'. There are ten text markers numbered "0" through "9".

Notes: **Set_Marker( )** is equivalent to **{GOTO, Set text marker}**.

The text markers can be accessed with the command **Marker(***m***)**. Setting a text marker to "CLEAR", "-1" or any negative number clears the marker.

See Also: Commands: Block_Begin( ), Block_End( ), Marker( )
Topics: "Setting Block and Text Markers" in Chapter 3

Examples:

Set_Marker(4,123)	Set text marker 4 past the 123rd character in the file.
Set_Marker(0,Cur_Pos)	Set text marker 0 at the current character.

SVL Set_Visual_Line(*n*)

Usage: Set_Visual_Line(20)

Description: **Set_Visual_Line(***n***)** rewrites the Visual Mode window with the current line displayed on window line '*n*'.

Set_Visual_Line(0) centers the current line in the Visual Mode window.

Notes: Invalid values for '*n*' center the line in the window.

Set_Visual_Line( ) is used by keystroke macros that scroll the current cursor line to a different part of the screen.

See Also: Commands: Win_Scroll_Margin( )

Sleep(*n*)

Usage: Sleep(25)

Description: **Sleep(*n*)** delays for '*n*'/10 seconds (i.e '*n*' is in units of 1/10 second). The maximum value for '*n*' is 255 (25.5 seconds). VEDIT will appear completely dead during this delay period, but will respond to **[CANCEL]** (**<Ctrl-C>**).

Notes: The delay has an accuracy of about 5% on an IBM PC under Windows/DOS.

Examples:

Sleep(25)	Delay for 2.5 seconds.
Repeat(60) { Sleep(10) }	Delay for 60 seconds.

Sort(*p,q*)

Usage: Sort(0,File_Size) Sort(BB,BE,KEYCOLS,CB,CE)

Description: **NOTE:** We suggest using the new **Sort_Merge( )** command instead. **Sort( )** is internally implemented as a special case of the **Sort_Merge( )** command.

Sort(*p,q***)** sorts the entire lines (records) specified by file positions '*p*' and '*q*'. The sort is in ascending order using the entire line as the "key". Position '*p*' can be anywhere on the first line to be included in the sort. However, if '*q*' is at the beginning of a line, that line is not included.

Sort(*p,q***,KEYCOLS,***c1,c2***)** sorts the entire lines (records) specified by file positions '*p*' and '*q*'. The sort is in ascending order using columns '*c1*' through '*c2*' (inclusive) as the "key". Position '*p*' can be anywhere on the first line. '*q*' can be anywhere on the last line. The option "KEYCOLS" must be specified.

Lines that are shorter than '*c2*' columns are sorted as if they had trailing "nulls"; they are considered "less than" lines of which they are a subset.

Lines that are shorter than '*c1*' columns are sorted to the beginning of the block in the order they are encountered.

See Also: Topics: "Sorting Lines" in Chapter 4 of the VEDIT User's manual

Examples:

Sort(0,File_Size)	Sort all lines in the current file, using the entire line as the "key".

Sort_Load("*file*")

Options: NOERR

Usage: Sort_Load("collebc.tbl")

Description: Sort_Load("***file***") loads an alternate collate table used by the **Sort_Merge( )** command. If no table is explicitly loaded, **Sort_Merge( )** automatically loads the default file **`colldef.tbl`**.

Sort_Load("***file*",NOERR)** suppresses the error message if the file is not found.

Notes: An alternate collate table can be loaded each time VEDIT is started by adding this command to the **`ustartup.vdm`** file.

See Also: Commands: Sort_Merge( )
Topics: "Sorting Lines in a File / Block" in Chapter 4 of the User's Manual

SMX Sort_Merge(*p,q*)

Usage: Sort_Merge("10:16,25:35",0,File_Size)

Description: **Sort_Merge("***keyfield-list***",*p,q***)** sorts the block of lines (records) specified by file positions '*p*' and '*q*'. The sort is, by default, in ascending order and is case insensitive. It uses the last loaded collate table; if none has been loaded, it uses **`colldef.tbl`**.

'*keyfield-list*' specifies the columns to be used as the primary, and up to nine secondary, sort key fields. Any VEDIT numeric expressions can be used to specify the beginning and ending column numbers. Any desired delimiters, such as commas, colons and semicolons, can be used between the column numbers. Examples are:

"1:10"
"0,15; 1,5; 30,45"
"Column_Begin, Column_End"
"Cur_Col-5, Cur_Col+5"

NOTE: Do not use "-" to indicate a range of columns; it is used as a "minus" and not as a "dash". However, with the **SUPPRESS** option, "-" can be used to indicate a range of columns.

Position '*p*' can be anywhere on the first line to be included in the sort. However, if '*q*' is at the beginning of a line, that line is not included.

Sort_Merge(...,REVERSE) sorts in descending order.

Sort_Merge(...,CASE) distinguishes between upper and lower case. I.e. all upper case letters will occur before the lower case letters.

Sort_Merge(...,NOCOLLATE) sorts based on absolute (hex) value of each character without using a collate table. Therefore, it distinguishes between upper and lower case, and between tab characters and spaces.

Sort_Merge(...,RESET) clears the **Block_Begin** and **Block_End** markers (if they are set).

Sort_Merge(...,NORESTORE) does not restore the edit position after the sort; it will remain at the end of the sorted file or block.

Sort_Merge(...,SUPPRESS) does not perform any expression evaluation of the column numbers specified in '*keyfield-list*'. This also allows "-" to be used as a column range indicator.

Notes: As described under "Technical Description of Sorting Algorithm" in Chapter 4 of the User's Manual, this command uses 22 "extra" buffers (104-125) and creates numerous temporary files in the VEDIT Temp Directory during its operation, especially when sorting huge files.

See also: Commands: Sort_Load( ) Topics: "Sorting Lines in a File / Block" in Chapter 4 of the User's Manual

Examples: **Sort_Merge("1,100",0,File_Size)**

Sort the entire file in ascending order, equating upper and lower case letters. The first 100 columns of each line are used as the primary "sort key".

Sort_Merge("10,15;1,5;30,45",0,File_Size)

Sort the entire file in ascending order. The field in columns 10 thru 15 is used as primary "sort key"; the field in columns 1 thru 5 is the secondary sort key, and the field in column 30 thru 45 is the third sort key.

Sort_Merge(@(15),#20,#21)

Text register 15 contains the keyfield-list of column numbers. Numeric register 20 contains the position of the first line to be sorted; numeric register 21 contains the position of the last line to be sorted.

Sound(*n*,*k*) - See Alert( )

SRD SR_Display( )
SRS SR_Set("*ss*","*rs*")

Options: EXTRA, RESET, SET

Usage: SR_Display( ) SR_Set("tipo","typo",SET)

Description: **SR_Display( )** displays the current search and replace strings on separate lines following "SS=" and "RS=".

SR_Display(EXTRA) displays them on one line separated by a comma and without the headers.

SR_Set("*ss***","***rs***")** sets the search and replace strings to '*ss*' and '*rs*' without performing a replacement operation. It also sets **{SEARCH, Next}** into search mode.

SR_Set("*ss***","***rs***",SET)** sets **{SEARCH, Next}** into replace mode.

SR_Set("","",RESET) resets (clears) the current search and replace strings and cancels the **{SEARCH, Next}** function.

Notes: The primary purpose for these commands is to implement VEDIT's edit-session-restore feature.

See Also: Commands: Search( ), Replace( ), Search_Status

Statline_Message( ) - See Message( )

Strip_High(*m*)

Usage: Strip_High(12) Strip_High(ALL) Strip_High(#10,#11)

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

Strip_High(*p,q*) strips the block of characters between file positions '*p*' and '*q*'.

Strip_High(*p,q*,NORESTORE) sets the edit position past the end of the stripped block.

The command options "**COLUMN**", "**COLSET**" and "**LINESET**" specify a columnar or line-range block for stripping.

Notes: Be careful not to use this command on IBM PC "graphics" characters which also have their high bit set. The effects of this command cannot be undone with "Undo".

See Also: Topics: "WordStar Files" in Chapter 3

Examples:

Begin_Of_File( ) **Strip_High(ALL)**	Strip the high bit from all characters in the file.

SXL Syntax_Load("*file*")

Usage: Syntax_Load("clipper.syn")

Description: **Syntax_Load("*file*")** loads the syntax highlighting definition file '*file*'. These files typically have a ".syn" filename extension. Multiple .SYN files can be loaded into VEDIT at one time.

Notes: This command is generally not needed because VEDIT automatically loads .SYN syntax highlighting files as needed.

Color syntax highlighting is usually set up by the File-open configuration feature.

The command **Config(PG_E_SYNTAX,1)** enables syntax highlighting. It is equivalent to enabling **{CONFIG, Programming, Enable color syntax highlighting}**. The string variable **SYN_NAME** determines which .SYN file the current file/buffer will use; each buffer has its own **SYN_NAME** variable. It can be set with e.g. **Config(SYN_NAME,"clipper.syn").**

VEDIT must be able to access the supplied macro **`loadsyn.vdm`** which is automatically executed to process the .SYN file.

See Also: Commands: Config_String(SYN_NAME), Template_Load( )
Topics: "Color Syntax Highlighting" and "File-open Configuration" in Chapter 5 of the VEDIT User's Manual

Sys System("*command*") / System

Options: LOCAL, NOMSG, OK

Usage: System( ) System("dir") System("vspell |@(98)")

Description: **System( )** enters (shells to) the operating system without leaving VEDIT. Any desired OS commands and programs can be executed. Give the OS command "**exit**" to return to VEDIT.

System("*command***")** executes the following OS (UNIX) command and returns to VEDIT. **System("***command***")** can run a DOS command such as "dir" (UNIX: "ls") or another program such as a compiler, V-PRINT or V-SPELL.

'*command*' may contain "**|@(*r*)**" to use the contents of text register '*r*' as all or part of the OS command.

Upon returning to VEDIT, the prompt "Press any key to continue" is displayed. This lets you see the DOS screen before the windows are rewritten. **System("***command***",NOMSG)** suppresses this prompt.

System("*command***",LOCAL)** suppresses the screen rewrite upon returning to VEDIT. This is useful when executing several **System("***command***")** commands consecutively.

System("*command***",OK)** suppress all screen prompts and rewrites; it also suppresses scrolling the screen before executing the command. This is useful for OS commands that run completely invisibly.

Return: Returns the "return code" (DOS: "error-level") of the last OS program executed.

Notes: Windows/DOS only: The DOS prompt will change to that specified with **Config_String(OS_PROMPT)**. By default it is "pathname>>" as a reminder that you are shelled out.

DOS only: If you run another program from within VEDIT it may not have enough memory to run properly due to the memory used by VEDIT. You can solve this problem by either using the and supplied VSWAP program or by invoking VEDIT with the "**-S**" option. Both are described in the VEDIT User Manual. If V-SWAP is installed in memory and **{CONFIG, File Handling, Use V-SWAP when entering DOS}** is set, VEDIT will be swapped out of memory.

System( ) is similar to **{MISC, DOS shell}**.

System("*command***")** is similar to **{MISC, Run program}**.

See Also: Commands: OS_Type, Is_VSWAP
Topics: **{MISC, DOS shell}** in the VEDIT User's Manual

Examples: **System("vprint chapter1")**

Run the program V-PRINT to format and print the file CHAPTER1.VPR.

System("dir >dirfile",OK)

Reroute the DOS "dir" command into the file "dirfile". Since there is no screen output, the "OK" option runs the command "invisibly".

TO Tab_Out( )

Usage: Tab_Out(40)

Description: **Tab_Out(*n*)** types spaces to column '*n*'. If the cursor (or "write" position) is already at or past column '*n*', it types two (2) spaces.

Notes: **Win_Hor( )** can also be used to set the "write" position to any desired column. However, **Tab_Out( )** prevents overwriting existing text on the screen if the cursor is already past the desired column.

Tab_Out( ) can be re-routed to the printer using **Out_Print( ); Win_Hor( )** cannot.

TPL Template_Load("*file*")

Usage: Template_Load("c.vtm")

Description: **Template_Load("*file*")** loads the template editing macro file '*file*'. These files typically have a ".vtm" filename extension. Multiple .VTM files can be loaded into VEDIT at one time.

Notes: This command is generally not needed because VEDIT automatically loads .VTM template editing files as needed.

Template editing is usually set up by the File-open configuration feature.

The command **Config(PG_E_TEMPLA,1)** enables template editing. It is equivalent to enabling **{CONFIG, Programming, Enable template editing}**. The string variable **VTM_NAME** determines which .VTM file the current file/buffer will use; each buffer has its own **VTM_NAME** variable. It can be set with e.g. **Config(VTM_NAME, "c.vtm").**

VEDIT must be able to access the supplied macro **`regpreg.vdm`** which is automatically executed to process the .VTM file.

See Also: Commands: Config_String(VTM_NAME), Syntax_Load( )
Topics: "Template editing" and "File-open Configuration" in Chapter 5 of the VEDIT User's Manual

Time( ) - See Date( )

TT Time_Tick

Usage: Internal Value

Description: **Time_Tick** returns the time in milliseconds since VEDIT was started. It typically has a 1/18 second resolution.

TRB Translate_Block(*p,q*)
TRL Translate_Load("*file*")

Options: COLUMN, COLSET, LINESET, NORESTORE, REVERSE

Usage: Translate_Block(BB,BE) TRB(0,File_Size,REVERSE)

Description: **Translate_Block(*p,q*)** translates the block of text between file positions '*p*' and '*q*' by using the first of two translation tables. By default, it translates from ASCII to EBCDIC, but other translation tables can be loaded with **Translate_Load( )**.

Translate_Block(*p,q*,REVERSE) translates the block of text by using the second of two translation tables. By default, it translates from EBCDIC to ASCII.

During the translation, each byte in the block is converted to another byte according to the translation table. The size of the file does not change.

Translate_Block(*p,q*,NORESTORE) sets the edit position past the end of the translated block.

The command options "**COLUMN**", "**COLSET**" and "**LINESET**" specify a columnar or line-range block for translating.

Translate_Load("*file*") loads the character translation tables from '*file* ' into VEDIT. The file consists of two 256-byte tables followed by the table name.

Notes: The EBCDIC translation table **`ebcdic.tbl`** is built into VEDIT and does not need to be loaded.

Translate_Block(BB,BE) is equivalent to **{BLOCK, Edit/translate, Translate to EBCDIC}**.

Translate_Block(BB,BE,REVERSE) is equivalent to **{BLOCK, Edit/translate, Translate from EBCDIC}**.

Translate_Load( ) is equivalent to **{BLOCK, Edit/translate, Load translate table}**.

See Also: Commands: Detab_Block( ), Translate_Char( )
Topics: "Block Operations" in Chapter 3, "Translating a Block or File" in Chapter 4 of the VEDIT User's Manual

Examples: **Translate_Block(0,File_Size,REVERSE)**

Translate the entire file using the 2nd translation table, e.g. from EBCDIC to ASCII.

TRC Translate_Char(*n*)

Options: ANSI, REVERSE

Usage: Translate_Char(Cur_Char) TRC('A',REVERSE)

Description: **Translate_Char(*n*)** returns the value of character '*n*' after translating it using the first translation table, e.g. from ASCII to EBCDIC.

Translate_Char(*n*,REVERSE) returns the value of character '*n*' after translating it using the second translation table, e.g. from EBCDIC to ASCII.

Translate_Char(*n*,ANSI) returns the value of character '*n*' after translating it from the OEM (IBM-PC or DOS) character set to the best equivalent ANSI character.

Translate_Char(*n*,ANSI+REVERSE) returns the value of character '*n*' after translating it from the ANSI character set to the best equivalent OEM character.

Return: Returns the value of the translated character.

Notes: See the notes for the **Translate_Block( )** command.

See Also: Commands: Translate_Block( )

Examples: **Ins_Char(Translate_Char(Cur_Char),OVERWRITE)**

Translate the character at the edit position from ASCII to EBCDIC.

TRL Translate_Load( ) - See Translate_Block( )

T Type(*m*)
TB Type_Block(*p,q*)

Options: COLUMN, COLSET, LINESET, NORESTORE

Usage: Type(14) Type_Block(#1,#2) TB(BB,BE,COLUMN)

Description: **Type(*n*)** displays all text lines from the edit position up to and including the '*m*'th "newline". **Type(-*n*)** displays the previous '*n*' lines and all text on the current line up to the edit position. **Type(0)** displays the text (if any) preceding the edit position on the current line.

Fewer than '*m*' lines are displayed (without error) if either end of the file/edit buffer is reached.

Type_Block(*p,q*) displays the block of text starting with the '*p*'th character in the file up to, but not including the '*q*'th character. Counting starts with 0.

Type_Block(*p,q*,NORESTORE) sets the edit position past the end of the displayed block.

The command options "**COLUMN**", "**COLSET**" and "**LINESET**" specify a columnar or line-range block for display.

Notes: The commands **Type(0) T( )** displays the current line regardless of where the edit position is on it. The **Type( )** command is useful in iteration loops for displaying selected lines.

The **Type( )** command displays characters according to the current display mode. The display mode can be changed with **{CONFIG, Characters/Cursors, Screen display mode}** or the equivalent **Config(S_DSP_MODE)**.

See Also: Commands: Line( ), Print( ), Reg_Type( ), Out_Print( )
Topics: "How Control and Graphics Characters are Displayed" in Chapter 4 of the VEDIT User's Manual

Examples:

```
Begin_Of_File( )
Repeat(ALL) {
   Search("money",ADVANCE+ERRBREAK)
   Message("Line #:") Cur_Line Type(0)Type( )
}
```

Display every line with the word "money" in it following the line number on which it appears.

TC Type_Char( ) - See Char_Dump( )

TF Type_File("*file*")

Options: NOMSG

Usage: Type_File("file.txt") TF("a:file.txt",200,400,NOMSG)

Description: **Type_File("***file***")** displays a range of lines from another file. If no line range is specified, the entire file is displayed. Line numbers are displayed before each line unless suppressed with the "**NOMSG**" option. The displayed line numbers may subsequently be used with the **Ins_File( )** command to extract a portion of the file.

Notes: Press **[CANCEL]** (<**Ctrl-**\>) or <**Ctrl-C**> to stop the file display.

See Also: Commands: Ins_File( )

Examples:

Type_File("library.asm") Display the entire file "library.asm" with line numbers.

Type_File("library.asm",1,MAXNUM,NOMSG)
Display the entire file "library.asm" without line numbers.

Out_Ins Type_File("vedit.key") Out_Ins(CLEAR)
Insert the file VEDIT.KEY, with line numbers, into the edit buffer at the current edit position.

TN Type_Newline(*n*)
TS Type_Space(*n*)

Usage: Type_Space(2) Type_Newline( )

Description: **Type_Space(*n*)** types '*n*' blanks (spaces) to the console.

Type_Newline(*n*) types '*n*' "newlines" to the console.

Notes: These commands are primarily used in command macros which are creating a "menu" or "form" on the screen. **Type_Newline( )** is often used to force the next prompt to start several lines down; however, "**\n**" can also be used to start a prompt on a new line.

The **Win_Vert( )** and **Win_Hor( )** commands can also be used to position the cursor anywhere in the window.

See Also: Commands: Ins_Newline( ), Tab_Out( ), Win_Vert( ), Win_Hor( )
Topics: "Screen Display Commands" in Chapter 3

Examples: **Type_Newline(5) Get_Input(9,"Enter filename: ")**

Skip down five lines before displaying the prompt.

UD Undo_Delete( )
UE Undo_Edit( )
UL Undo_Line( )

Usage: Undo_Delete( ) UE( )

Description: **Undo_Delete( )** inserts the top item (block of text) from the deletion stack into the edit buffer. The "first-in, last-out" stack has five levels. Any deletion of three or more characters is added (pushed) onto the deletion stack.

Undo_Line( ) undoes the edit changes made to the current line. At the "COMMAND:" prompt, all changes since the last "COMMAND:" prompt are undone.

Undo_Edit(*n*) undoes the last '*n*' edit changes. Issued again, it backs up further.

Notes: **Undo_Delete( )** and **Undo_Edit(1)** and **Undo_Line( )** are equivalent to the **{EDIT, UNDO, Delete}** and **{EDIT, UNDO, Edit}**, and **{EDIT, UNDO, Line}** functions. All changes from one "COMMAND:" prompt to the next are considered one edit operation even though it may involve hundreds of commands (and require hundreds of undo levels). Command macro operations can only be undone if **Config(CM_E_UNDO)** is set to "1" or "2".

See Also: Topics: "Undo in Command Mode" in Chapter 2

Examples:

```
Del_Line(10)
End_Of_File
Undo_Delete
```

Delete 10 lines of text, move to the end of the file and re-insert the 10 lines of text.

UR Undo_Reset

Usage: Undo_Reset( )

Description: **Undo_Reset( )** resets the Undo facility and clears both the main Undo stack and the deletion stack. No edit operations performed before the **Undo_Reset( )** can be undone.

Notes: **Undo_Reset( )** and the equivalent **{UNDO, Reset}** function have several applications. With constrained memory, it may make more memory available to the text registers and edit buffers. It also prevents someone from "undoing" too far back.

For example, you could reset the Undo to establish a starting point. Then try out some edit changes (in either Visual or Command Modes). If desired, you can then return to the starting point with the command **Undo_Edit(1000)**.

See Also: Topics: "Undo in Command Mode" in Chapter 3.

U Update( )

Options: SUPPRESS

Usage: Update( ) Update(SUPPRESS)

Description: **Update( )** displays (updates) the current edit buffer in a window. If the current edit buffer does not have an attached (associated) window, it creates a new overlapping window, similar to entering Visual Mode. If the attached window is not fully visible, i.e. other windows are overlapping it, it is made fully visible by bringing it to the top.

Update(SUPPRESS) does not make the attached window fully visible, i.e. it does not change the order of the windows on the screen. It is possible that the updated window is completely overlapped and not visible.

Update( ) is primarily used inside macros that need to update a Visual Mode window. Visual Mode window(s) normally update automatically while waiting for keyboard characters.

See Also: "Windows" in Chapter 4 of the VEDIT User's Manual.

Ver Version( )
VN Version_Num

Usage: Ver Version

Description: **Version** displays the VEDIT version number in the format: "VEDIT Ver. 6.02 03/20/02".

Version_Num returns the current version number as an integer, e.g. "602". This is primarily useful in macros which need to test for a specific version of VEDIT.

Return: **Version_Num** returns the version of VEDIT currently in use.

Notes: **{HELP, Status display}** also displays the version number.

V Visual( )

Usage: Visual Visual(10)

Description: **Visual( )** enters Visual Mode. Visual Mode is exited with either the **[VISUAL EXIT]** or **[VISUAL ESCAPE]** functions.

Visual(*n*) enters Visual Mode, but exits automatically after '*n*' operations (keystrokes) are performed. It has limited use, but is used in our "vi" editor emulation.

The error "KEYBOARD LAYOUT CORRUPTED" is given if the keyboard layout table is invalid. This is most likely due to loading the wrong file with the **Key_Load( )** command. If this happens, load a valid table or give the command **Exit** to save all files and exit VEDIT.

See Also: Topics: "Return (exit) to Visual Mode" in Chapter 2

VM Visual_Macro(*n*)

Usage: Visual_Macro Visual_Macro(SET) Visual_Macro(4)

Description: **Visual Macro** returns the current value of the "Visual Mode macro" flag. This (technical) flag determines how VEDIT reenters Visual Mode after a macro executes. It consists of several "Mask" (bit) values that are added together.

08 Macro originated from Visual Mode. Auto-return to Visual Mode in place of the "COMMAND:" prompt (or a locked-in macro).

04 Display "Press any key to continue..." upon entering Visual Mode if the window contains Command Mode output.

01 Enter Visual Mode on startup.

Masks "08" and "04" are set when a command macro is executed from Visual Mode, e.g. **{MISC, Load/Execute macro}** or a keystroke macro containing **[VISUAL EXIT]**.

Visual Macro(SET)sets masks "08" and "04" to force an auto-return to Visual Mode with a possible "Press any key to continue..." prompt.

Visual Macro(CLEAR) disables the Visual Mode macro flag. It can be used to prevent an auto-return to Visual Mode due to an error or other break out. It should be used with (non-trivial) macros that use **Reg_Lock_Macro()** to trap errors and other break outs. Some of the supplied macros, such as **`compare.vdm`** and **`wildfile.vdm`** use **Visual Macro(CLEAR)**.

Visual Macro(*n***)** sets the Visual Mode macro flag to value '*n*'. Trying the commands **Visual_Macro(8)** and **Visual_Macro(12)** from the "COMMAND:" prompt will help you understand this command.

See Also: Commands: Break_Out( ), Reg_Lock_Macro( ), Reg_Prot()

WA Win_Attach(*w*)
WDET Win_Detach(*w*)

Options: LINKED

Usage: Win_Attach(5) Win_Detach(9)

Description: **Win_Attach(***w***)** explicitly attaches window '*w*' to the current edit buffer. More than one window can be attached to an edit buffer; this is needed to display different regions of a file in separate windows. The first window attached is the "primary" window and additional attached windows are "secondary" windows.

Win_Detach(*w***)** detaches window '*w*' from any edit buffer. When a window is attached, it is also detached from any other edit buffer. An edit buffer uses its "primary" window for editing in Visual Mode.

If no window has been explicitly attached to an edit buffer before it is displayed in Visual Mode, the edit buffer will automatically attach itself to a window by the same number as the buffer. This is described under "Windows and Edit Buffers" in Chapter 2.

Win_Attach(*w***,LINKED)** attaches the window and "locks" its cursor position to the current window so that both windows scroll simultaneously. This is used by **{VIEW, Hex-mode split}**.

Notes: **Win_Switch(***w***,ATTACH)** switches to a "secondary" window and makes it the new "primary" window. A window can be attached to at most one edit buffer at a time. **Win_Status(***w***)** lets you determine to which edit buffer (if any) a window is attached.

See Also: Commands: Buf_Switch( ),Win_Switch( ), Win_Status( )
Topics: "Window Commands" in Chapter 2

Examples:

Buf_Switch(2)	Switch to edit buffer 2; create window "1" on the right and attach it to buffer 2.
Win_Split(1,40,RIGHT)	
Win_Attach(1)	

WB Win_Border
WCOL Win_Cols
WL Win_Lines

Usage: Internal Values

Description: **Win_Border** returns the type of borders in the current window: 0 = no borders; 1 = minimal borders; 2 = full borders without scroll bars; 3 = full borders with scroll bars. This is primarily only useful in the DOS version.

Win_Cols returns the number of text columns in the current window.

Win_Lines returns the number of text lines in the current window.

Notes: The setting of **{CONFIG, Display options, Window borders}** and any options on the **Win_Split( )** command determine the type of borders a window will have.

WCASC Win_Cascade( )
WTILE Win_Tile()

Usage: Win_Cascade( ) Win_Tile( )

Description: **Win_Cascade( )** moves and resizes all windows so that they overlap each other in a cascading (offset) fashion with the top-left corner of each window border visible. The most recently used windows will be on top.

Win_Tile( ) moves and resizes all windows so that all windows fit on the screen without overlapping (the resulting windows may be very small). The windows are tiled (left to right, top to bottom) in the order in which they were created.

Notes: These commands do not affect "reserved" windows; they use the entire screen except for screen lines used by reserved windows.

Win_Cascade( ) is equivalent to **{WINDOW, Cascade}**.

Win_Tile( ) is equivalent to **{WINDOW, Tile}**.

See Also: Topics: "Window Commands" in Chapter 2

Win_Cfg_Pop( )
Win_Cfg_Push( )

Usage: Win_Cfg_Push( ) Win_Cfg_Pop( )

Description: **Win_Cfg_Push( )** saves the current window arrangement by "pushing" it onto the same text register stack as used by **Reg_Push( )**.

Key_Cfg_Pop( ) restores the previous window arrangement by "popping" it from the text register stack.

Notes: These commands allow a macro to save the current window arrangement, delete, create and split windows for the macro, and then restore the window arrangement when the macro is done.

These commands must be used with care, or you may crash VEDIT. In particular, these commands must be balanced with any **Reg_Push( )** and **Reg_Pop( )** commands.

In order to provide some protection again accidentally crashing VEDIT, **Win_Cfg_Pop( )** checks that the first two bytes about to be popped are "C0 01" (hex); if not it gives a "NESTING (STACK) ERROR".

See also: Commands: Key_Cfg_Push( ), Reg_Push( )

WCLR Win_Clear( )
WEOL Win_EOL( )
WEOS Win_EOS( )

Usage: Win_Clear( ) Win_EOL( ) Win_EOS( )

Description: These commands erase part or all of the current window. (They do not delete text in the buffer. They are not editing functions.) These commands are primarily used in command macros which are creating a "menu" or "form" on the screen.

Win_Clear( ) erases (clears) the entire window and homes the cursor. **Win_EOL( )** erases from the cursor position to the end of the window line. **Win_EOS( )** erases from the cursor position to the end of the window (screen).

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

Win_Clr( ) is often used following **Win_Color( )** to select a new window color and then clear the window using the new color.

See Also: Commands: Win_Color( ), Win_Hor( ), Win_Vert( ),
Topics: "Window Commands" in Chapter 2

Examples:

Win_Vert(5) Win_Hor(15) **Message("Name:")** **Win_EOL**	Position the cursor to row 5, column 15 in the window, display the text "Name:" and clear the rest of the line.

WC Win_Color(*n*1,*n*2)
WCE Win_Color_Erase

Options: EXTRA

Usage: Win_Color(6) Win_Color(6,127)

Description: **Win_Color(*n*)** changes the current window's text (and erase) color to '*n*'. Any writing to the window will be made in the new color.

Win_Color(*n1,n2***)** changes the current window's text color to '*n1*' and the "erase" color to '*n2*'. Text written to the window will be displayed in the color '*n1*', while any window erasing and scrolling will be in the color '*n2*'. This gives an unusual effect, but can be useful for clearly see any trailing spaces on a line.

Win_Color(*n1,n2***,EXTRA)** sets the configured text/erase attributes to '*n1*' and '*n2*', so that any subsequently created windows will have those attributes. Normally, '*n1*' and '*n2*' will have the same value.

Win_Color, without parameters, only returns the value of the current window's text color without changing colors.

Win_Color_Erase, without parameters, only returns the value of the current window's "erase" color without changing colors.

Return: Returns the (new) value of the current window's text color.

Notes: The attribute values are hardware dependent. On an IBM PC all allowable screen attribute values are supported; both foreground and background colors can be set. The on-line help for **{CONFIG, Colors}** and Appendix F of the VEDIT User Manual list most color values.

On a (non-IBM PC) CRT terminal, only "0" for normal video and "1" for reverse video are supported.

The default attribute values are set during configuration. The **Screen_Init( )** command restores the attributes to their configured values.

Before changing the attributes inside a macro, you can save the current values by reading the internal values **Win_Color** and **Win_Color_Erase** commands.

See Also: Commands: Color_Highlight, Color_Prompt
Topics: "Window Commands" in Chapter 2

Examples:

Win_Color(23)	Change to white characters on blue background on a color IBM PC.

WCRE Win_Create(*w,l,c,nl,nc*)

Options: ATTACH, PIXEL

Usage: Win_Create(9,5,20,10,40) Win_Create(2,0,0,0,0,PIXEL)

Description: **Win_Create(***w,l,c,nl,nc***)** creates the overlapping window '*w*' and switches to it. The window's top-left corner origin is at line '*l*' and column '*c*'. Its size is '*nl*' text lines and '*nc*' columns (not including borders and scroll bars). Overlapping windows always have full borders.

Win_Create(...,ATTACH) creates the window and attaches it to the current edit buffer.

Win_Create(...,PIXEL) specifies the new window's origin and size in exact pixels. (Windows only)

Win_Create(*w*,0,0,0,0,PIXEL) creates a full-sized overlapping window.

Notes: The total size of the window will be two more than '*nl*' and '*nc*' because of the full borders.

See Also: Commands: Win_Delete( ), Screen_Init( ), Win_Split( ), Win_Switch( ), **{WINDOW}** menu
Topics: "Window Commands" in Chapter 2

Examples:

Win_Create(9,5,20,10,40)	Create window "9" with its top-left border corner at origin line 5, column 20, and a size of 10 text lines and 40 columns.

WDEL Win_Delete(*w*)
SI Screen_Init( )

Options: ALL, ATTACH

Usage: Win_Delete(4) Screen_Init(ATTACH)

Description: **Win_Delete( )** deletes the current window and **Win_Delete(***w***)** deletes window '*w*'. If '*r*' was created as a split window, its parent window(s) will expand in size. Only the main window "1" cannot be deleted. The new active window is the "parent" of the deleted window. This command does not affect the edit buffers or change the status of any files.

Screen_Init( ) deletes all windows and then auto-creates the main window #1 as a full-sized window. The window color attributes are reset to their configuration values.

Screen_Init(ATTACH) initializes the screen, deleting all windows. It then auto-creates an overlapping full-sized window for each edit open buffer, if **{CONFIG, Auto-create windows for buffers}** is enabled.

Screen_Init(ALL) initializes the screen and deletes all windows. Window #1 is not auto-created; VEDIT will temporarily have no windows open. A window will be auto-created for the next displayed output. If it is Command Mode output, the Command Mode window "$" is created; if it is Visual Mode, a window with the same ID (number) as the edit buffer is created.

Notes: **Win_Delete(***w***)** is equivalent to **{WINDOW, Delete}**.

Screen_Init(ATTACH) is equivalent to **{WINDOW, Reset}**.

You may need a **Win_Switch( )** command to switch to the desired window following a window deletion.

Macros should not assume that Window #1 always exists, or even that any window exists.

See Also: Commands: Win_Status(w)
{WINDOW} menu
Topics: "Window Commands" in Chapter 3

Examples:

Win_Delete(4)	Delete window "4" and switch to window "1".
Win_Switch(1)	

WDET Win_Detach( ) - See Win_Attach( )

WDM Win_Display_Mode(*n*)

Usage: Internal Values

Description: **Win_Display_Mode(*n*)** changes the text display mode in the current window to '*n*'. Valid values are:

0 Display graphics chars; display CTRL except CR/LF literally.
1 Display graphics chars; display CTRL chars as "^x".
2 Display graphics chars as "<nnn> "; display CTRL chars literally.
3 Display graphics chars as "<nnn> "; display CTRL chars as "^x".
4 Display graphics chars; display CTRL & CR/LF literally.
8 Display all character in Hexadecimal.
16 Display all character in Octal.
32 Display all characters in EBCDIC.

Win_Display_Mode, without parameters, just returns the current display mode without changing it.

Return: All forms of the **Win_Display_Mode( )** command return the (new) display mode in the current window.

Notes: To change the configured value of the text display mode which will be used for new windows, use **{CONFIG, Characters/Cursors, Screen display mode}** or **Config(S_DSP_MODE)**.

See Also: Topics: "Screen Display and Keyboard Characters" in Chapter 4 of the VEDIT User's Manual.

WEOL Win_EOL( ) - See Win_Clear( )

WF Win_Free
WX Win_Next
WPRV Win_Previous

Usage: Internal Values

Description: **Win_Free** returns the ID number of the next unused (free) window. This ID number can subsequently be used in a **Win_Split( )** or **Win_Create( )** command to create new windows.

Win_Next returns the ID number of the next window attached to any edit buffer. If there is only one window, it returns the ID number of the current window.

Win_Next(BUFFER) returns the window ID number of the next (higher numbered) window attached to the current buffer; if there is no such window, it returns the ID number of the lowest numbered window attached to the next buffer. If there is only one buffer, it is the same as **Win_Next**. It is used internally by **{WINDOW, Next window}**.

Win_Previous returns the ID number of the previous window attached to any edit buffer. If there is only one window, it returns the ID number of the current window.

Win_Previous(BUFFER) returns the window ID number of the previous (lower numbered) window attached to the current buffer; if there is no such window, return the ID number of the highest numbered window attached to the previous buffer. If there is only one buffer, it is the same as **Win_Previous**. It is used internally by **{WINDOW, Previous window}**.

See Also: Commands: Buf_Window_Next, Buf_Window_Previous

WINH Win_Height (Windows only)
WINW Win_Width
WINX Win_X_Org
WINY Win_Y_Org

Usage: Internal Values

Description: **Win_Height** and **Win_Width** return the size of the current editing window in pixels.

Win_X_Org and **Win_Y_Org** return the horizontal and vertical origin of the upper-left corner of the current editing window. The origin is in pixels relative to the upper-left corner (0,0) of the editing area within VEDIT, just below the optional toolbar and ruler.

See Also: Commands: App_Height, App_Width, Desktop_Height, Desktop_Width, Win_Move( )

Examples: **Win_Move(Win_Num,WINX+10,WINY,WINW,WINH)**

Move the current window to the right by 10 pixels.

WH Win_Hor(*n*)
WV Win_Vert(*n*)

Usage: Win_Vert(5) Win_Hor(15) Win_Hor(1) Win_Vert(1)

Description: These commands position the cursor within the current window for subsequent text that is displayed in the window by the macro commands.

Win_Hor(*n*) positions the cursor horizontally to column '*n*'. The leftmost column is "1".

Win_Vert(*n*) positions the cursor vertically to row (line) '*n*'.

Win_Hor and **Win_Vert**, without parameters, only return the current cursor position without moving it.

Return: All forms of the **Win_Hor( )** and **Win_Vert( )** commands return the current (or new) cursor position.

Notes: These commands have nothing to do with the cursor positioning in Visual Mode; they perform "CRT emulation" functions for command macros, e.g. to create a "menu" or "form" on the screen.

See Also: Commands: Win_Clear( ), Win_EOL( ), Win_EOS( )
Topics: "Window Commands" in Chapter 3

Examples:

Win_Vert(5)	Positions the cursor to row 5, leaving it in its current column.
Win_Vert(1) Win_Hor(1)	Positions the cursor to "home" - the upper left corner of the window.

Win_Level(*w*)
Win_Overlap

Usage: Internal Values (Very technical and rarely used)

Description: **Win_Level(*w*)** returns the display (overlapping) level for window '*w*'.

0	entire window is visible
n	up to 'n' windows are overlapping window '*w*'.

The current window is always at level 0.

Win_Overlap returns the overlapping status of the windows on the screen. It can be used to determine if any windows are overlapping each other. Values are:

-1	No text windows exist
0	One full sized text window
1	Two or more text windows; active window is full (zoom) size
2	Split (child) windows cover full screen
3	One text window; not full sized
4	All overlapping-style windows fully visible
5	Text windows are overlapping

Win_Move(*w*,*x*,*y*,*cx*,*cy*)

Usage: Win_Move(Win_Num,0,0,Win_Width,Win_Height)

Description: **Win_Move(*w,x,y,cx,cy*)** moves and/or resizes window '*w*' so that it has pixel origin '*x*','*y*' and size '*cx*' (width) and '*cy*' (height). An origin of "0,0" is the upper-left corner of the editing area within VEDIT, just below the optional toolbar and ruler.

Win_Move(APP,*x,y,cx,cy*) moves and/or resizes the entire VEDIT program (application) window. An origin of "0,0" is the upper-left corner of the Windows desktop.

See Also: Commands: App_Height, App_Width, Desktop_Height, Desktop_Width, Win_Height, Win_Width

Examples:

Win_Move(Win_Num,0,0,WINW,WINH)

Move the current window to the upper-left-most corner.

Win_Move(WN,WINX,WINY,WINW+FONTW,WINH+FONTH

Make the current window one text column wider and one text line taller.

Win_Move(APP,(DTW-APPW)/2,(DTH-APPH)/2,APPW,APPH)

Center VEDIT on the desktop. Often used after **Screen_Size( )**.

WX Win_Next - See Win_Free

WN Win_Num
WT Win_Total

Usage: Internal Values

Description: **Win_Num** returns the ID number of the current window.

Win_Total returns the total number of existing windows. It is not affected when a window is zoomed.

Examples:

```
#101 = Win_Num
Win_Switch('H')
...
Win_Switch(#101)
```

The current window's ID number is saved in #101. After switching to another window and perform other commands, the macro switches back to the original window.

Win_Page_Size

Usage: Internal Value

Description: **Win_Page_Size** returns the number of lines scrolled by each **[PAGE DOWN]** and **[PAGE UP]** key. It is the number of lines in the screen minus the amount of overlap.

The amount of overlap can be changed with **Config(S_PG_OVERLAP)**. The configured value is with respect to a 25 line window, therefore a window with more lines will automatically have more overlap.

WR Win_Reserved(*w*,*n*,*location*)
WSPL Win_Split(*w*,*n*,*location*)

Options: ATTACH, BOTTOM, TOP, LEFT, RIGHT, NOBORDER, MINBORDER, FULLBORDER

Usage: Win_Split(2,40,RIGHT) Win_Reserved($,5,BOTTOM)

Description: **Win_Reserved(***w*,*n*,*location***)** creates a new "reserved" window of '*n*' lines at the "TOP" or "BOTTOM" of the screen. All other windows are resized to account for these reserved screen lines. Reserved windows cannot be overlapped and are therefore always visible on the screen.

Win_Split(*w*,*n*,*location***)** creates a new window by splitting the current window into two windows. It creates window '*w*' of size '*n*' lines/columns in the '*location*' specified (BOTTOM, TOP, LEFT, RIGHT).

'*w*' is a number between 2 and 127. Values 2 - 32 create normal "numbered" editing windows. Values 33 - 127 create special "named" windows, which out of convention should not be used as Visual Mode editing windows; however they can be used in command macros for prompts, menus, help lines, etc. Named windows display their equivalent ASCII character ID name on their border.

The window with name "$" is the special Command Mode window.

The special window size of "0" splits the window into two equal sized windows.

The command option "**+ATTACH**" creates a new window and attaches it to the current edit buffer. This is equivalent to following the command with **Win_Attach(***w***)**.

The command option "**+NOBORDER**" creates a new window without borders (when possible). The option "**+MINBORDER**" creates a new window with minimal borders. The option "**+FULLBORDER**" creates a new window with full borders.

Notes: **Win_Split( )** reduces the size of the current window as needed; however, an error is given if it attempts to reduce the current window

smaller than one text line and 10 columns. This command does not switch to the new window; this is done with the **Win_Switch( )** command.

The screen attributes for the new window can be changed with the **Win_Color( )** command. This allows windows to be displayed in different colors.

For named windows, the window ID '*w*' can be entered as a normal numeric ASCII constant, e.g. '**A**' (quote A quote). As a shortcut, the single-quotes can be left off. Therefore, the following are equivalent: **Win_Split('$',5,BOTTOM)** and **Win_Split($,5,BOTTOM)**.

See Also: Commands: Win_Create( ), Win_Color( ), Win_Delete( ), Screen_Init( ), Win_Switch( ), Win_Cols, Win_Lines, Win_Reserved_Bottom, Win_Reserved_Bottom_ID,Win_Total
{WINDOW} menu
Topics: "Window Commands" in Chapter 2

Examples:

Win_Split(2,40,RIGHT) Create window "2" with 40 columns in the right part of the current window.

Win_Reserved($,5,BOTTOM)

Create the special Command Mode window as a reserved window with 5 lines at the bottom of the screen. All other windows are resized as needed.

WRB Win_Reserved_Bottom (Windows only)
WRBID Win_Reserved_Bottom_ID
WRT Win_Reserved_Top
WRTID Win_Reserved_Top_ID

Usage: Internal Values

Description: **Win_Reserved_Bottom** and **Win_Reserved_Top** return the number lines in the "reserved" windows at the bottom/top of the VEDIT screen area; return 0 if none.

There can be at most one reserved window at the top and one at the bottom. Reserved windows cannot be overlapped by other windows, and are often used to display a prompt, help line or the "COMMAND:" prompt.

Win_Reserved_Bottom_ID and **Win_Reserved_Top_ID** return the window ID number of the reserved window at the bottom/top of the screen; return 0 if none.

Notes: The **`keyedit.vdm`** macro uses **Win_Reserved_Bottom** and **Win_Reserved_Bottom_ID** to check if there is a reserved window, temporarily delete it, and restore it when the macro is done.

See Also: Commands: Win_Reserved( )
Topics: "Window Commands" in Chapter2

WSM Win_Scroll_Margin(*n*)

Usage: Win_Scroll_Margin(80) WSM(0) Win_Scroll_Margin

Description: **Win_Scroll_Margin(*n*)** sets the horizontal scroll amount for Visual Mode to '*n*'. This is the column corresponding to the left edge of the window. Visual Mode will only use this value if:

n < cursor column < n+window width.

Win_Scroll_Margin, without parameters, only returns the current scroll value without changing it.

Return: All forms of the **Win_Scroll_Margin( )** command return the (new) value of the horizontal scroll margin in the current window.

Notes: **[SCROLL LEFT]** and **[SCROLL RIGHT]** change the scrolling.

See Also: Commands: Set_Visual_Line( )
Topics: "Scrolling the Screen" in Chapter 4 of the VEDIT User's Manual

Examples: **Win_Scroll_Margin(20)** Scroll the screen by 20 columns. This is similar to **[SCROLL RIGHT]**.

WSPL Win_Split( ) - See Win_Reserved( )

WSTAT Win_Status(*w*)

Usage: Internal Values

Description: **Win_Status(*w*)** returns the status of window '*w*':

-1	window does not exist
0	window exists, but not attached to any buffer
'*n*'	window is attached to edit buffer '*n*'.

See Also: Commands: Buf_Status

WS Win_Switch(*w*)

Options: ATTACH, STATLINE

Usage: Win_Switch(3) WS($) Win_Switch(1,ATTACH)

Description: **Win_Switch(*w*)** switches to the window '*w*'. If window '*w*' does not exist, the command is ignored — no error is given. **Win_Switch(*w*)** only switches to a window; it does not also switch edit buffers. Command mode output will be displayed in the new window.

Win_Switch(*w*,ATTACH) switches to window '*w*' and, if it is attached to an edit buffer, switches to the edit buffer and makes the new window the buffer's "primary" editing window. (Any previous "primary" window becomes a "secondary" one.) It also moves the edit position to the cursor position in the new window.

Win_Switch(STATLINE) switches to use the status line as a one line window. This permits displaying messages on the status line and is more flexible than using the **Get_Input( )**, **Get_Num( )** and **Message( )** commands with the "STATLINE" option. You should explicitly switch back to a normal window when done. The status line will automatically refresh at the next "COMMAND:" prompt or when entering Visual Mode.

Notes: When **Win_Switch(***w***)** 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 will automatically switch to the "$" window if it exists. Otherwise it will use the previous Command Mode window, or if there is none, the current window.

Win_Num returns the ID number of the current window.

Win_Switch(*w***,ATTACH)** is equivalent to **{WINDOW, Switch}**.

See Also: Commands: Win_Attach( ), Win_Num, Win_Status
Topics: "Window Commands" in Chapter 3

Examples:

Win_Switch(1)	Switch to window 1.
Win_Switch(10,ATTACH)	Switch to window 10 and, if it is attached to an edit buffer, switch to the edit buffer and make it the buffer's "primary" editing window.

WTILE Win_Tile( ) - See Win_Cascade( )

WTTL Win_Title(*r*,"*text*")

Usage: Win_Title(Win_Num,"List of files") Win_Title(Win_Num)

Description: Win_Title(***r*,"*text*"**)** sets the title for window '*w*' to '*text*'. The title is displayed on the window's top border. This overrides the default title, such as the filename, that VEDIT displays for each window.

'*text*' is limited to 80 characters. Titles longer than the width of the window will be truncated; however, if the title contains a "\", only the text to the right of the "\" may be displayed in narrow windows.

Win_Title(***r*)** restore the default title for window '*w*'.

WT Win_Total - See Win_Num

WV Win_Vert( ) - See Win_Hor( )

WINW Win_Width - See Win_Height
WINX Win_X_Org - See Win_Height
WINY Win_Y_Org - See Win_Height

WZ Win_Zoom( )

Options: CLEAR, TOGGLE

Usage: Win_Zoom( )

Description: **Win_Zoom( )** "zooms" (maximizes) the editing windows so that it fills the entire screen (except for the status line). It is often easier to edit in the larger window. **Win_Zoom(CLEAR)** de-zooms (restores) the window.

Win_Zoom(TOGGLE) toggles between zoom/de-zoom; this is equivalent to **{VIEW, Zoom}**.

Notes: Use the internal value **Is_Zoomed** to test if a window is zoomed.

See Also: Commands: Screen_Init( ), Win_Switch( ), Is_Zoomed
{VIEW} menu

Examples: **Win_Zoom( )** Zoom the current window to fill the screen.

Write_Line("*file*",*m*)

Usage: Write_Line("file1",10)

Description: **Write_Line("*file*",*m*)** copies '*m*' lines of text to '*file*'. The range of lines copied is the same as for the **Del_Line( )** or **Type( )** commands. The text in the edit buffer is unchanged.

If '*file*' already exists, a confirmation prompt to overwrite the file is given; **Write_Line(...,OK)** skips the confirmation prompt.

Use the closely related **Block_Save_As( )** command to write a stream or columnar block to a file.

See Also: Commands: Block_Save_As( ), Ins_File( ), Type( )
Topics: "Block Operations" in Chapter 3

Examples: **BOF() Write_Block("part1",100)**

Copy (write) the first 100 lines in the current file to the file "part1".

XALL - See Exit( )

XBUF1 - See Extra_Buffer_1

Appendices

A - Edit Function Codes

Each edit function has a corresponding two letter code. These codes are used by the **Do_Visual( )** and **Message( )** commands. The DOS version `vphelp.hlp` help file also uses them. The **Get_Key( )** and **Prev_Key( )** commands convert the two letter code to an equivalent numeric value. For example, the numeric value for "\BS\" is **'B'+'S'*256** = 21314.

These codes are also listed in the on-line help topic "Edit Function Codes" (DOS help topic "VMCODES").

Table of Edit Function Codes

Function name	Letter Code	Function name	Letter Code
[BACKSPACE]	\BS\	[NEXT WORD]	\NW\
[CANCEL]	\CA\	[PAGE UP]	\PU\
[CURSOR UP]	\CU\	[PAGE DOWN]	\PD\
[CURSOR DOWN]	\CD\	[PREV PARAGRAPH]	\PP\
[CURSOR RIGHT]	\CR\	[PREV WORD]	\PW\
[CURSOR LEFT]	\CL\	[REPEAT]	\RE\
[DELETE]	\DC\	[REPEAT LAST]	\RL\
[DEL PREV WORD]	\DP\	[RETURN]	\RT\
[DEL NEXT WORD]	\DN\	[SCREEN BEGIN]	\SB\
[ENTER CTRL]	\EC\	[SCREEN END]	\SE\
[ERASE BOL]	\EB\	[SCROLL UP]	\SU\
[ERASE EOL]	\EE\	[SCROLL DOWN]	\SD\
[ERASE LINE]	\EL\	[SCROLL RIGHT]	\SR\
[ESCAPE]	\ES\	[SCROLL LEFT]	\SL\
[HELP]	\HE\	[TAB CHARACTER]	\TC\
[INSERT TOGGLE]	\IT\	[T-REG COPY]	\RC\
[LINE BEGIN]	\LB\	[T-REG MOVE]	\RM\
[LINE END]	\LE\	[T-REG INSERT]	\RI\
[MENU]	\ME\	[VISUAL ESCAPE]	\VS\
[NEXT LINE]	\NL\	[VISUAL EXIT]	\VE\
[NEXT PARAGRAPH]	\NP\	All other keys	\B4\
[NEXT TAB STOP]	\NT\		

B - Command Syntax

Basics

VEDIT's command syntax loosely follows the syntax of the "C" programming language. The format of commands is:

Command(arguments)

Command names can be entered in any combination of upper and lower case letters. To improve readability, we usually capitalize the first letter of each command word, e.g. **Type_Space( )**.

The "**_**" character is optional and is only intended to improve readability.

Most commands have a short abbreviation. It is often, *but not always*, the first letter of each command word.

Therefore, the following commands are all identical:

Type_Space( ) typespace( ) TS( ) ts( )

Commands that take no arguments can usually have the "**( )**" left off. Our convention is to include the "**( )**" for commands that perform an operation and leave the "**( )**" off for commands that only return a value.

Commands that take a single numeric argument, e.g. **Type_Space( )**, will use the default argument of "1" if no argument is specified.

Therefore, the following commands are all identical:

Type_Space(1) Type_Space( ) Type_Space TS

Commands may be entered one per line or many per line.

Arguments

Commands take *numeric* and/or *string* arguments. Arguments are enclosed in **(...)** following the command name. When there are two or more arguments, they are separated from each other with commas.

Each numeric argument can be a numeric expression consisting of numeric constants (e.g. **12345**), numeric variables (e.g. **#10**), reserved words (e.g. **ALL**) or the return value from another command (e.g. **File_Size**).

Numeric arguments have the range of +/- 2,147,483,647. When a large number is needed, for example to specify an "infinite" repeat count, the reserved word "**ALL**" can be used; its value is greater than one billion.

Each string argument can either be a string constant (e.g. **"hello"**) a string variable stored in a text register (e.g. **@20**), or one of the predefined string values, (e.g. **CUR_DIR**).

Command Options

Command options are usually specified with reserved words such as **NOERR** and **STATLINE**. When two or more command options are needed, they can be added together as in **NOERR+STATLINE** or "ORed" together as in **NOERR | STATLINE**. Use of "|" is preferable, but harder to read than "+".

The command option **COUNT** is followed by an additional numeric argument.

Search("text",COUNT,3)

The command options **COLSET** and **KEYCOLS** are followed by two additional numeric arguments.

Search_Block("text",Block_Begin,Block_End,COLSET,10,40)

When **COUNT** and **COLSET** are both used, the argument for **COUNT** comes first.

ARGUMENTS

'*n*'	A positive number or expression in the range 0 - 2,147,483,647.
'*m*'	A number or expression which may be negative.
'*c*'	A conditional expression. Value of "0" is FALSE, value of "1" (or any other non-zero value) is TRUE.
'*r*'	A text register number, usually in range 0 - 99. Special purpose registers have number 100-127. Some commands allow an edit buffer to be specified by '*r*'+BUFFER.
'*b*'	An edit buffer number, usually in the range 1 - 99. "Extra" edit buffers have number 100 - 125.
'*w*'	A window number/name. Window numbers are in the range 1 - 36. Values above 36 are considered special single character window names, e.g. "$" is the Command Mode window, "STATLINE" is the status line.
"*text*"	A text string in which all characters are treated literally.
"*mtext*"	A message text string. Similar to "*text*", but "**\n**" specifies a "newline" and "**\t**" a tab character. Use the **TAB8** option to assume tabs stops at every 8 columns.
"*ss*" and "*rs*"	A search/match or replacement string. Various Pattern matching or Regular expression characters have a special meaning.
"*file*"	A filename with optional drive and pathname. The syntax "|**@(*r*)**" permits the contents of text register '*r*' to be used as any portion of the filename.
"*fspec*"	Similar to "file" except that multiple files may be specified with "?" and "*".
'*p,q*'	Two positive numbers that specify file positions for a "stream" block of text. The character at position '*q*' is not included.
'*p,q*,COLUMN'	Specify a columnar block. File positions '*p*' and '*q*' specify the corners of the columnar block. The character at position '*q*' is included.
'COLSET,*c1,c2*'	Specify a columnar block. File positions '*p*' and '*q*' specify the first and last lines of the block; they can be anywhere on the line. '*c1*' and '*c2*' specify the first and last columns of the block.
'*l1,l2*,LINESET'	Specify a line-range block. '*l1*' and '*l2*' specify the first and last lines of the block. All chars on lines '*l1*' through '*l2*' (inclusive) are included.

SPECIAL CHARACTERS

//	Comment - all following characters to End-Of-Line are a comment.
Label: **:Label:**	Label for "Goto" command is followed by ":". Preceding label with ":" permits faster execution.
;	Separates items inside a "**for (...)**" statement. Otherwise ignored, but allowed for compatibility with C language syntax.
?	Breakpoint — Enter tracing mode.
?(*expr***)**	Conditional breakpoint — Enter tracing mode only if the express '*expr*' is TRUE.
??	Perform back-trace; only valid at "COMMAND:" prompt.

FLOW CONTROL

A condition '*c*' is TRUE if it has a numeric value of 1 or greater. It is FALSE if it has a value of 0 (zero) or less.

if (*c***) {**
commands
}

If '*c*' is TRUE, execute the '*commands*'. If '*c*' is false, skip over the "**{...}**" and continue with any following commands.

if (*c***) {**
commands-1
}
else {
commands-2
}

If '*c*' is TRUE, execute the '*commands-1*', then skip over the "**else{...}**" and continue with any following commands. If '*c*' is false, execute the '*commands-2*'.

while (*c***) {**
commands
}

If '*c*' is initially TRUE, execute the '*commands*'. Then re-test '*c*' and repeatedly execute '*commands*' as long as '*c*' is TRUE.

If '*c*' is initially FALSE, skip over the "**{...}**" and continue with any following commands, i.e. '*commands*' is executed zero times.

do {
commands
} while(*c***)**

Execute the '*commands*'. Test '*c*' and repeatedly execute '*commands*' as long as '*c*' is TRUE. Note: '*commands* is executed at least once.

for (*cm1***;** *c* **;** *cm2* **) {**
commands
}

Execute the command(s) '*cm1*', then test the condition '*c*'. If TRUE, execute the '*commands*'; then execute the command(s) '*cm2*' and re-test the condition '*c*'. When '*c*' is FALSE, skip over the "**{...}**" and continue with any following commands. Note: '*commands*' may be executed zero times.

repeat(*n***) {**
commands
}

Repeatedly execute '*commands*' for a total of '*n*' times. If '*n*' is zero (or negative), '*commands*' are not executed at all.

C - Numeric Expressions

NUMERIC COMPONENTS

Number	A simple decimal number in the range +/- 2,147,483,647.
0X*hhhhhh*	Hexadecimal numbers are preceded with "**0x**" or "**0h**".
'*c***'**	ASCII value of the character '*c*', e.g. **'S'** has value 83.
^*c*	ASCII value of the control character, e.g. **^C** has value 3.
#*xxx*	Numeric register *xxx*. '*xxx*' is a simple number in the range 0 - 127.
#@*xxx*	Indirect specification of a numeric register. E.g. if numeric register 10 contains 95, then "#@10" specifies numeric register 95.
Reserved-Word	Reserved words are typically used to specify command options.
Command	Each command returns a numeric value. Therefore, commands may be used as arguments within other commands.

NUMERIC OPERATORS

+	Addition
-	Subtraction (also performs unary minus function)
*	Multiplication
/	Division
%	Remainder of division
&	Bitwise AND
|	Bitwise OR
^	Exclusive OR (XOR)
<<	Left Shift
>>	Right Shift
~	Bitwise complement (also called 1's complement)

RELATIONAL OPERATORS

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

LOGICAL OPERATORS

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

PRECEDENCE OF NUMERIC OPERATORS

Table of Operator Precedence

Highest:	! ~ ++ — + -	Unary
	* / %	Multiplication, Division, Remainder
	+ -	Addition, Subtraction
	<< >>	Shift
	< > == etc.	Relationals
	&	Bitwise AND
	| ^	Bitwise OR, Exclusive OR (XOR)
	&&	Logical AND
	||	Logical OR
Lowest:	=	Assignment

NUMERIC REGISTERS

#*xxx* = *m*	Set numeric register '*xxx*' to value '*m*'.
#*xxx*++	Increment register '*xxx*'.
#*xxx*—	Decrement register '*xxx*'.
#*xxx* += *m*	Add '*m*' to register '*xxx*'.
#*xxx* -= *m*	Subtract '*m*' from register '*xxx*'.
#*xxx* *= *m*	Multiply register '*xxx*' by '*m*'.
#*xxx* /= *m*	Divide register '*xxx*' by '*m*'.

PREDEFINED VALUES (Reserved Words)

TRUE	**= 1**
FALSE	**= 0**
CLEAR	**= -1**
MAXNUM	**= 2147483647**

NOTE: "**ALL**" can be used to indicate a huge count (instead of **MAXNUM**); its value is greater than 1 billion.

D - String Arguments

STRING CONSTANT

A string constant is enclosed in "delimiters" that cannot occur in the string. The allowable delimiters are:

` ' " % & * , . : ; / ~ ^ =

Some commands take a string argument of '*mtext* which is a message to be displayed. Here "**\n**" specifies a newline (CR+LF) and "**\t**" specifies a tab character.

Examples:

Ins_Text(/My name is "TOM"/)

Message("This is line one\nThis is line two\n")

STRING VARIABLES

The text registers are used as string variables. Over 100 text registers are available, but to prevent conflicts with cut and paste operations and with loading macros, only 10 through 90 should be used as string variables.

@(*r*)	Use text register '*r*' as the string argument.
|@(*r*)	Use text register '*r*' as a portion of the string within a string constant. Only supported for search and replace strings, filenames and in selected other commands.

With the "**|@(*r*)**" syntax, '*r*' can be a numeric expression that evaluates to a text register number. This allows the text register to be specified indirectly.

NOTE: The text register can include any characters, even the string delimiter.

Examples:

Reg_Set(10,/My name is "TOM"/) **Ins_Text(@10)**	Equivalent to example above
#1=10 **Reg_Set(#1,/My name is "TOM"/)** **Ins_Text(@#1)**	Equivalent to example above; it demonstrates how a text register can be specified indirectly with a numeric variable.
Reg_Ins(10,/.txt/) **File_Open("myfile|@(10)")**	Open file "myfile.txt"
File_Open(@10) **File_Open("|@(10)")**	These two commands are identical.

PREDEFINED STRING VALUES

Several predefined string values allow easy access to commonly used string variables: (as always, the "_" is optional)

VEDIT_EXE	Full pathname to the executable VEDIT.EXE.
CUR_DIR	Full pathname of the current directory.
PATHNAME	Full pathname of the current output (write) file. (e.g. "c:\work\january\invoice.txt")
PATH_ONLY	Just the path (directory) of the current file. (e.g. "c:\work\january")
FILENAME	Just the filename of the output file. (e.g. "invoice.txt")
FILE_ONLY	Just the filename without the extension. (e.g. "invoice")
FILE_EXT	Just the filename extension with the "." (period). (e.g. ".txt")
EXT_ONLY	Just the filename extension without the "." (period). (e.g. "txt")
INPUT_FILE	Full pathname of the current input (read) file.
CMD_LINE	The complete command line with which VEDIT was invoked.
PID	The process ID number as a string (3-5 digits).

Each of the **Config_String( )** parameters can also be accessed:

HOME	VEDIT Home Directory. (e.g. "c:\vedit")
MACRO	VEDIT Macro Directory. (e.g. "c:\vedit\macros")
BACKUP	VEDIT Backup Directory.
VEDIT_TEMP	VEDIT Temporary Directory. (e.g. "c:\vedit\backup").
FILE_CFG	File-open configuration directory. (e.g. "c:\vedit\file-cfg")
USER_CFG	User's Configuration Directory. Usually the same as "HOME", unless VEDIT was installed on a network server.
USER_MACRO	User Macro Directory. (e.g. "c:\vedit\user-mac")
VEDIT_INI	Pathname to the **`vedit.ini.vdm`** file
PR_DEF	Default printer (non-Windows versions).
PR_W_CURRENT	Current Windows version printer.
OS_PROMPT	Shell prompt.
WORD_SEP	Word separating characters.
PARA_SEP	Paragraph separating characters.
MATCH_PAREN	Groups of parentheses matching characters.
TOOL_MENU	Name of the **{TOOL}** menu.
USER_MENU	Name of the **{USER}** menu.

The string name is simply used as the string argument. The string arguments '*ss*', '*rs*', '*file*' and '*fspec*' can use a string value as a portion of the string. The syntax is "**|(*string-value*)**".

Examples:

Reg_Set(10,HOME)	Text register 10 is set to the pathname to the VEDIT Home Directory
File_Open("|(HOME)\startup.vdm")	Open the **`startup.vdm`** file in the VEDIT Home Directory

E - Command Summary

	Alert( )	Sound a beep on the IBM PC speaker.
APPH **APPW**	**App_Height** **App_Width**	(Windows) Return the height and width of the VEDIT program window in pixels.
APPX **APPY**	**App_X_Org** **App_Y_Org**	(Windows) Return the horizontal (x) and vertical (y) origin of the VEDIT program window in pixels.
	At_BOB	Return 1 (TRUE) if the edit position is at the beginning of the portion of the file currently in memory.
	At_BOF	Return TRUE if the edit position is at the beginning of the file.
	At_BOL	Return TRUE if the edit position is at the beginning of a line.
	At_EOB	Return TRUE if the edit position is at the end of the portion of the file currently in memory.
	At_EOF	Return TRUE if edit position is at the end of the file.
	At_EOL	Return TRUE if the edit position is at the end of a line.
	Atoi(*r*)	Return value of numeric expression in T-Reg '*r*'. **Atoi( )** is another name for **Num_Eval_Reg( )**.
BOF	**Begin_Of_File( )**	Move to the beginning of the file.
	Begin_Of_File(LOCAL)	Move to the beginning of the portion of the file currently in memory.
BOL	**Begin_of_Line( )**	Move to the beginning of the current line.
BB	**Block_Begin**	Return the value of the block-begin marker. (-1 if not set).
	Block_Begin(*n*)	Set the block-begin marker to file position '*n*'.
	Block_Begin(CLEAR)	Clear the block-begin and block-end markers. Same as **Block_Begin(-1)**.
	BB(CLEAR+EXTRA)	Only clear the block-begin and block-end markers if **Config(E_BM_MODE)** is set to "0".
BCP	**Block_Copy(*m*)**	Copy the next/previous '*m*' lines to the current edit position. (Duplicates '*m*' lines.)
	Block_Copy(*p*,*q*)	Copy the block of text between file positions '*p*' and '*q*' to the edit position. Advance the edit position.
	Block_Copy(*p*,*q*,BEGIN)	Leave the edit position at the beginning of the block.
	Block_Copy(*p*,*q*,DELETE)	Delete the block of text from the original position after it is copied. Same as **Block_Move(*p*,*q*)**.
	Block_Copy(*p*,*q*,DELETE+FILL)	Copy (move) the block of text to the current edit position and then replace (fill) the original block with spaces.
	Block_Copy(*p*,*q*,EXTRA)	Copy the block; the block and column markers are either reset, maintained or moved to the new block, depending upon the setting of **{CONFIG, Emulation, Block marker emulation mode}**.

	Block_Copy(*p,q*,OVERWRITE)	Overwrite the characters at the edit position with the block of text being copied.
	Block_Copy(*p,q*,COLUMN)	Copy a columnar block of text. File positions '*p*' and '*q*' define the "corners" of the columnar block.
	Block_Copy(*p,q*,COLSET,*c1,c2*)	Copy a columnar block of text. File positions '*p*' and '*q*' define the lines, '*c1*' and '*c2*' define the columns of the block.
	Block_Copy(*l1,l2*,LINESET)	Copy a line-range block of text. '*l1*' and '*l2*' specify the first and last lines of the block.
BE	**Block_End**	Return value of the block-end marker (-1 if not set).
	Block_End(*n*)	Set the block-end marker to file position '*n*'. If the block-begin marker has not yet been set, sets the block-begin marker instead.
	Block_End(CLEAR)	Clear the block-end marker. Same as **Block_End(-1)**.
BFL	**Block_Fill(*ch,p,q*)**	Fill (overwrite) the block of text between file positions '*p*' and '*q*' with character '*ch*'.
	BFL(*ch,p,q*,NORESTORE)	Leave the edit position just past the end of the block.
	BFL(*ch,p,q*,RESET)	The **Block_Begin**, **Block_End** markers are cleared.
	BFL(*ch,p,q*,COLUMN)	Fill a columnar block.
	BFL(*ch,p,q*,COLSET,*c1,c2*)	Fill a columnar block.
	BFL(*ch,l1,l2*,LINESET)	Fill a line-range block.
	BFL(*ch,p,q*,INSERT)	Insert an empty block of size '*q*' - '*p*' bytes at file position '*p*' using fill character '*ch*'.
	BFL(*ch,p,q*,COLUMN+INSERT)	Insert an empty columnar block.
BM	**Block_Mode**	Return the type of block mode set for Visual Mode. 0="stream", COLUMN="column", LINEBLOCK= "line" mode.
	Block_Mode(CLEAR)	Sets "stream" mode for blocks in Visual Mode.
	Block_Mode(COLUMN)	Set "column" mode for blocks in Visual Mode.
	Block_Mode(LINEBLOCK)	Set "line" mode for blocks in Visual Mode.
BMV	**Block_Move(*p,q*)**	Move the block of text between file positions '*p*' and '*q*' to the edit position. Same as **Block_Copy(*p,q*, DELETE)**.
BSA	**Block_Save_As("*file*",*p,q*)**	Write (save) the block of text between file positions '*p*' and '*q*' to the file '*file*'.
	BSA("*file*",*p,q*,COLUMN)	Write (save) a columnar block of text.
	BSA("*file*",*p,q*,COLSET,*c1,c2*)	Write (save) a columnar block of text.
	BSA("*file*",*l1,l2*,LINESET)	Write (save) a line-range block of text.
	BOL_Pos	Return the file position at the beginning of the current line.
	Break	Breaks out of any **while**, **do-while**, **for** or **repeat** loop and continues with any commands following the "}".

	Break_Out( )	Stops all macro execution and returns to the "COMMAND:" prompt or the "locked-in" macro.
	Break_Out(EXTRA)	Stops all macro execution and returns to Visual Mode.
	Break_Out(EXTRA+CONFIRM)	Returns to Visual Mode, possibly with a "Press any key to continue" prompt.
	Break_Out(DELETE)	Stops all macro execution and deletes (empties) the currently executing text register.
	Browse_Mode	Return the value of "Browse mode" for the current file.
	Browse_Mode(SET)	Enable browse mode for the current file, same as **{FILE, Browse Mode}**.
	Browse_Mode(CLEAR)	Disable browse mode when possible.
BC	**Buf_Close( )**	Close the current edit buffer, saving the file, if any. If a modified buffer has no assigned filename, prompts for one. Switches to one of the remaining buffers.
	Buf_Close(ALL)	Close all buffers, except for the main buffer. Save and close all files possible.
	Buf_Close(CONFIRM)	Prompt whether a modified buffer is to be saved. If the user selects [No], it performs a **Buf_Quit( )** instead.
	Buf_Close(DELETE)	Delete all windows attached to the buffer being closed.
	Buf_Close(EVENT)	Execute the File-Close event macros, even if disabled.
	Buf_Close(MAINBUF)	Switch to the main buffer "1" after closing the current buffer.
	Buf_Close(NOEVENT)	Suppress executing the File-Close event macros, even if enabled.
	Buf_Close(NOMSG)	Suppress "Saving..." and "Editing buffer..." messages.
BY	**Buf_Empty( )**	Empty the current edit buffer without closing it. Quit (abandon) any text or file in the buffer. Requests confirmation if the buffer has been altered. Same as **File_Quit( )**.
	Buf_Empty(OK)	Skip the confirmation prompt.
	Buf_Empty(NOEVENT)	Suppress executing the File-close event macros, even if enabled.
	Buf_Empty(EVENT)	Execute the File-close event macros, even if disabled.
BF	**Buf_Free**	Return the ID number of the next free (unused) edit buffer. (-1 if none available.)
	Buf_Free(EXTRA)	Return the ID number of the next free "extra" buffer in the range 100-125. (-1 if none available.)
BX	**Buf_Next**	Return the ID number of the next (open) edit buffer. (1 if only main buffer #1 is open.)
BN	**Buf_Num**	Return the ID number of the current buffer.
BNA	**Buf_Num_Altered**	Return the number of altered edit buffers (files) currently open.
BNW	**Buf_Num_Window**	Return the number of windows attached to the current edit buffer.

	Buf_Org_Filetype **Buf_Org_Window**	Return the file-type and window number (or custom values) saved for the current buffer. Used by `hexsplit.vdm`.
	Buf_Org_Filetype(*n*) **Buf_Org_Window(*n*)**	Save the file-type and window number (or custom values) in the current buffer.
BPV	**Buf_Previous**	Return the ID number of the previous (open) edit buffer. (1 if only main buffer #1 is open.)
BQ	**Buf_Quit( )**	Close the current edit buffer. Quit (abandon) any text or file in the buffer after requesting confirmation. Switches to one of the remaining buffers.
	Buf_Quit(OK)	Skip the confirmation prompt.
	Buf_Quit(DELETE)	Delete all windows attached to the buffer being closed.
	Buf_Quit(MAINBUF)	Switch to the main buffer "1" after closing the current buffer.
	Buf_Quit(ALL)	Close all buffers, except for the main buffer. Quit (abandon) all files.
BSTAT	**Buf_Status**	Return the status of the current edit buffer:
	Buf_Status(*r*)	Return the status of edit buffer '*r*': -1 - '*r*' is not open. 0 - '*r*' is open, but not attached to any window. w - '*r*' is open and attached to window '*w*'.
BS	**Buf_Switch(*r*)**	Switch to edit buffer '*r*', opening it if necessary.
	Buf_Switch(*r*,ATTACH)	Auto-create and attach a full-screen overlapping window to the buffer if necessary.
	Buf_Switch(*r*,EVENT)	Execute the Buffer-switch event macro.
	Buf_Switch(*r*,EXTRA)	When opening an "extra" buffer, allocate more memory if available.
	Buf_Switch(*r*,LOCAL)	Do not perform file buffering on the current edit buffer before switching to '*r*'.
	Buf_Switch(*r*,NOMSG)	Suppress the "Editing buffer..." message. (Usually suppressed automatically.)
	Buf_Switch(*r*,SUPPRESS)	Do not grab more memory when opening edit buffer '*r*'. (Default for extra buffers.)
BT	**Buf_Total**	Return the number of edit buffers currently open.
BW	**Buf_Window**	Return the ID number of the primary attached window. (0 if none.)
BWN	**Buf_Win_Next**	Return the ID number of the next (higher numbered) window attached to the current buffer.
BWP	**Buf_Win_Previous**	Return the ID number of the previous (lower numbered) window attached to the current buffer.
	Cab_Extract("*file.cab*")	(Win32 only) Extract all compressed files in the specified .CAB file into the current directory.
	Call(*r*)	Execute T-Reg '*r*' as a command macro. When finished, continue processing the following commands.
	Call(*r*+BUFFER)	Execute edit-buffer '*r*' as a command macro.

	Call(*r*,"*label*")	Begin execution at the label '*label*' instead of at the beginning of the register/buffer.
	Call("*label*")	Call the subroutine '*label*' in the current text register.
CALLF	**Call_File(*r*,*file*)**	Load the command macro in '*file*' into T-Reg '*r*' and execute it. When done, continue processing the following commands. Performs an extended search for '*file*'. Equivalent to **Reg_Load(*r*, "*file*",EXTRA) Call(*r*)**.
CLB	**Case_Lower_Block(*p*,*q*)**	Convert all letters to lower case in the block of text between file positions '*p*' and '*q*'. The edit position is not changed.
	CLB(*p*,*q*,NORESTORE)	The edit position is set just past the end of the block.
	CLB(*p*,*q*,RESET)	The Block_Begin and Block_End markers are cleared.
	CLB(*p*,*q*,COLUMN)	Convert a columnar block of text.
	CLB(*p*,*q*,COLSET,*c1*,*c2*)	Convert a columnar block of text.
	CLB(*l1*,*l2*,LINESET)	Convert a line-range block of text.
CSB	**Case_Switch_Block(*p*,*q*)**	Switch the case of all letters in the block of text.
CUB	**Case_Upper_Block(*p*,*q*)**	Convert all letters to upper case in the block of text.
	Chain(*r*)	Chain to the command macro in T-Reg '*r*' without "returning" to the current macro.
CHAINF	**Chain_File(*r*,"*file*")**	Load the command macro in '*file*' into T-Reg '*r*' and chain to it; '*r*' can be the currently executing register. Looks for '*file*' in the VEDIT Home Directory.
C	**Char(*m*)**	Move the edit position by '*m*' characters.
CD	**Char_Dump(*n*)**	Dump the character with ASCII value '*n*' to the console followed by a "newline".
	Char_Dump(NOCR)	Suppress the "newline" following the character. See also **Type_Char()**.
CMAT	**Chars_Matched**	Return the number of matching chars in the last **Compare( )**, **Match( )**, **Reg_Compare( )**, **Replace( )** or **Search( )** command.
	Chdir( )	Display the current drive/directory, same as **Name_Dir( ). See Name_Dir( ) for options.**
	Chdir("*d:path*")	Change the current directory to drive '*d*' and/or directory '*path*'.
CCB	**Clip_Copy_Block(*p*,*q*)**	(Win32 only) Copy the block of text between file positions '*p*' and '*q*' to the Windows clipboard. See **Reg_Copy_Block( )** for options.
	Clip_Ins( )	(Win32 only) Insert the Windows clipboard at the edit position. See **Reg_Ins( )** for options.
	Color_Highlight	Return an appropriate color for highlighting selection letters and key names; this value is set by **Config(C_HIGHLIGHT)**.
	Color_Prompt	Return an appropriate color for highlighting simple prompt lines, set by **Config(C_PROMPT)**.

CB	**Column_Begin**	Return the value of the block-begin marker's column. (0 if not set).
	Column_Begin(*n*)	Set the block-begin marker's column to '*n*'.
CE	**Column_End**	Return the value of the block-end marker's column. (0 if not set).
	Column_End(*n*)	Set the block-end marker's column to '*n*'.
CM	**Column_Mode**	Return the value of the Visual Mode "column-mode" setting.
	Column_Mode(SET)	Set "column-mode" for blocks in Visual Mode.
	Column_Mode(CLEAR)	Clear "column-mode" for blocks in Visual Mode.
	Compare(*r*) **Compare(*r*+BUFFER)**	Compare byte-by-byte the text at the edit position with T-Reg or edit buffer '*r*'. The comparison is not case sensitive. Advances the edit position(s) over as many characters as matched. Returns {0,1,2} corresponding to {=,>,<}.
	Compare(*r*,CASE)	The comparison is case sensitive.
CF	**Config( )**	Display the current value of all configuration parameters. Similar to **Config_Display( )**.
	Config(*x*)	Display current values of those configuration parameters beginning with '*x*_'.
	Config(*name*)	Return the value of configuration parameter '*name*'.
	Config(*name*,*n*)	Change configuration parameter '*name*' to value '*n*'. For edit-buffer dependent parameters, changes the current buffer's and "global" values. Returns the old parameter value.
	Config(*name*,*n*,ALL)	Change the parameter in all buffers and the "global" value.
	Config(*name*,*n*,LOCAL)	Only change the edit-buffer dependent parameter for the current buffer.
CFD	**Config_Display()**	Display all configuration parameters including strings and tab stops.
	Config_Display(EXTRA)	Display complete "Config(...)" commands with descriptive text for all parameters.
	Config_Display(EXTRA+SHORT)	Display complete "Config(...)" commands, but without the descriptive text.
	Config_Display(GLOBAL)	Display the current value of all configuration parameters. For edit-buffer dependent values, display the "global" value.
	Config_Display(LOCAL)	Only display the edit-buffer dependent values.
	Config_Display(SUPPRESS)	Suppress some configuration parameters (mostly hardware related) that should not be saved into the **`vedit.cfg`** file.
CFL	**Config_Load("*file*")**	Load new configuration parameters from the file '*file*'. If not found in the current directory, it will look in the *User Config Directory*.
	CFL("*file*",NOERR)	Suppress error message if the file is not found.

CFSAV	**Config_Save("*file*")**	Save entire configuration to the file '*file*'.
	Config_Save("*file*",OK)	Skip confirmation prompt when '*file*' already exists.
CFS	**Config_String( )**	Display the current value of all configuration strings.
	Config_String(*name*,"*text*")	Change configuration string '*name*' to "*text*".
CFT	**Config_Tab( )**	Display tab stops in current edit buffer.
	Config_Tab(*n*)	Set uniform tab stops of every '*n*' column. Sets the current edit buffer's and "global" values.
	Config_Tab(*n1*,*n2*,...)	Set tab stops at the selected columns.
	Config_Tab(*n*;ALL)	Change the tab stops for all edit buffers and the "global" values. Note the ";" (semicolon).
	Config_Tab(*n*;LOCAL)	Only change the tab stops for the current edit buffer. Note the ";" (semicolon).
CFV	**Config_Vedit("*file.exe*")**	(DOS) Save the entire configuration into the VEDIT.EXE file named '*file*'.
	CFV("*file*",KEYBOARD)	Also saves the keyboard layout into VEDIT.
	Continue	Skip the current iteration of any **While**, **Do-while**, **For** or **Repeat** loop, causing the loop to be re-tested.
CC	**Cur_Char**	Return the value of the character at the edit position. At the End-Of-File, return 26 (<Ctrl-Z>).
	Cur_Char(*m*)	Return the value of the '*m'th* next/previous character.
CN	**Cur_Col**	Return the horizontal column number for the character at the edit position.
CL	**Cur_Line**	Return the line number in the file for the edit position.
CP	**Cur_Pos**	Return the edit position (offset) in the file. The Beginning-Of-File is position 0 (zero).
CRN	**Cursor_Col**	Return the column number of the cursor position in Visual Mode. Same as the "COL:" display.
	Date( )	Display the current system date as mm-dd-yyyy.
	Date(BEGIN)	Display the date as dd-mm-yyyy.
	Date(VALUE,'/')	Display the date as mm/dd/yyyy.
	Date(NOCR)	Omit the following CR+LF newline.
	Date(NOMSG)	Omit the heading "Date:".
DB	**Del_Block(*p,q*)**	Delete the block of text between file positions '*p*' and '*q*'.
	Del_Block(*p,q*,COLUMN)	Delete a columnar block of text.
	Del_Block(*p,q*,COLSET,*c1,c2*)	Delete a columnar block of text.
	Del_Block(*l1*,*l2*,LINESET)	Delete a line-range block of text.
DC	**Del_Char(*m*)**	Delete '*m*' characters from the text.
DL	**Del_Line(*m*)**	Delete '*m*' lines of text, starting at the edit position.

	Del_Line(0)	Delete characters from the beginning of the line up to the edit position.
DTH **DTW**	**Desktop_Height** **Desktop_Width**	(Windows) Return the height and width of the desktop (screen)in pixels, e.g. 600x800 or 768x1024.
DTAB	**Detab_Block(*p,q*)**	Convert tabs to spaces in the block of text between file positions '*p*' and '*q*'.
	DTAB(*p,q*,NORESTORE)	The edit position is set just past the end of the converted block.
	DTAB(*p,q*,COLUMN)	Detab a columnar block.
	DTAB(*p,q*,COLSET,*c1,c2*)	Detab a columnar block.
	DTAB(*l1,l2*,LINESET)	Detab a line-range block.
DI1	**Dialog_Input_1(*r*,"...")**	(Windows) Create a dialog box with a title, text, buttons, and optional check boxes, radio buttons and input strings. See the on-line help for the latest details.
DIR	**Directory("*fspec*")**	Display the disk directory. Optional drive, path, and wildcard characters "**?**" and "*****" may be specified.
	Dir("*fspec* -s")	Include filenames in any subdirectories
	Dir("*fspec*",COUNT,*n*)	Display the filenames in '*n*' columns instead of 4.
	Dir("*fspec*",NOERR)	Suppress error message if '*fspec*' does not match any files; set **Error_Flag** to "2". Also suppress error message if the pathname contains a non-existent directory; set **Error_Flag** to "3".
	Dir("*fspec*",NOMSG)	Omit the "Directory:" and pathname header line and display the filenames in 1 column.
	Dir("*fspec*",SHORT)	Display long filenames in the short 8.3 format.
	Dir("*fspec*",SUPPRESS)	Omit subdirectory names and system/hidden files.
DKF	**Disk_Free("*drive*")**	Return the amount of free disk space on drive '*drive*' in Megabytes.
DKI	**Disk_Info("*drive*")**	(DOS) Display detailed disk information about drive '*drive*'.
DKO	**Disk_Open("*drive*")**	(DOS) Open disk drive '*drive*' for sector editing.
DKOD	**Disk_Open_DOS("*drive*")**	(DOS) Open DOS disk (partition) drive '*drive*' for sector editing.
DKS	**Disk_Size("*drive*")**	Return the total size of drive '*drive*' in Megabytes.
DOV	**Do_Visual("*vm**vm**xx*")**	Execute the Visual Mode functions '*vm-codes*' and remain in Command Mode. Appendix A lists the '*vm-codes*' which are enclosed by "\ \". Normal displayable characters '*xx*' are not enclosed by "\ \".
EOF	**End_Of_File( )**	Move past the last character in the file.
	End_Of_File(LOCAL)	Move past the end of the portion of the file currently in memory.
EOL	**End_Of_Line( )**	Move to the end of the current line.
	EOL_Pos	Return the file position at the end of the current line, just before the newline character(s).

EF	**Error_Flag**	Return the value of error flag which is reset/set by each command.
EM	**Error_Match**	Return the value of search/match error flag which is reset/set only by **Compare( )**, **Match( )**, **Search( )**, **Replace( )** and **Reg_Compare( )**
	Error_OS	Return the value of the write error flag which is reset/set by the last disk write operation.
	Escape_Mode	Return the current value of the alternate [ESCAPE] function. (0 if none).
	Escape_Mode(*vm-code*)	Set the [ESCAPE] function to perform the specified Visual Mode function instead of popping up the **{ESCAPE}** menu. Codes are listed in Appendix A.
	Exit	Exit VEDIT after prompting whether each modified file is to be saved or abandoned. If a file to be saved has no assigned filename, prompts for one.
	Exit(*n*)	(DOS/QNX) Return '*n*' to the OS as VEDIT's "return code" instead of the default value of zero.
XBUF1 **XBUF2** **XBUF3** **XBUF4**	**Extra_Buffer_1** **Extra_Buffer_2** **Extra_Buffer_3** **Extra_Buffer_4**	Returns the ID # of the first four "Extra" edit buffers. This is 100-103 for the Windows version and 33-36 for the DOS version.
FA	**File_Attrib("*file*")**	Return the attributes (read-only, hidden, etc.) of '*file*'. Return -1 if '*file*' does not exist.
	File_Attrib("*file*",*n*)	Set the attributes of '*file*' to '*n*'. Returns the old attributes.
	File_Attrib("*file*",*n*,RESET)	Reset (clear) the specified attribute bit(s) '*n*'.
	File_Attrib("*file*",*n*,SET)	Set the specified attribute bit(s) '*n*'.
FCHK	**File_Check("*file*")**	Check if '*file*' is currently being edited in any buffer. If it is, returns the buffer number 1 - 99; if not, returns -1.
FC	**File_Close( )**	Save and close the current file. If the current buffer contains text but has no assigned filename, prompts for one. Remains in the current edit buffer.
	File_Close(CONFIRM)	Prompts for confirmation to save or abandon the file if it has been altered.
	File_Close(EVENT)	Execute the File-close event macros, even if disabled.
	File_Close(NOEVENT)	Suppresses executing the File-close event macros, even if enabled.
	File_Close(NOMSG)	Suppresses the "Saving:..." message.
	File_Close(OK)	Suppress the "Save changes to disk..." prompt during disk sector editing. (DOS only)
FCP	**File_Copy("sfile","dfile")**	Copy the file '*sfile*' to '*dfile*'.
	File_Copy(...,OK)	Skip confirmation to overwrite an existing '*dfile*'.
	File_Copy(...,NOERR)	Suppress error message if '*sfile*' does not exist.
FDEL	**File_Delete("*fspec*")**	Delete the file(s) specified by '*fspec*' from disk. Optional drive, path, and wildcard characters "?" and "*".

	File_Delete("*fspec*",OK)	Delete the specified file, skipping the directory display and confirmation prompt.
	FDEL("*fspec*",OK+EXTRA)	The options OK+EXTRA are needed to delete, without confirmation, multiple files specified with "*".
	FDEL("*fspec*",NOERR)	Suppress error message if the file(s) is not found.
FEXIST	**File_Exist("*fspec*")**	Test for existence of the file(s) '*fspec*'. Returns the number of matching filenames (0 if none).
	File_Exist("*fspec*",NOERR)	Suppress the error message if the pathname contains a non-existent directory; sets **Error_Flag** = 3.
	File_Exist("*fspec*",SUPPRESS)	Omit directory names and system/hidden files.
FMD	**File_Mkdir("*dir*")**	Make (create) the new directory '*dir*'.
FMV	**File_Move("oldfile","newfile")**	Move the file '*oldfile*' to '*newfile*'. Identical to File_Rename( ).
FO	**File_Open("*file*")**	Open (or create) the file '*file*' for editing. It is opened in the first available buffer. If the file is already open in another buffer, it switches to that buffer.
	FO('"*file1*" -a "*file2*"')	Open '*file1*' for editing, but save it as '*file2*'. Note how single and double quotes must be used to support long filenames with embedded spaces.
	FO('"*file*" -l *n*')	Open '*file*' and set the initial edit position on line '*n*'.
	FO('"*file1*","*file2*", ...')	Open multiple files. The first file is opened in the normal manner; the additional files are opened in additional edit buffers. Each file may include the "**-a**", "**-l**" and "**-t**" options.
	FO("*file*?.*")	The wildcards "?" and "*" may be used to specify the files being opened. All matching files will be opened.
	FO("*file*",ATTACH)	Create a new overlapping window for each opened file. The windows are only created if **{CONFIG, Display options, Auto-create window style}** is enabled (default).
	FO("*file*",BROWSE)	Open the file in "browse-only mode"; it cannot be altered.
	FO("*file*",CHGDIR)	Change the current (default) directory to the directory containing '*file*'.
	FO("*file*",EVENT)	Execute the File-open event macro, even if disabled.
	FO("*file*",FORCE)	Allow a file to be opened in an "extra" buffer. Note that files opened in "extra" buffers are not automatically saved.
	FO("*file*",MRU)	(Windows only) Add the specified file(s) to the Most-Recently-Used list in the **{FILE}** menu and File-selector when the file is closed.
	FO("*file*",NOEVENT)	Suppress executing the File-open event macro, even if enabled.
	FO("*file*",NOMSG)	Suppress the "New File" message if the file is created.

	FO("*file*",OVERWRITE)	Suppress creating a backup file, even if backup files are enabled.
FOPENR	**File_Open_Read("*file*")**	Open the file '*file*' for input (reading).
FOPENW	**File_Open_Write("*file*")**	Open the file '*file*' for output (writing).
FQ	**File_Quit( )**	Quit (abandon) any text or file in the buffer; remain in the same buffer. Requests confirmation if the buffer has been altered. Same as Buf_Empty( ).
	File_Quit(OK)	Skip the confirmation prompt.
FR	**File_Read(*n*)**	Append '*n*' lines from the input file to the edit buffer.
	File_Read(0)	Append lines until edit buffer is nearly full.
	File_Read(*n*,REVERSE)	Read back '*n*' lines from the output file.
	File_Read(0,REVERSE)	Read back lines until edit buffer is nearly full.
FREN	**File_Rename("oldfile","newfile")**	Rename or move the file '*oldfile*' to '*newfile*'. Identical to File_Move( ).
	File_Rename(...,OK)	Skip confirmation to overwrite existing '*newfile*'.
	File_Rename(...,NOERR)	Suppress error message if '*oldfile*' does not exist.
FRD	**File_Rmdir("*dir*")**	Remove (delete) the directory '*dir*'; it must be an empty directory with no files.
FS	**File_Save**	If modified, save the file in the current edit-buffer to disk and keep it open for further editing.
	File_Save(ALL)	Save all modified files to disk.
	File_Save(NOMSG)	Save file(s), but suppress "Saving..." message.
	File_Save(BEGIN)	Save file(s), but leave the edit position at the beginning of the file. Useful when editing huge files.
FSA	**File_Save_As("*file*")**	Save the current file under the new name '*file*' and allow further editing under this new filename.
	File_Save_As("*file*",OK)	Suppress the confirmation prompt if '*file*' already exists.
FSIZE	**File_Size**	Return the size of the current file in bytes.
FTRUNC	**File_Truncate( )**	Truncate and close the current output file WITHOUT writing additional text. Rarely used — primarily after a **File_Write( )** command. CAUTION. INCORRECT USE WILL RESULT IN DELETED FILES!
	File_Truncate(OK)	Skip the confirmation prompt.
FW	**File_Write(*n*)**	Write '*n*' lines to disk from the beginning of the buffer.
	File_Write(0)	Write out the edit buffer up to the current line.
	File_Write(*n*,REVERSE)	Write last '*n*' lines in the edit buffer to the ".rR$" file.
	File_Write(0,REVERSE)	Write the current line and the rest of the edit buffer to the ".rR$" file.
	Font_Charset	Return the character set for the current display font. 0=ANSI, 255=OEM. The DOS version always returns 255.

FONTH **FONTW**	**Font_Height** **Font_Width**	(Windows) Return the height and width of the current display font in pixels. This includes any additional "LineSpacing" set in the **`vedit.ini`** file.
FP	**Format_Para(*n*)**	Format the current paragraph and advance the edit position to the next paragraph. The current left margin is used unless '*n*' is specified as the temporary left margin. Justify with an even right edge if **{CONFIG, Word processing, Justify paragraphs}** is set.
GE	**Get_Environment(*r*,"*name*")**	Set T-Reg '*r*' to the (string) value of environment variable '*name*'.
GF	**Get_Filename(*r*,"*fspec*")**	Selects a filename using a dialog box. '*fspec*' is the initial "filter". When selected, the complete pathname is placed into T-Reg '*r*'.
GI	**Get_Input(*r*,"*mtext*")**	Prompt for user input with '*mtext*'. Get the entire keyboard input line including the terminating "newline" and save it in T-Reg '*r*'.
	GI(*r*,"*mtext*",NOCR)	Get keyboard input line without the "newline" character(s).
	GI(*r*,"*mtext*",TAB8)	Any tabs in '*mtext*' use tab stops at every 8 columns.
	GI(*r*,"*mtext*",STATLINE)	Prompt with '*mtext*' on the status line.
	GI(*r*,"*mtext*",APPEND)	Append input line to any contents of T-Reg '*r*'.
	GI(*r*,"*mtext*",INSERT)	Insert the input line at the beginning of T-Reg '*r*'.
	GI(*r*,"*mtext*",COUNT,*n*)	Limit the length of the input line to '*n*' characters.
	GI(*r*,"*mtext*","*default*")	Set a default input line of '*default*'.
GK	**Get_Key("*mtext*")**	Prompt for user input with '*mtext*'. Return the value of the next key pressed. Normal characters return value 32 - 255. Function/control keys return their decoded function code.
	Get_Key("*mtext*",RAW)	Control characters are not decoded and return 0 - 31. Function keys are not decoded and return their "scan code" * 256.
	GK("*mtext*",NOCANCEL)	Allow **[CANCEL]** (**<Ctrl-C>**) as valid function key. Otherwise pressing [CANCEL] stops any macro.
	GK("*mtext*",TAB8)	Any tabs in '*mtext*' use tab stops at every 8 columns.
	GK("*mtext*",STATLINE)	Prompt with '*mtext*' on the status line.
GN	**Get_Num("*mtext*")**	Prompt for user input with '*mtext*'. Return the value of the numeric expression that is entered. (0 if invalid).
	GN("*mtext*",SUPPRESS)	Return the value of the simple decimal number.
	GN("*mtext*",TAB8)	Any tabs in '*mtext*' use tab stops at every 8 columns.
	GN("*mtext*",STATLINE)	Prompt with '*mtext*' on the status line.
	GN("*mtext*","*default*")	Use '*default*' as the default input string.
	Goto *label*	Jumps to the label "*label*:" or ":*label*:".
GC	**Goto_Col(*n*)**	Move the edit position as close as possible to column '*n*' on the current line.

	Goto_Col(*n*,EXTRA)	Also force the Visual Mode cursor to column '*n*'. Should immediately be followed by **Visual( )**.
GL	**Goto_Line(*n*)**	Go to the beginning of line '*n*' in the current file.
GM	**Goto_Marker(*m*)**	Goto previously set text marker '*m*'.
GP	**Goto_Pos(*n*)**	Go to file position '*n*' in the current file. **Goto_Pos(0)** goes to the beginning-of-file.
H	**Help**	Start up on-line help for Command Mode.
	Help("*text*")	Immediately search help file for the topic '*text*'.
IC	**Ins_Char(*n*)**	Insert the character with decimal value '*n*' into the edit buffer and advance the edit position.
	Ins_Char(*n*,OVERWRITE)	Overwrite the existing character and advance.
	Ins_Char(*n*,COUNT,*n2*)	Repeat the insertion '*n2*' times.
INSF	**Ins_File("*file*")**	Insert '*file*' into the edit buffer and advance the edit position.
	Ins_File("*file*",BEGIN)	Leave the edit position at the beginning of the inserted file.
	Ins_File("*file*",*n1*,*n2*)	Insert the line range '*n1*' - '*n2*' of '*file*' into the edit buffer.
	Ins_File("*file*",1,ALL,COLUMN)	Insert the file as a columnar block.
II	**Ins_Indent(*n*)**	Insert the optimum number of tabs and spaces to reach column '*n*'.
IM	**Insert_Mode**	Return the value of the Visual Mode "Insert" mode setting.
	Insert_Mode(SET)	Set "Insert" mode in Visual Mode. Same as **Insert_Mode(1)**.
	Insert_Mode(CLEAR)	Clear "Insert" mode; set "overstrike" mode. Same as **Insert_Mode(0)**.
IN	**Ins_Newline(*n*)**	Insert '*n*' newlines, according to the current file type, into the edit buffer and advance the edit position.
IT	**Ins_Text("*text*")**	Insert '*text*' into the edit buffer.
	IT("*text*",OVERWRITE)	Overwrite existing characters with '*text*'.
	Ins_Text("*text*",COUNT,*n*)	Insert '*n*' copies of '*text*'.
IA	**Is_Altered**	Return TRUE if the current edit buffer has been altered since the last file save.
IAF	**Is_Altered_File**	Return TRUE if the current file (buffer) has been altered since it was opened. Saving the file does not change this flag.
IAE	**Is_Auto_Execution**	Return TRUE if a "**-x**" invocation macro is currently running.
IDKO	**Is_Disk_Open**	Return TRUE if the current edit buffer is being used for disk sector editing.
	Is_Expired	Return TRUE if the support period for the VEDIT product is expired, of if VEDIT is running as a trial version and the trial period is expired.

	Is_File_Selector	(Win32) Return TRUE if the File selector windows is currently displayed.
ILF	**Is_Long_Filename**	Return TRUE if long filenames are being used with Windows. Long filename support can be turned off with the "**-d**" invocation option.
	Is_Mono	Return TRUE if VEDIT is using its monochrome screen attributes (colors).
INF	**Is_New_File**	Return TRUE if the current file was created, i.e. it didn't exist when it was opened.
IOR	**Is_Open_Read**	Return TRUE if the current buffer has an open input file
IOW	**Is_Open_Write**	Return TRUE if current buffer has an open output file.
IO	**Is_Option(*x*)**	Return TRUE if invocation option '*x*' was specified.
	Is_OS2	Return TRUE if VEDIT is currently running under the OS/2 operating system.
	Is_Quiet	Return "1" if VEDIT was invoked with the "**-q**" invocation option. Return "2" if the Windows version is currently minimized.
ISRI	**Is_Redirect_Input**	Return TRUE if "keyboard" input to the macro language is currently being redirected from a file.
	Is_Saveas	Return TRUE if the output file is different from the input file.
ISV	**Is_Startup_Vdm**	Return TRUE if the **`startup.vdm`** macro was loaded and executed at startup.
	Is_Support	Return the product support expiration date; the year is returned in the lower four hex digits, the month in the next two hex digits.
	Is_VEDIT_PLUS	Return TRUE for VEDIT (PLUS), FALSE for VEDIT Lite.
	Is_VSWAP	(DOS) Return TRUE if V-SWAP is installed and enabled.
	Is_Windows	Return Microsoft Windows version #. Win95 = 395 or 400, Win98 = 410; Win2000 = 500; WinXP = 501.
	Is_Win32_Version	Return TRUE if the 32-bit Windows version of VEDIT is running.
	Is_WinNT	Return TRUE if the 32-bit Windows version of VEDIT is running under Windows NT/2000/XP.
	Itoa(*n*,*r*)	Place the ASCII conversion of numeric expression '*n*' into T-Reg '*r*'. Same as **Num_Str( )**.
KA	**Key_Add("*key-seq*","*edit-seq*")**	Add the key assignment to the end of the keyboard layout table.
	Key_Add(...,NOCONFIRM)	Skip confirmation to overwrite any existing assignment to '*key-seq*'.
	Key_Add(...,INSERT)	Insert the key assignment at the beginning of the keyboard layout table.
	Key_Cfg_Pop()	Restore the previous keyboard layout by "popping" it from the text register stack.

	Key_Cfg_Push()	Save the current keyboard layout by "pushing" it on the text register stack.
KD	**Key_Delete("*key-seq*")**	Delete the keyboard assignment to '*key-seq*'.
	Key_Delete("*key-seq*",NOERR)	Suppress error if '*key-seq*' not assigned.
	Key_Delete("*edit-seq*",REVERSE)	Delete the keyboard layout entry which assigns any key to the specified "edit sequence".
	Key_Jam(*n*)	(DOS only) Jam the keyboard character or scan-code 'n' into the hardware keyboard buffer.
	Key_List()	List (display) the entire keyboard layout.
KL	**Key_Load("*file*")**	Load a new keyboard layout from the file '*file*'. If the file is not found in the current directory, the *VEDIT home directory* will be searched.
	Key_Load("*file*",NOERR)	Suppress error message if the file is not found.
	Key_Pop(*n*)	Pop (remove) the first '*n*' key assignments from the beginning of the keyboard layout table. USE WITH CAUTION! See **Key_Add(...,INSERT)**.
	Key_Purge( )	Purge any pending keystrokes, keystroke macro, **[REPEAT]** or **[REPEAT LAST]** function.
KRM	**Key_Record_Mode**	Return the value of the [REPEAT LAST] record mode.
	Key_Record_Mode(*n*)	Set the [REPEAT LAST] record mode to '*n*': 0 - Keystroke recording completely off. 1 - Normal keystroke recording for Visual Mode. 2 - All keystrokes are recorded until **KRM(0)**.
KS	**Key_Save("*file*")**	Save current keyboard layout, including any keystroke macros, to the file '*file*'.
	Key_Save("*file*",BINARY)	Save the keyboard layout in a "binary" format.
	Key_Save("*file*",OK)	Skip confirmation prompt when '*file*' already exists.
KSTAT	**Key_Status**	Return TRUE if a key (keystroke macro) is pending.
KT	**Key_Total**	Return the total number of keyboard characters typed since starting VEDIT. (Windows version does not count dialog boxes.)
LSP	**Last_Search_Pos**	Return the edit (file) position before the last search.
L	**Line(*m*)**	Move the edit position by '*m*' lines.
	Line(0)	Move to the beginning of the current line.
	Line(*m*,NOERR)	Suppress error message if B-O-F or E-O-F reached.
	Line(*m*,ERRBREAK)	Perform **Break** if B-O-F or E-O-F reached.
MN	**Macro_Num**	Return the ID number of the T-Reg/Buffer currently executing as a command macro.
ML	**Margin_Left**	Return the value of the left margin.
	Margin_Left(*n*)	Set the left margin (indent column) for the current edit buffer to '*n*'.
MR	**Margin_Right**	Return the value of the right margin.

	Margin_Right(*n*)	Set the right margin (word wrap column) for the current edit buffer to '*n*'.
	Marker(*m*)	Return the file position of text marker '*m*'. (-1 if not set).
MW	**Mark_Word( )**	Mark the word at the edit position as a highlighted block, setting **Block_Begin** and **Block_End**.
	Match("*ss*")	Compare the text at the edit position with '*ss*' and return the results of the comparison: {0,1,2} for {=, >, <}. '*ss*' may contain pattern matching codes.
	Match("*ss*",CASE)	Perform a case sensitive match, otherwise the match is case insensitive.
	Match("*ss*",WORD)	The matched text must be a distinct "word", i.e. surrounded by separators (non-alphanumeric).
	Match("*ss*",REGEXP)	Match using regular expressions with "minimized" matching.
	Match("*ss*",REGEXP+MAX)	Match using regular expressions with "maximized" matching.
	Match("*ss*",ADVANCE)	Advance the edit position past the matched characters if successful.
	Match("*ss*",ALL)	Match as many consecutive occurrences of '*ss*' as possible. Successful if at least one occurrence matched.
	Match("*ss*",COUNT,*n*)	Must match '*n*' times to be successful.
MI	**Match_Item**	Return the item number that matched in a search/match using the "**|{...}**" pattern matching code.
MP	**Match_Paren( )**	If the edit position is at one of the parentheses pairs "{ } [ ] < > ( )", find its matching pair; otherwise, search forwards for the next parentheses character.
	Max(*n*,*m*)	Return the greater of the two numeric values '*n*' and '*m*'. See **Min( )**.
MF	**Mem_Free**	Return number of bytes free in the current edit buffer.
	Mem_Free(*n*)	Free '*n*' bytes of memory space in current edit buffer, if possible, by buffering the file back to disk. (Rarely used.)
	Mem_Free(1)	Squeeze the edit buffer to approximately 8K in size.
MSTAT	**Mem_Status( )**	Display number of free bytes in current edit-buffer; number of bytes used in the edit buffer; total number of bytes in all text registers.
M	**Message("*mtext*")**	Type (display) '*mtext*'. Multiple lines may be typed by using "**\n**" or by entering '*mtext*' as multiple lines.
	Message("*mtext*",TAB8)	Any tabs in '*mtext*' use tab stops at every 8 columns.
	Message("*mtext*",STATLINE)	Display the one-line message on the status line. See also **Statline_Message( )**.
	Message("...\<*vm*>...",EXTRA)	The key assigned to edit-function '*vm*' is displayed highlighted. Appendix A lists the '*vm codes*'.

	Min(*n*,*m*)	Return the lesser of the two numeric values '*n*' and '*m*'. See **Max( )**.
	Mouse_Active	Return non-zero (TRUE) if the mouse is active.
ND	**Name_Dir( )**	Display current drive/directory.
	Name_Dir(NOMSG)	Omit the "Directory:" header.
	Name_Dir(NOCR)	Omit the following CR+LF newline.
NF	**Name_File( )**	Display names of the input and output files.
	Name_File(EXTRA)	Display filenames with complete drive and path.
NR	**Name_Read( )**	Display the name of the input (read) file.
	Name_Read(EXTRA)	Display filename with its drive and path.
	Name_Read(NOMSG)	Omit the "Input file:" header.
	Name_Read(NOCR)	Omit the following CR+LF newline.
NW	**Name_Write( )**	Display the name of the output (write) file. See **Name_Read( )** for options.
NC	**Newline_Chars**	Return the number of chars in a "newline" according to the current file type. (Windows/DOS=2; Unix/Mac=1; Record/binary=0).
NTS	**Next_Tab_Stop**	Return the column position of the next tab stop based on the current value of **Cur_Col**.
	N_Option	Return the value of the number following the "**-N**" invocation option.
	N_Option(*n*)	Force the value returned by **N_Option** to '*n*.
NAB	**Num_All_Bufs**	Return the maximum number of all edit buffers in VEDIT including "extra" buffers. It is 125 for the Windows version and 36 for the DOS version.
NEB	**Num_Edit_Bufs**	Return the maximum number of normal edit buffers in VEDIT.(Windows = 99; DOS = 32)
NDS	**Num_Display(*x*,*y*)**	Display values of non-zero numeric registers '*x*' - '*y*'.
	Num_Display(*x*,*y*,ALL)	Display values even if zero.
	Num_Display(*x*,*y*,NOMSG)	Display values even if zero and omit the "#xx =" header.
	Num_Display(*x*,*y*,COUNT,*n*)	Display values in '*n*' columns.
NE	**Num_Eval( )**	Evaluate the numeric expression in the text at the edit position and return its value.
	Num_Eval(ADVANCE)	Advance the edit position past the numeric expression.
	Num_Eval(SUPPRESS)	Only a simple number is evaluated.
NED	**Num_Eval_Date()**	(Windows) Evaluates a date mm/dd/yyyy or dd-mm-yyyy as the number of days since 01-01-0001 assuming the Julian calendar and leap years.
NER	**Num_Eval_Reg(*r*)**	Evaluate the numeric expression in T-Reg '*r*' and return its value. **Atoi( )** is another name for **Num_Eval_Reg( )**.

NI	**Num_Ins(*n*)**	Insert the ASCII value of numeric expression '*n*' into the edit buffer and advance the edit position.
	Note:	**Num_Ins( )** has the same options as **Num_Type( )**.
NID	**Num_Ins_Date(*n*)**	(Windows) Inserts '*n*' as a date, e.g. mm-dd-yyyy, where '*n*' is the number of days since 01-01-0001 assuming the Julian calendar and leap years.
	Num_Ins_Date(*n*,BEGIN)	Insert the date as dd-mm-yyyy.
	Num_Ins_Date(*n*,NOCR)	Suppress the newline (CR+LF) after the date.
	Num_Ins_Date(*n*,VALUE,'/')	Insert the date as mm/dd/yyyy.
	Num_Pop(*x*,*y*)	Pop (restore) numeric registers '*x*' through '*y*' from the register stack.
	Num_Push	Return the number of numeric registers which have been pushed on the stack with **Num_Push( )**.
	Num_Push(*x*,*y*)	Push (save) numeric registers '*x*' through '*y*' onto the numeric register stack; the registers are not changed. Maximum of 254 registers. (DOS: max of 128)
NS	**Num_Str(*n*,*r*)**	Place the ASCII value of numeric expression '*n*' into T-Reg '*r*'. **Itoa(**) is another name for **Num_Str( )**.
	Num_Str(*n*,*r*,APPEND)	Append the ASCII value to the existing contents of T-Reg '*r*'.
	Num_Str(*n*,*r*,INSERT)	Insert the ASCII value at the beginning of T-Reg '*r*'.
NT	**Num_Type(*n*)**	Type (display) the value of numeric expression '*n*', right justified followed by a "newline".
	Num_Type(*n*,LEFT)	Display the number left justified.
	Num_Type(*n*,NOCR)	Suppress the "newline" following the number.
	Num_Type(*n*,FILL)	Use "0" for any padding, instead of spaces.
	Num_Type(*n*,EXTRA)	Use extra padding for positive numbers to have the same width (6 or 11 columns) as negative numbers.
	Num_Type(*n*,FORCE)	Use a field width of 10 columns for all positive numbers; negative numbers use a width of 11 columns.
	Num_Type(*n*,FORCE+EXTRA)	Use a field width of 11 columns for all positive and negative numbers.
	Num_Type(*n*,HEX)	Display the number in hex, left justifed, in the format "0Xhhhh:hhhh" with as many hex digits as needed.
	Num_Type(n,HEX+NOMSG)	Display the number in hexin the format "hhhhhhhh" with as many 'hh' hex digits as needed; the "0X" and ":" are suppressed.
	O_Option	Return the value of the number following the "**-o**" invocation option.
OS	**OS_Type**	Return the operating system type: (1=Windows, 2=DOS, 4=UNIX/XENIX, 5=QNX, 6=Linux)
OF	**Out_File("*file*")**	Re-route console output to the file '*file*'.
	Out_File(CLEAR)	Disable **Out_File( )**, close the re-routed file, allowing output to go back to the console.

OI	**Out_Ins( )**	Re-route following console output into the edit buffer and advance the edit position.
	Out_Ins(CLEAR)	Disable **Out_Ins( )**, allowing output to go back to the console. Same as **Out_Ins(0)**.
	Out_OS( )	(DOS) Re-route console output directly to DOS. This bypasses the normal screen/window handler and allows(e.g. ANSI escape sequences) to be sent to DOS.
	Out_OS(CLEAR)	Disable **Out_OS( )**
OP	**Out_Print( )**	Re-route console output to the printer.
	Out_Print(CLEAR)	Disable **Out_Print( )**.
OR	**Out_Reg(*r*)**	Re-route console output to T-Reg '*r*'.
	Out_Reg(CLEAR)	Disable **Out_Reg( )**.
OM	**Overwrite_Mode**	Return the value of "Overwrite mode" for the current edit buffer.
	Overwrite_Mode(*n*)	Set the overwrite mode for the current buffer to '*n*'. Valid values are 0, 1 and 2.
PK	**Previous_Key(*n*)**	Return the '*n*'th previous keystroke. Normal characters return 32 - 255. Function/control keys return function-codes with value > 255.
	Previous_Key(*n*,RAW)	Return the '*n*'th previous raw keystroke. Normal characters return 32 - 255. Control keys return 0 - 31. Function keys return their hardware scan-code * 256.
PTS	**Previous_Tab_Stop**	Return the column of the previous tab stop; 0 if none.
PR	**Print(*m*)**	Print the next/previous '*m*' lines of text. Control characters are printed according to the current Print mode.
	Print(*m*,RAW)	Print in "Raw" mode without margins and print all control characters as-is, without conversion.
	Print(*m*,NOEVENT)	Suppress the print-job start string, even if enabled.
	Print(*m*,EVENT)	Send the print-job start string, even if disabled.
PB	**Print_Block(*p*,*q*)**	Print the block of text between file positions '*p*' and '*q*'. See Print( ) and Type_Block( ) for other options.
PE	**Print_Eject( )**	Page eject - advance printer to next page, typically with a Form-Feed character.
	Print_Eject(0)	Reset only the internal line counter used for printing.
PF	**Print_Finish( )**	Finish printing; typically send a page eject to the printer. Optionally send the "Printer Finish string". Then close the print-job.
	Print_Finish(SUPPRESS)	Suppress the page eject, even if enabled.
	Print_Finish(NOEVENT)	Suppress the print-job finish (reset) string, even if enabled.
	Print_Finish(EVENT)	Send the print-job finish string, even if disabled.
PCPL	**Printer_CPL**	(Windows only) Return the printer's "Characters per line", depending upon the printer font and size.

PLN	**Printer_Line**	Return the line number on the printed page that the next **Print( )** command will print to, e.g. 1 - 66.
PLPP	**Printer_LPP**	Return the printer's "Lines per page".
	Process_ID	Return the "Process ID" number assigned by the OS to the current instance of VEDIT.
	Qall	Quit (abandon) all files and exit VEDIT. Requests confirmation.
	Qally	Skip the confirmation prompt. Use with extreme care. Same as **Qall(OK)**. (BAD HABIT TO GET INTO!!)
	Qall(*n*)	Return '*n*' to the OS as VEDIT's "return code" instead of the default value of zero.
RINP	**Redirect_Input("*file*")**	Use '*file*' as "keyboard" input redirection for the **Get_Input( )**, **Get_Key( )**, **Get_Num( )**, etc. commands.
	Redirect_Input(CLEAR)	Disable input redirection, allowing input to come from the keyboard.
RCOMP	**Reg_Compare(*r*,"*text*")**	Compare byte-by-byte the contents of T-Reg '*r*' with '*text*'. The comparison is not case sensitive. Returns {0,1,2} corresponding to {=,>, <}.
	Reg_Compare(*r*,"*text*",CASE)	The comparison is case sensitive.
RC	**Reg_Copy(*r*,*m*)**	Copy next/previous '*m*' lines from the edit buffer into T-Reg '*r*'. The edit position is unchanged.
	Reg_Copy(*r*,*m*,APPEND)	Append to any existing contents in T-Reg '*r*'.
	Reg_Copy(*r*,*m*,INSERT)	Insert at the beginning of T-Reg '*r*'.
	Reg_Copy(*r*,*m*,NORESTORE)	The edit position is set just past the end of the copied text.
RCB	**Reg_Copy_Block(*r*,*p*,*q*)**	Copy the block of text between file positions '*p*' and '*q*' into T-Reg '*r*'.
	RCB(*r*,*p*,*q*,DELETE)	Copy (move) the block of text into T-Reg '*r*' and delete it from the edit buffer.
	RCB(*r*,*p*,*q*,DELETE+FILL)	Copy (move) the block of text into T-Reg '*r*' and then replace (fill) the source block with spaces.
	RCB(*r*,*p*,*q*,RESET)	Clear the Block_Begin and Block_End markers.
	RCB(*r*,*p*,*q*,CLIPBOARD)	Copy the block of text into T-Reg '*r*' and then copy '*r*' to the Windows clipboard.
	RCB(*r*,*p*,*q*,COLUMN)	Copy a columnar block of text. File positions '*p*' and '*q*' define the "corners" of the columnar block.
	RCB(*r*,*p*,*q*,COLSET,*c1*,*c2*)	Copy a columnar block of text. File positions '*p*' and '*q*' define the lines, '*c1*' and '*c2*' define the columns of the block.
	RCB(*r*,*l1*,*l2*,LINESET)	Copy a line-range block of text. '*l1*' and '*l2*' specify the first and last line numbers of the block.
RE	**Reg_Empty(*r*)**	Empty T-Reg '*r*'.
	Reg_Empty(*r*,EXTRA)	Empty T-Reg '*r*', even if it is currently executing as a macro.

RF	**Reg_Free**	Return the ID number of the next free (empty) T-Reg in the range 10 - 99.
RI	**Reg_Ins(*r*)**	Insert the contents of T-Reg '*r*' into the edit buffer and advance the edit position. If the register was saved as a columnar block, it is inserted as a columnar block.
	Reg_Ins(*r*,BEGIN)	Leave the edit position at the beginning of the inserted block.
	Reg_Ins(*r*,OVERWRITE)	Overwrite the existing text at the edit position.
	Reg_Ins(*r*,RAW)	Insert T-Reg '*r*' as a stream block, regardless of how it was saved.
	Reg_Ins(*r*,COLUMN)	Insert T-Reg '*r*' as a columnar block, regardless of how it was saved.
	Reg_Ins(*r*,LINEBLOCK)	Insert T-Reg '*r*' at the beginning of the current line as a line block, regardless of how it was saved.
	Reg_Ins(*r*,CLIPBOARD)	Copy the Windows clipboard to T-Reg '*r*', and then insert (paste) '*r*' into the edit buffer. Add "**+COLUMN**" or "**+LINEBLOCK**" to insert the clipboard as a columnar or line block.
RL	**Reg_Load(*r*,"*file*")**	Load '*file*' into T-Reg '*r*'.
	Reg_Load(*r*,"*file*",APPEND)	Append '*file*' to any existing contents of T-Reg '*r*'.
	Reg_Load(*r*,"*file*",INSERT)	Insert '*file*' at the beginning of T-Reg '*r*'.
	Reg_Load(*r*,"*file*",EXTRA)	If '*file*' not found in the current directory, also look in the *User Macro Directory* then in the *VEDIT Macro Directory*, and last in the *VEDIT Home Directory*.
	Reg_Load(*r*,"*file*",NOERR)	Suppress error message if file not found.
RLP	**Reg_Load_Part(...,*offset*,*length*)**	Load a partial '*file*' — beginning at '*offset*' load '*length*' bytes into T-Reg '*r*'.
RLM	**Reg_Lock_Macro**	Return the ID number of the currently "locked-in" macro. (0 if none).
	Reg_Lock_Macro(*r*)	Set up to execute T-Reg '*r*' as the "locked-in" macro in place of the "COMMAND:" prompt. '*r*' cannot be "0".
	Reg_Lock_Macro(CLEAR)	Disable any "locked-in" macro. Same as **Reg_Lock_Macro(0)**.
	Reg_Lock_Macro(*r*,EXTRA)	Keep the locked-in execution enabled even if syntax or logical errors occur during command execution. USE WITH CARE!
RMF	**Reg_Mem_Free**	Return the number of bytes free (available) for text register usage.
	Reg_Pop(*r*,*s*)	Pop (restore) T-Regs '*r*' through '*s*' from the register stack. (Be sure to push and pop in the correct order.)
RP	**Reg_Print(*r*)**	Print contents of T-Reg '*r*'. Control characters are printed according to the current Print mode.
	Reg_Print(*r*,*n*)	Print using temporary print mode of '*n*'. Expand control characters, print in hexadecimal, EBCDIC, etc.

	Reg_Print(*r,n*,RAW)	Print in "Raw" mode without margins and print all control characters as-is, without conversion.
	Reg_Print(*r,n*,NOEVENT)	Suppress the print-job start (init) string, even if enabled.
	Reg_Print(*r,n*,EVENT)	Send the print-job start (init) string, even if disabled.
	Reg_Prot(*r,s,n*)	Set write-protection of T-Regs '*r*' through '*s*' to '*n*': 0 - write protection off 1 - Visual Mode cannot alter the registers; however, command macros can alter the registers. 2 - Registers cannot be altered, loaded or emptied
	Reg_Push	**Reg_Pop** and **Reg_Push** without arguments return the number of text registers which have been pushed on the stack with **Reg_Push( )**, **Key_Cfg_Push( )** and **Win_Cfg_Push( )**.
	Reg_Push(*r,s*)	Push (save) T-Regs '*r*' through '*s*' onto the text register stack and empty the registers. Max of 128 registers.
	Reg_Push(*r,s*,SET)	The specified registers are not emptied.
RSAV	**Reg_Save(*r*,"*file*")**	Save contents of T-Reg '*r*' in the file '*file*'.
	Reg_Save(*r*,"*file*",OK)	Skip confirmation prompt when '*file*' already exists.
RS	**Reg_Set(*r*,"*text*")**	Set (place) '*text*' into T-Reg '*r*'.
	Reg_Set(*r*,"*text*",APPEND)	Append '*text*' to any existing contents of T-Reg '*r*'.
	Reg_Set(*r*,"*text*",INSERT)	Insert '*text*' at the beginning of T-Reg '*r*'.
RSIZE	**Reg_Size(*r*)**	Return the number of bytes (used) in T-Reg '*r*'.
RSTAT	**Reg_Status( )**	Display the total number of bytes in all text registers followed by the size of each register.
RT	**Reg_Type(*r*)**	Type (display) contents of T-Reg '*r*'. Control and graphics characters are expanded according to the current display mode.
	Reg_Type(*r,n*)	Type contents of T-Reg '*r*' using '*n*' as a mask to selected the desired display mode (control expansion of control and graphics characters).
RTB	**Reg_Type_Block(*r,p,q*)**	Type the block of characters between positions '*p*' and '*q*' in T-Reg '*r*'.
	Reg_Type_Block(*r,p,q,n*)	Type the block of characters using '*n*' as a mask to set the display mode.
RDI	**Registry_Delete_Item("*key**iname*")**	(Windows only) Delete item '*iname*' from the registry key '*key*'.
RDK	**Registry_Delete_Key("*key*")**	Delete the entire registry key '*key*' and all items in it.
RGI	**Registry_Get_Item(*r*,"*key**iname*")**	Get the registry item '*key**iname*' as a string value and place it in text register '*r*'.
RGN	**Registry_Get_Number(*x*,"*key**iname*")**	Get the registry item '*key**iname*' as a numeric value and place it in numeric register '*x*'.

RSI	**Registry_Set_Item("*key\iname* = *text*")**	Set the registry item '*key\iname*' to the string value '*text*'. Create the entire key and item if necessary.
RSN	**Registry_Set_Number("*key\iname* = *n*")**	Set the registry item '*key\iname*' to the numeric value '*n*'. See on-line help for options.
	Remainder	Return the remainder from the last division.
	Repeat_Count	Return pending **[REPEAT]** count, or 1 if there is none.
	Repeat_Flag	Return TRUE if **[REPEAT]** is pending.
R	**Replace("*ss*","*rs*")**	Search for the next occurrence of '*ss*', and if found, replace it with '*rs*'.
	Replace("*ss*","*rs*",ALL)	Search for all occurrences of '*ss*' and replace with '*rs*'.
	Replace("*ss*","",CONFIRM)	Prompt for the replacement options, same as **[REPLACE]**.
RB	**Replace_Block("*ss*","*rs*",*p*,*q*)**	Restrict the search within the block defined by file-positions '*p*' and '*q*'.
	Note:	See **Search( )** for other command options.
RPOS	**Restore_Pos( )**	Restore the edit position from the most recent position saved with **Save_Pos( )**. If the stack is empty, the command has no effect.
	Restore_Pos(RESET)	Empty (reset) the edit position stack.
RTAB	**Retab_Block(*p*,*q*)**	Convert spaces to the optimum number of tabs and spaces in the block of text between file positions '*p*' and '*q*'. All options for **Detab_Block()** apply too.
	Return(*n*)	Stop the currently executing macro and set **Return_Value** to 'n' for subsequent testing.
RV	**Return_Value**	Return the value of the last **Return( )** executed due to a **Call( )**. Also the return value from selected commands.
REVV	**Reverse_Video(*n*)**	Return the reverse video of screen attribute '*n*'.
	Save_Env( )	Special command for edit-session-restore feature. See on-line help for details.
SPOS	**Save_Pos( )**	Save the current edit position on a special stack of "text markers". Maximum of five positions.
SCOL	**Screen_Cols**	DOS/QNX: Return the current number of screen columns. Windows: Return number of columns in a full-size window.
SIBM	**Screen_IBM**	Return the IBM display type. (0=None, 1=Mono, 2=CGA, 4=EGA, 5=VGA).
SIBMM	**Screen_IBM_Mode**	Return the IBM hardware video display mode.
SI	**Screen_Init( )**	Initialize screen. Delete all windows and then auto-create the main window #1 as a full-sized window. Reset to configured screen colors.
	Screen_Init(ALL)	Initialize the screen and delete all windows. Window #1 is not auto-created. A window will be auto-created for the next displayed output.

	Screen_Init(ATTACH)	Initialize screen. Then create an overlapping full screen window for each open edit buffer if **{CONFIG, Auto-create windows for buffers}** is enabled.
SL	**Screen_Lines**	DOS/QNX: Return the number of screen lines. Windows: Return number of lines in a full-size window.
	Screen_Mono( )	Force monochrome screen attributes.
SR	**Screen_Reset( )**	Reset VEDIT to the current screen size and mode; rewrite the entire screen.
SS	**Screen_Size(*n*)**	Change screen size to '*n*' lines, if possible.
	Screen_Size(TOGGLE)	(DOS) Toggle between 25 line, VGA 28 line and VGA/EGA 50/43 line modes.
	Screen_Size(*l*,*c*)	(Windows) Change the size of the VEDIT program window so that a full-size edit window has '*l*' lines and '*c*' columns, if possible.
ST	**Screen_Type**	Return the screen display type. 2 = IBM PC Monochrome 3 = IBM PC Color (CGA, EGA, VGA)
S	**Search("*ss*")**	Search forwards for the next occurrence of '*ss*' and set the edit position at the beginning of it.
	Search("*ss*",ADVANCE)	Set the edit position past the end of the matched text. Does not apply to Replace( ).
	Search("*ss*",REVERSE)	Search backwards for the nearest previous occurrence of '*ss*'.
	Search("*ss*",CASE)	Perform a case sensitive search, otherwise the search is case insensitive.
	Search("*ss*",WORD)	The matched text must be a distinct "word", i.e. surrounded by separators (non-alphanumeric).
	Search("*ss*",SIMPLE)	Perform a simple search without using pattern matching or regular expressions.
	Search("*ss*",REGEXP)	Search using Regular Expressions with minimized matching.
	Search("*ss*",REGEXP+MAX)	Search using Regular Expressions with maximized matching.
	Search("*ss*",EBCDIC)	Perform search in EBCDIC file by internally translating '*ss*' from ASCII to EBCDIC.
	Search("*ss*",HEX)	The search string consists of hex values "00" thru "ff" separated by spaces. Hex words, double-words and quad-words are also supported.
	Search("*ss*",BEGIN)	Start the search from the beginning of the file.
	Search("*ss*",LOCAL)	Restrict the search to the text currently in memory. Useful for performing a "near" search in a huge file.
	Search("*ss*",SET)	Set '*ss*' to be the current search string for use by **[SEARCH AGAIN]** or **Search("")**.
	Search("*ss*",NORESTORE)	Don't restore the edit position in case of an unsuccessful search; leave it at the end (or beginning) of the file. This saves time in huge files.

	Search("*ss*",CONFIRM)	Set temporary block markers to the matched text similar to **[SEARCH]**.
	Search("*ss*",NOERR)	Suppress the error message in case of an unsuccessful search.
	Search("*ss*",ERRBREAK)	Performs a **Break** out of any command loop in case of an unsuccessful search.
	Search("*ss*",COUNT,*n*)	Search for the '*n*'th occurrence of '*ss*'.
	Search( ) **Search("")**	Search forwards for the next occurrence of the search string set by **[SEARCH]** or the "**SET**" option.
SB	**Search_Block("*ss*",*p*,*q*)**	Matching text must be entirely within the block defined by file positions '*p*' and '*q*'.
	SB("*ss*",*p*,*q*,COLUMN)	Matching text must be entirely within a columnar block.
	SB("*ss*",*p*,*q*,COLSET,*c1*,*c2*)	Matching text must be entirely within a columnar block.
	SB("*ss*",*l1*,*l2*,LINESET)	Matching text must be entirely within a line-range block.
	Search_Options	Return the search options selected in the last Search/Replace dialog-box.
	Search_Options(*n*)	Sets the saved search options to '*n*'. This determines the search mode (e.g. SIMPLE or REGEXP) and options (e.g. WORD or BLOCK) for the next **{SEARCH, Next}** function.
	Search_Status( )	Return status of **[SEARCH]** / **[REPLACE]**. (0=Cancelled, 1=[SEARCH], 2=[REPLACE])
	Set_Altered_Flag(*n*)	Set/clear (override) the internal flag **Is_Altered** which keeps track of whether the current buffer has been altered. Can be used to force a file to appear unaltered, even if it was altered.
SM	**Set_Marker(*m*,*n*)**	Set text marker '*m*' to file position '*n*'.
	Set_Marker(*m*,CLEAR)	Clear text marker '*m*'. Same as **Set_Marker(*m*,-1)**.
SVL	**Set_Visual_Line(*n*)**	Rewrite current (visual mode) window with the current line displayed on window line '*n*'.
	Set_Visual_Line(0)	Center the current line in the visual mode window.
	Sleep(*n*)	Delay for '*n*' / 10 seconds. (Max = 25.5 seconds.)
	Sort(*p*,*q*)	(Obsolete - Use **Sort_Merge**) Sort the entire lines (records) specified by file positions '*p*' and '*q*'. The sort is in ascending order using the entire line as the "key".
	Sort_Load(*file*)	Load an alternate collate table for the **Sort_Merge( )** command.
SMX	**Sort_Merge("...")**	Sort all lines in a file (or block) according to a primary key field and up to nine secondary fields. Options include ascending/descending, case sensitive, no collate. Replaces the old **Sort( )** command.

	Sound(*n*,*k*)	Create a sound (tone) of frequency '*n*' hertz and duration '*k*' milliseconds.
	Sound(*n*,*k*,EXTRA)	Add 30 milliseconds of silence after the sound.
SRD	**SR_Display( )**	Display the current search and replace strings.
SRS	**SR_Set("*ss*","*rs*")**	Set the search and replace strings.
STATM	**Statline_Message("*itext*")**	Display '*itext*' on the status line; it will remain until the next keystroke. '*itext*' can contain "|@(***r***)" to use the contents of text register '*r*'.
	Strip_High(*m*)	Strip the 8th bit from all characters in the next/previous '*m*' lines of text.
	Strip_High(*p*,*q*)	Strip the block of characters between file positions '*p*' and '*q*'.
	Strip_High(*p*,*q*,COLUMN)	Strip a columnar block.
	Strip_High(*p*,*q*,COLSET,*c1*,*c2*)	Strip a columnar block.
	Strip_High(*l1*,*l2*,LINESET)	Strip a line-range block.
SXL	**Syntax_Load(*file*)**	Load the color syntax highlighting definition file '*file*'.
SYS	**System( )**	Temporarily shell out to a DOS/NT box. Return to VEDIT with the DOS/NT command "exit".
	System("*program*")	Execute the specified command or program. Returns to VEDIT after the command/program is done. Windows: run the specified windows program in normal mode; wait for it to finish.
	System('"*program*"')	Note how single and double quotes must be used to support long filenames with embedded spaces.
	System("*prog*",SIMPLE)	Windows: run the specified windows program in minimized mode.
	System("*prog*",MAX)	Windows: run the specified windows program in maximized mode.
	System("*prog*",NOWAIT)	Windows: do not wait for the shelled program to finish (terminate).
	Note:	The "**DOS**" option is ignored by the DOS version of VEDIT; however, it is need by the Windows version to run a DOS program or command.
	System("*command*",DOS)	Windows: Start a DOS/NT command box and run the specified command or program. This box runs in a normal window which must be manually closed when it is done.
	Sys("*command*",DOS+DELETE)	Windows: Start a DOS/NT command box and run the specified command or program. This box runs in a normal window which auto-closes when it is done.
	System("*command*",DOS+MAX)	Windows: Start a DOS/NT command box which runs in a maximized window.
	Sys("*command*",DOS+SIMPLE)	Windows: Start a DOS/NT command box which runs in a minimized window and auto-closes.

	System("*command*",SUPPRESS)	Windows: Suppress the "Shell out [Cancel]" dialog box which is otherwise displayed during the shelling process.
	System("*command*",NOMSG)	DOS: Suppress the "Press any key to continue" prompt upon returning to VEDIT.
	System("*command*",LOCAL)	DOS: Suppress the immediate screen rewrite upon returning. Useful when executing several DOS commands one after another.
	System("*command*",OK)	DOS: Suppress all screen prompts and rewrites; also suppress scrolling the screen before executing the command.
	Sys_xxxx( )	(DOS only) The **Sys_...** commands for direct hardware access are documented in the on-line help topic "HARDCMDS".
TO	**Tab_Out(*n*)**	Tab-out by typing spaces to column '*n*'. If already at or past column '*n*', types two (2) spaces.
TPL	**Template_Load(*file*)**	Load the template editing macro file '*file*'.
	Time( )	Display the current system time.
	Time(EXTRA)	Display system time with 1/18 second resolution.
	Time(NOMSG)	Omit the heading "Time:".
	Time(NOCR)	Omit the following CR+LF newline.
TT	**Time_Tick**	Return the time in milliseconds since VEDIT was started.
TRB	**Translate_Block(*p,q*)**	Translate the block of text between file positions '*p*' and '*q*' by using the first of two translation tables. By default, translate from ASCII to EBCDIC.
	TRB(*p,q*,REVERSE)	Translate the block of text by using the second of two translation tables. By default, translate from EBCDIC to ASCII.
	TRB(*p,q*,NORESTORE)	Advance the edit position past the end of the translated block.
	TRB(*p,q*,COLUMN)	Translate a columnar block of text.
	TRB(*p,q*,COLSET,*c1,c2*)	Translate a columnar block of text.
	TRB(*p,q*,LINESET)	Translate a line-range block of text.
TRC	**Translate_Char(*n*)**	Return the value of character '*n*' after translation using the first translation table, e.g. from ASCII to EBCDIC.
	Translate_Char(*n*,REVERSE)	Return the value of character '*n*' after translation using the second translation table, e.g. from EBCDIC to ASCII.
TRL	**Translate_Load("*file*")**	Load the character translation tables from '*file*' into VEDIT.
T	**Type(*m*)**	Type (display) the next/previous '*m*' lines of text.
TB	**Type_Block(*p,q*)**	Type (display) the block of text between file positions '*p*' and '*q*'.

	Type_Block(*p,q*,NORESTORE)	The edit position is set past the last typed character.
	Type_Block(*p,q*,COLUMN)	Type a columnar block of text.
	Type_Block(*p,q*,COLSET,*c1,c2*)	Type a columnar block of text.
	Type_Block(*l1,l2*,LINESET)	Type a line-range block of text.
TC	**Type_Char(*n*)**	Type (display) the character with numeric value '*n*'. Control and graphics characters are displayed according to the window's current display mode.
	Type_Char(*n*,COUNT,*x*)	Type the character '*x*' times.
TF	**Type_File("*file*")**	Type (display) the file '*file*' with line numbers.
	Type_File("*file*",*n1,n2*)	Type the specified line range of '*file*' with line numbers.
	Type_File("*file*",*n1,n2*,NOMSG)	Type the specified line range of '*file*' without line numbers.
TN	**Type_Newline(*n*)**	Type (display) '*n*' newlines.
TS	**Type_Space(*n*)**	Type (display) '*n*' spaces.
UD	**Undo_Delete( )**	Insert the last text block from the deletion stack at the current edit position. Up to the most recent five blocks can be inserted.
UE	**Undo_Edit(*n*)**	Undo the last '*n*' edit changes.
UL	**Undo_Line( )**	Undo the edit changes made to the current line. At the COMMAND: prompt, undo all changes since the last COMMAND: prompt.
UR	**Undo_Reset( )**	Reset the Undo facility and clear the deletion stack.
U	**Update( )**	Display (update) the current edit buffer in a window, ensuring that the window is on top of other windows.
	Update(SUPPRESS)	Display (update) the current edit buffer in a window; don't change the order of the windows.
VER	**Version( )**	Display the VEDIT version number.
VN	**Version_Num**	Return the current version number as an integer, e.g. "602".
V	**Visual( )**	Enter Visual Mode setting the cursor position from the current edit position.
	Visual(*n*)	Perform only '*n*' operations in Visual Mode before returning automatically to Command Mode.
VM	**Visual_Macro**	Return the current value of the "Visual Mode macro" flag. Consists of multiple "bit mask" values.
	Visual_Macro(*n*)	Sets the Visual Mode macro flag to value '*n*'.
	Visual_Macro(SET)	Set masks "08" and "04" to force an auto-return to Visual Mode with a possible "Press any key to continue..." prompt.
	Visual_Macro(CLEAR)	Disable the Visual Mode macro flag. Prevents an auto-return to Visual Mode.
WA	**Win_Attach(*w*)**	Attach window '*w*' to the current edit buffer.

	Win_Attach(*w*,LINKED)	Attach window '*w*' and lock its cursor position to the current window so that both windows scroll together.
WB	**Win_Border**	Return the type of borders in the current window: 0 = Window has no border 1 = Window has minimal borders 2 = Window has full borders without scroll bars 3 = Window has full borders with scroll bars
WCASC	**Win_Cascade( )**	Move and resize all windows so that they cascade-overlap each other.
	Win_Cfg_Pop()	Restore the previous window arrangement by "popping" it from the text register stack.
	Win_Cfg_Push()	Save the current window arrangement by "pushing" it on the text register stack.
WCLR	**Win_Clear( )**	Clear (erase) entire window; home the cursor.
WC	**Win_Color**	Return the current window's text color attribute.
	Win_Color(*n*)	Set current window's text color to '*n*'.
	Win_Color(*n1*,*n2*)	Set text color to '*n1*' and "erase" color to '*n2*'.
	Win_Color(*n1*,*n2*,EXTRA)	Also set configured text/erase attributes to '*n1*' and '*n2*'.
WCE	**Win_Color_Erase**	Return the current window's "erase" color attribute.
WCOL	**Win_Cols**	Return the number of text columns in the current window.
WCRE	**Win_Create(*w*,*l*,*c*,*nl*,*nc*)**	Create the overlapping window '*w*' and switch to it. The window's top-left corner origin is at line '*l*' and column '*c*'. It's size is '*nl*' text lines and '*nc*' columns.
	Win_Create(...,PIXEL)	(Windows only) Specify the new window's origin and size in exact pixels.
	Win_Create(*w*,0,0,0,0,PIXEL)	(Windows only) Create a full-sized window.
	Win_Create(...,ATTACH)	Attach the new window to the current edit buffer.
WDEL	**Win_Delete( )**	Delete the current window. Cannot delete the main window "1".
	Win_Delete(*w*)	Delete window '*w*'.
WDET	**Win_Detach(*w*)**	Detach window '*w*' from any edit buffer.
WDM	**Win_Display_Mode**	Return the current window's display mode.
	Win_Display_Mode(*n*)	Change the current window's text display mode to '*n*'.
WEOL	**Win_EOL( )**	Erase from cursor to end-of-line in window.
WEOS	**Win_EOS( )**	Erase from cursor to end-of-screen in window.
WF	**Win_Free**	Return the ID number of the next unused (free) window.
WINH	**Win_Height**	(Windows) Return the height of the current editing window in pixels. See also **Win_Width**.
WH	**Win_Hor**	Return the cursor's horizontal (column) position in the window.

	Win_Hor(*n*)	Position the cursor horizontally to window column '*n*'.
	Win_Level(*w*)	Return display (overlapping) level for window '*w*'. 0 = entire window is visible. n = up to 'n' windows are overlapping window '*w*'.
WL	**Win_Lines**	Return the number of text lines in the current window.
	Win_Move(*w,x,y,cx,cy*)	(Windows) Move/resize window '*w*' to pixel origin '*x*','*y*' and size '*cx*' (width) and '*cy*' (height).
	Win_Move(APP,*x,y,cx,cy*)	(Windows) Move/resize VEDIT program window '*w*' to pixel origin '*x*','*y*', width '*cx*' and height '*cy*'.
WX	**Win_Next**	Return the ID number of the next (higher numbered) window attached to any edit buffer.
	Win_Next(BUFFER)	Give precedence to windows that are attached to the current buffer, before returning the ID of windows attached to other buffers.
WN	**Win_Num**	Return the ID number of the current window.
WOVL	**Win_Overlap**	Return the overlapping status of windows. (See Chapter 4 or on-line help for details.)
	Win_Page_Size	Return the number of lines in a "page" for **[PAGE UP]** and **[PAGE UP]**. (Value depends upon the number of window lines.)
WPRV	**Win_Previous**	Return the ID number of the previous (lower numbered) window attached to any edit buffer.
	Win_Previous(BUFFER)	Give precedence to windows that are attached to the current buffer, before returning the ID of windows attached to other buffers.
WR	**Win_Reserved(*w,n,*BOTTOM)**	Create the reserved window '*w*' of '*n*' lines at the bottom of the screen; resize all other windows. See **Win_Split( )** for other options.
WRB	**Win_Reserved_Bottom**	Return the number of lines in the reserved window at the bottom of the VEDIT screen area; return 0 if none.
	Win_Reserved_Bottom_ID	Return the window ID number of the reserved window at the bottom of the VEDIT screen; return 0 if none.
WRT	**Win_Reserved_Top**	Return the number of lines in the reserved window at the top of the screen; return 0 if none.
	Win_Reserved_Top_ID	Return the window ID number of the reserved window at the top of the screen; return 0 if none.
WSM	**Window_Scroll_Margin**	Return the value of the current window's scroll margin.
	Window_Scroll_Margin(*n*)	Set the horizontal scroll margin used in Visual Mode to '*n*'.
WSPL	**Win_Split(*w,n,*BOTTOM)**	Split the current window to create window '*w*' of '*n*' lines at the bottom. A value of "0" for '*n*' splits the current window into two equal sized windows.
	Win_Split(*w,n,*TOP)	Create window '*w*' of '*n*' lines at the top.
	Win_Split(*w,n,*LEFT)	Create window '*w*' of '*n*' columns at the left.
	Win_Split(*w,n,*RIGHT)	Create window '*w*' of '*n*' columns at the right.

	WSPL(...+*ATTACH)*	Attach the new window to the current edit buffer.
	WSPL(...+*NOBORDER)*	Create window '*w*' without borders (when possible).
	WSPL(...+*MINBORDER)*	Create window '*w*' with minimal borders.
	WSPL(...+FULLBORDER)	Create window '*w*' with full borders.
WSTAT	**Win_Status(*w*)**	Return status for window 'w': -1 - Window doesn't exist. 0 - Window exists, but is not attached. b - Window is attached to edit buffer 'b'.
WS	**Win_Switch(*w*)**	Switch console output to window '*w*'.
	Win_Switch(*w*,ATTACH)	Switch to window '*w*', and if attached to an edit buffer, switch to the buffer and make it the buffer's "primary" window.
	Win_Switch(STATLINE)	Switch to use the status line as a one line window.
WTILE	**Win_Tile( )**	Move and resize all windows so that they tile the screen and are all visible without overlapping.
WTTL	**Win_Title(*w*,"*text*")**	Set the title for window '*w*' to '*text*'. The title is displayed on the window's top border.
WT	**Win_Total**	Return the total number of text windows.
WV	**Win_Vert**	Return the cursor's vertical (line) position in the window.
	Win_Vert(*n*)	Position the cursor vertically to window line '*n*'.
WINW	**Win_Width**	(Windows) Return the width of the current editing window in pixels. See also **Win_Height**.
WINX **WINY**	**Win_X_Org** **Win_Y_Org**	(Windows) Return the horizontal (x) and vertical (y) origin of the current editing window in pixels.
WZ	**Win_Zoom( )**	Zoom the current window to full screen.
	Win_Zoom(CLEAR)	De-zoom the window.
	Win_Zoom(TOGGLE)	Zoom/de-zoom toggle.
	Write_Line("*file*",*m*)	Write the next/previous '*m*' lines of text to the file '*file*'.
	Xall	Exit VEDIT, saving edit changes in all buffers that contain altered files and have an assigned filename. There are no prompts.
	Xall(*n*)	Return '*n*' to the OS as VEDIT's "return code" instead of the default value of zero.
XBUF1 **XBUF2** **XBUF3** **XBUF4**	**Extra_Buffer_1** **Extra_Buffer_2** **Extra_Buffer_3** **Extra_Buffer_4**	Returns the ID # of the first four "Extra" edit buffers. This is 100-103 for the Windows version and 33-36 for the DOS version.

F - Search Modes Summary

Pattern Matching Codes

NOTE: Only the codes "**|H*hh***", "**|N**", "**|O*ooo***", "**|*ddd***" and "**|@(*r*)**" can be used on the replacement side.

|**A**	Match any alphabetic letter, upper or lower case.
|**B**	Match a blank - one space or tab.
|**C**	Match any control character.
|**D**	Match any numeric digit - "0" - "9".
|**F**	Match any alphanumeric - a letter or a digit.
|**G**	Match any graphics (high-bit) character.
|**H***hh*	Match the character with hexadecimal value '*hh*'. Can also be used on the replacement side.
|**I**	Match any word separator, including **Config_String(WORD_SEP)**.
|**K**	Match any non-standard control character other than Tab, Carriage-Return and Line-Feed.
|**L**	Match "newline": Carriage-Return and/or Line-Feed. CR is optional in DOS/Windows files.
|**M**	Multi - match any sequence of zero or more characters.
|**N**	Match "newline" characters, similar to "|**L**". CR is required in DOS/Windows files. Can also be used on the replacement side.
|**O***ooo*	Match the character with octal value '*ooo*'. Can also be used on the replacement side.
|**P**	Match any "parenthesis" - { } [ ] ( ) < >.
|**S**	Match any separator - not a letter, digit or "_" (underscore).
|**T**	Match the Tab character (value 09).
|**U**	Match any upper case letter.
|**V**	Match any lower case letter.
|**W**	Match white space - single or multiple Spaces or Tabs.
|**X**	Match extended white space - one or more Spaces, Tabs, Carriage-Returns and/or Line-Feeds.
|**Y**	Match multiple characters until the next pattern matches.
|*ddd*	Match the character with decimal value '*ddd*'. Can also be used on the replacement side.
|**000**	Match the Null (value 00) character
|<	Match beginning of line (zero length match).
|>	Match end of line (zero length match).
|*****	Match multiple characters on the same line - zero, one or more characters until the string following the "|*" is satisfied.
|**?**	Match any character.
|**!**	Match any character except following character or pattern.
|**{*set*}**	Matches one occurrence of any item in the "pattern set".
|**[*set*]**	Matches one optional occurrence of any item in "pattern set".

|@(*r*)	Access contents of text register '*r*' as a variable string. Can also be used on the replacement side.
| |	Use "| |" when you need to search for a "|".

Regular Expressions

Expressions that match a single character:

.	(Period) Simple wildcard that matches any character.
[*list*]	Matches any one character in the '*list*'.
[^*list*]	Matches any one character not in the '*list*'.
[~*list*]	Same. "**[~**" is equivalent to "**[^**".
\b	Matches the ASCII backspace character (hex 08).
\d*DDD*	Matches the character with decimal value '*DDD*'. All three digits ***MUST*** be present.
\e	Matches the ASCII <Esc> character (hex 1B).
\f	Matches the ASCII Form-Feed character (hex 0C).
\h*HH*	Matches the character with hexadecimal value '*HH*'. Both digits ***MUST*** be present.
\n	Matches the Line-Feed character (hex 0A). This is the "newline" character for UNIX type text files. To search for multiple-line patterns, use "**\N**" instead.
\N	Matches the "newline" character(s) and allows searching for multiple line patterns. The "newline" depends upon the current file type and can be CR, CR+LF or LF. ("**\N+**" and "**\N***" are currently not supported.)
\o*OOO*	Matches the character with octal value '*OOO*'. All three digits ***MUST*** be present.
\r	(Lower case) Matches the ASCII CR character (hex 0D).
\s	Matches the ASCII space character (hex 20).
\t	Matches the ASCII tab character (hex 09).
\0	(Zero) Matches the ASCII Null character (hex 00).
\	"\" followed by a special character matches that character. The special characters are: ^ $. * + ? - ~ \ | [ ] { }

Expressions that match multiple characters:

*****	Matches zero or more occurrences of the ***preceding*** single character matching expression.
+	Matches one or more occurrences of the ***preceding*** single character matching expression.
?	Matches zero or one occurrences of the ***preceding*** single character matching expression.
\1 - \9	Matches the same text as was matched by the previous '*n*'th group.

Other:

^	(Caret) Matches the beginning of a line (when it is the first character in a regular expression).
$	Matches the end of a line (when it is the last character in a regular expression).

{ }	Groups expressions for future reference in either the search string or replacement string.
|	Matches any text that is matched by the preceding OR the following expression. It cannot occur within { }.
\@(*r*)	Use the contents of text register '*r*' in this position in the search (or replace) string.

Replacement Side:

\b	The ASCII backspace character (hex 08).
\d*DDD*	The character with decimal value '*DDD*'. All three digits ***MUST*** be present.
\e	The ASCII <Esc> character (hex 1B).
\f	The ASCII Form-Feed character (hex 0C).
\h*HH*	The character with hexadecimal value '*HH*'. Both digits ***MUST*** be present.
\n	The Line-Feed character (hex 0A). This is the "newline" character for UNIX type text files.
\N	The "newline" character(s) depending upon the current file type and can be <CR><LF>, <LF> or <CR>.
\o*OOO*	The character with octal value '*OOO*'. All three digits ***MUST*** be present.
\r	The Carriage-Return character (hex 0D).
\s	The ASCII space character (hex 20).
\t	The ASCII tab character (hex 09).
\0	(Zero) The ASCII Null character (hex 00).
\@(*r*)	Use the contents of text register '*r*' in this position in the replacement string.
\1 - \9	Same text as was matched by the n'th group on the search side.
&	Entire text that was matched by the search expression.

Precedence of Regular Expression Operators:

Regular Expression Operator Precedence

Highest:	\
	[ ]
	* + ?
	{ }
	Concatenation
Lowest:	|

G - Text Register Usage

Most of VEDIT's text and numeric registers between 100 and 127 are reserved for special purposes. This table lists the reserved registers and suggested uses for the general purpose registers.

0 The "scratchpad" or default "cut and paste" register in Visual Mode.

1 - 9 Used as additional "cut and paste" registers from Visual Mode.

10 - 99 Used to hold command macros or as string variables in command macros. However, they can also be used for "cut and paste" operations.

100 Used by any auto-execution macro specified with the "**-x**" invocation option. It is also the default register for **{MISC, Load/execute macro}**. It should be reserved for the "main" macro that is running.

101 Should be reserved for the "subroutine" macros used by the main macro executing in register 100.

102 Should be reserved for the "locked-in" macro used by the main macro executing in register 100.

103 - 106 Temporary registers used as needed by keystroke macros. This prevents keystroke macros from interfering with command macros that may be running.

107 - 109 Reserved for use by the "File-open configuration macro", the "File-open/close event macros", the "Buffer-switch event macro" and the "Template editing macro". This prevents these special macros from interfering with other macros that may be running.

110 The "File-open event macro". It is executed after each file is opened, if **CONFIG(F_E_F_MACRO)** is enabled. It is executed after the "File-pre-open event macro" in register 112 and after the "File-open configuration macro" in register 115.

111 The "File-close event macro". It is executed just before each file is closed.

112 The "File-pre-open event macro". It is executed just before each file is opened.

113 The "File-post-close event macro". It is executed immediately after each file is closed.

114 The "Buffer-switch event macro". It is executed immediately after each buffer switch in Visual Mode or due to the macro language command **Buf_Switch(*r*,EVENT)**.

115 The "File-open configuration macro". It is executed after each file opened if **{CONFIG, File-open config, Enable file-open configuration}** is enabled. It is executed after the "File-pre-open event macro" in register 112 and before the "File-open event macro" in register 110.

116 Reserved for future use.

117 Internally used text register. It is used by and emptied by many block commands.

118 Internally used by the **Syntax_Load( )** and **Template_Load( )** commands to run the **`loadsyn.vdm`** and **`regprep.vdm`** macros.

119 Reserved for "subroutine" macros set up within a ".key" file.

120 Internally used text register. It is emptied with each keystroke and by many block commands. Numeric register 120 is also used internally.

121 Internally used register that holds the filename from the File selection dialog boxes. Numeric register 121 is also used internally.

122 Internally used to load the **`print.vdm`**, **`sallbuff.vdm`**, **`srchincr.vdm`**, **`loadsyn.vdm`**, **`keyedit.vdm`**, **`startup.vdm`** and **`veditsav.vdm`** macros.

123 Holds the custom editing functions for the **{TOOLS}** menu. Otherwise it must be empty. Numeric register 123 can be used within the **{TOOL}** menu.

124 Holds the custom editing functions for the **{USER}** menu. Otherwise it must be empty. Numeric register 124 can be used within the **{USER}** menu.

125 Internally used to hold the keyboard layout in a binary format. It MUST NOT be altered. (125 - 127 are accessible for use by the **`veditsav.vdm`** macro which implements the edit-session-restore feature.)

126 Internally used to hold the current window structure. It MUST NOT be altered.

127 Internally used to hold the last command line entered at the "COMMAND:" prompt. (It has a constant size.) It MUST NOT be altered.

INDEX

B

C

D