

# The WinEdt Hacker's Guide

Jason Alexander

November 21, 1999

---

## The WinEdt Hacker's Guide

Macro programming in WinEdt can be frustrating at first. Common sources of frustration for first-time macro programmers are: not knowing what macro functions are available (or appropriate) for the desired task, not understanding the flexibility of the macro functions that are available, or idiosyncrasies of the WinEdt language.

This chapter discusses all these areas, gives numerous examples of macro programming in WinEdt, and shows some dirty tricks and some conventions that are beginning to be formed by the WinEdt community. Please send reports of any errors or inadequacies (or neater examples, as well!) to [webmaster@winedt.org](mailto:webmaster@winedt.org).

This chapter discusses WinEdt's macro feature on several different levels simultaneously. I have adopted a "dangerous bend" convention (first used by the computer scientist Donald E. Knuth) to flag sections which go beyond what a first-time macro programmer would be expected to know:



If your reading this on the first time through, didn't you pay attention to the warning in previous paragraph? :- ) I'll let it go this time, but you should really try to avoid sections like these in the future until you have experience with WinEdt's macro language.

Sections which one might want to defer reading until the second or third time are marked with two dangerous bend signs. Whenever possible, macro examples will be given following the "two-column" format employed in *The L<sup>A</sup>T<sub>E</sub>X Companion* by Goossens, Mittelbach, and Samarin. The actual WinEdt macro code usually appears on the left-hand side with the corresponding WinEdt output appear on the right. For example,

```
InsText("Winedt's language");           WinEdt's language
NewLine;                                ...is powerful.
InsText("...is powerful.");
```

## 1 Basics

The simplest macros in WinEdt's library insert text or additional lines into the current document.

```
Ins("String to insert");
```

Inserts a specified string (or the contents of a string register, e.g. "%!9") using the current wrapping settings.

```
InsText("String to insert");
```

Same as `Ins` except WinEdt will not wrap the inserted string if it goes beyond the right margin.

```
NewLine;
```

Inserts a new line at the current position.

```
InsLine;
```

Inserts a new line at the current position if and only if the current line is not empty. Used judiciously, this macro can prevent multiple blank lines from appearing in a document.

```
Ins("This is some text...");
```

```
This is some text...
```

```
NewLine;
```

```
NewLine;
```

```
...and this is some more.
```

```
Ins("...and this is some more.");
```

```
Ins("This is some text...");
```

```
InsLine;
```

```
This is some text...
```

```
InsLine;
```

```
...and this is some more.
```

```
Ins("...and this is some more.");
```

If you want to insert several copies of the same thing, use `Repeat(n, "...")` — where `n` is a number (or a register containing a numeric value) and the string parameter contains the argument to be repeated. The next example demonstrates how to insert multiple line breaks into a document using `Repeat` and `NewLine`:

```
Ins("From alpha...");
```

```
From alpha...
```

```
Repeat(2, "NewLine");
```

```
Ins("...to omega.");
```

```
...to omega.
```

`Repeat` also provides a way to move the cursor to a new position in the document when you know the offset from the current position. E.g.,

```
Repeat(20, CMD("Char Left"));
```

```
Repeat(5, CMD("Line Down"));
```

moves the cursor twenty characters left and five lines down from the current position. (Luckily, more efficient ways of moving the cursor exist since traveling long distances via `Repeat` takes time.)

One important caveat about `Repeat` (which actually applies to many macros in WinEdt's library) concerns using `Repeat` with macros that take string arguments. Since the second argument of `Repeat` is itself a string, if you attempt to repeat a command that takes a string argument, you must take care to correctly nest the strings. One correct use of `Repeat` to insert ten copies of the letter `a` into the document is the following:

```
Repeat(10, "Ins('a')");          aaaaaaaaaa
```

## 1.1 First use of nested strings

One correct way of nesting strings uses different quotation marks to indicate the string inside the string (as above). Three more ways of writing this macro correctly are:

```
Repeat(10, 'Ins("a")');  
Repeat(10, 'Ins(''a'')');  
Repeat(10, "Ins(""a'')");
```

It really doesn't matter which convention you adopt, so long as you understand it and can work with it. See § 2.1 for a greater discussion of this topic.



Nothing keeps you from putting a lengthy sequence of macros in the second argument of `Repeat`: any valid macro sequence can appear in this second argument, including additional uses of `Repeat`. This allows one to write macros that perform a certain action a fixed number of times. Notice the word *fixed* occurring in that last sentence. Because the first argument of `Repeat` is a numeric constant specifying how many times the second argument ought to be repeated, you must always know at the time of writing how many times the action must be repeated. More complicated macros, requiring an action to be repeated until some condition occurs (where it is not known ahead of time how many times the action will need to be repeated) should use the more powerful `Loop` macro.

Since we've already talked about moving the cursor around the screen, we might as well mention a few more ways to do it. If you just want to automate the kinds of cursor movements that you can do with the keyboard—i.e., move around the screen, or scroll a bit—you'll be happy to know that these operations get handled via the flexible `CMD` macro.

```
CMD("Char Left");  
    Move the cursor one character to the left.  
  
CMD("Char Right");  
    Move the cursor one character to the right.  
  
CMD("Line Up");  
    Move the cursor up one line.  
  
CMD("Line Down");  
    Move the cursor down one line.  
  
CMD("Page Up");  
    Move the cursor up one page.  
  
CMD("Page Down");  
    Move the cursor down one page.
```

The `CMD` macro provides access to many, many more commands than listed here. Think of `CMD` as the macro equivalent of a Swiss-army knife: over 170 different things can be done with `CMD` (admittedly, not all of them are ones that you will use frequently, if at all). Using `CMD` is a simple two-step process: (1) look up the name of the desired command in WinEdt's on-line documentation, (2) call `CMD`, giving it the name of the command in quotation marks as above.



Basically, `CMD` invokes commands that are essential to WinEdt's behavior — like cursor movements, screen refreshes, inserting bullets, setting bookmarks, and so on. One can also invoke all the dialog boxes used to customize WinEdt's behavior, e.g., `CMD("Preferences...")`. Notice that there is some overlap with the `Menu` macro here, since both `Menu` and `CMD` can be used to bring up the Preferences dialog box. The need for two different

macros, though, stems from the degree to which WinEdt allows customization of its menus. For example, one *could* delete the Preferences... choice from the Options menu, making it impossible to invoke the Preferences dialog box with the Menu macro. However, the Preferences dialog box will always be available via the CMD macro.

The real power of WinEdt's macro language doesn't begin to emerge until we consider macros capable of conditional behavior — taking different actions depending upon the current state of some parameter. But this means that WinEdt must allow for certain parameters to vary over time. It's time to consider *variables* or *registers*.

## 2 Variables/Registers



The WinEdt documentation uses the terms 'variable' and 'register' interchangeably, so let's first note the difference between the two terms, as traditionally understood, and then see where WinEdt's macro language falls. Most (if not all) programming languages allow you to declare variables with names descriptive of their contents, like `mass`, `number_of_loops`, and so on. When you declare a variable, a chunk of memory gets allocated for storing a certain type of data.<sup>1</sup> The important point being that variables have *names*. A register, on the other hand, can't be given a special name: it's a chunk of storage that you have to refer to by whatever name the designer gave it.

So what does WinEdt have? WinEdt has registers—so you can't give them custom names—but WinEdt's registers are more flexible than variables in languages like C. WinEdt will automatically allocate memory on an as-need basis for each of its registers—you don't have to worry about that at all (if you were the sort of person who would worry about it in the first place). If you want, you can copy several pages of a document into a register that was previously used to hold a number, and WinEdt won't complain. So even if you don't like not being able to declare named variables to make your macros easier to understand, WinEdt's registers can be quite powerful.

We can distinguish between four different types of registers: *local*, *global*, *editor* and *document*. There are eleven local registers, eleven global registers, and numerous document and editor registers. Luckily, though, one can usually identify the type of a register by looking at its name alone (though there are some exceptions). Identifying registers is easy: all WinEdt registers have a %-sign in front of them.

Both editor and document registers consist of a single letter, upper or lower. The difference between these types of registers is primarily semantic rather than syntactic: editor registers contain values specifying the current state of WinEdt (current line position, column position, and so forth) and document registers contain values identifying properties of the active file type, file name, path, etc.). One cannot directly change the value of a document register, but macro functions exist for changing the value of editor registers.

For example, `%n`, `%p`, and `%t` are all document registers; `%n` contains the name of the active file (the file currently having focus in the editor), `%p` contains the path of the active file, and `%t` contains the extension of the current file. So, if the active file is:

```
c:\My Documents\Sundials-R-Us\Gnomon Sales\data.foo
```

then `%p` equals `c:\My Documents\Sundials-R-Us\Gnomon Sales`, `%n` equals `data`, and `%t` equals `.foo`. Changing the active file causes the values of these registers to change.


Examples of editor registers are: `%c`, containing the column position of the cursor and `%l`, containing the line position of the cursor. Note that certain editor registers eliminate the need for

---

<sup>1</sup>With some exceptions. C's union type allows you to store different types of data in the same (approximately) chunk of memory.

certain macros. Need the word the cursor is currently positioned on? Use %W, which contains the word located at the current cursor position (%L gives you the contents of the entire *line*). As I said above, editor registers can be changed: to change the value of %C, you have to move the cursor (manually or via the GotoCol or GotoCL macros).

Local registers are the easiest to spot: %!0, ..., %!9, and %!? are the eleven local registers. Global registers differ by lacking an exclamation point: %0, ..., %9 are global registers (so is %I, but it has a special function we'll return to later). From the point of view of a macro programmer local registers are no different than global registers save that *global registers retain their values between WinEdt sessions and should not have multi-line values*. The prohibition on multi-line values arises because global registers have their values stored in the `winedt.ini` file and multi-line values can corrupt the `winedt.ini` file.

 So what? Well, most of the time you will probably want to use local registers when writing WinEdt macros (especially ones that just save on some typing). But using global registers can allow you to create *state-sensitive* macros that respond differently depending upon the current state of the editor, where the editor remembers its current state from time to time.

Take a look at the macros located in %B\Macros\math. These macros employ the method of global registers to toggle between two modes: a  $\LaTeX$  math mode, and a  $\LaTeX$  normal text mode. A number of macros are provided to hook to active strings. When done so, `a gets translated to `\alpha` when math mode is on, but gets left as `a in normal text mode.

## 2.1 Strings and things

Most of the macros in WinEdt's standard library take arguments, some several. When reading arguments there are two different *types* of arguments WinEdt may expect to find: *numeric* and *string*. A numeric argument can be a constant — e.g., 17 — a register (local or global) currently set to a numeric value, or an integer expression using the standard arithmetic operators `+-*/`. An “integer expression” means any well-formed expression using those operators, registers containing well-formed integer expressions, and constants (with parentheses used for grouping). The following are examples of valid integer expressions:

$$\begin{aligned} &2+3 \\ &((2+3)/16) \\ &\%!1+\%!2 \\ &((\%!1*\%!2)/12)*(-17/\%!3) \end{aligned}$$

The following macro illustrates a simple use of integer expressions:

```
1 LetRegNum(1,1);
2 LetRegNum(2, "((12+\%!1)/4)");
3 LetRegNum(3, "\%!2 + \%!2");
4 InsText("Reg. \%!1 = \%!1"); NewLine;
5 InsText("Reg. \%!2 = \%!2"); NewLine;
6 InsText("Reg. \%!3 = \%!3");
```

Reg. %!1 = 1  
Reg. %!2 = 3  
Reg. %!3 = 6

### 2.2 Using integer expressions

Line 1 stores the numerical value 1 in register %!1. In line 2, register %!2 receives the numerical value 3 because the result returned by WinEdt when performing arithmetic division truncates

decimals. Finally, line 3 assigns to register %!3 the value 6 because when WinEdt processes the second argument of `LetRegNum`, register %!2 expands to the value 3 and  $3 + 3 = 6$  (in case you didn't know).

Note the use of repeated %-signs in lines 4–6. Since WinEdt uses percent signs in the names of registers, you must tell WinEdt when you really want a percent sign instead of a register name. This is done by including two percent signs (%%) at the appropriate place. People familiar with other macro languages, such as  $\text{\TeX}$ , are probably already familiar with the need to repeat certain “special” characters; if you are not such a person, just remember to type %% whenever you want % to appear in the text and you will get by fine.<sup>2</sup>

One occasionally needs to nest strings when writing simple macros (see example 1.1) and almost always when writing more complicated macros (see examples 2.3, and 2.4). Mastering nested strings is essential to becoming a real pro at WinEdt's macro language. WinEdt recognizes several different ways to nest strings:

**Correct:**

```
"this is a 'string' inside a String"
'this is a "string" inside a String'
"this is a ""string"" inside a String"
'this is a ''string'' inside a String'
```

**Incorrect:**

```
'this is NOT a 'string' inside a String'
"this is NOT a "string" inside a String"
```



Although 2.2 used `LetRegNum` three times, that wasn't necessary. You might have a complicated integer expression that you want to evaluate several times, allowing the values stored in the registers appearing within that expression to change between evaluations. Using `LetRegNum` like we did above will give you headaches, since `LetRegNum` immediately evaluates its second argument to obtain an integer value for the assignment.

The way around this problem is to store the complicated integer expression in a local register as a *string*. This allows one to suppress expansion of the named registers until you actually want to evaluate the expression. Here's an example:

---

<sup>2</sup>Until you start working with nested strings. Then things become considerably more complicated even though the rules stay the same. Working with registers and percent signs inside nested strings will be dealt with in a later section, but see 2.3 and 2.4 for a taste of what will come.

```

1 BeginGroup;
2 LetReg(1, '((100*%!2)+(10*%!3)+(%!2*%!3)');
3 LetRegNum(2,0);
4 LetRegNum(3,0);
5 Loop('>
6   Loop(">
7     LetRegNum(4, '!'%!1');>
8     InsText('!%%!2 = %%!2, %%!3 = %%!3, %%!4 = %%!4');>
9     NewLine;>
10    LetRegNum(3, %%!3+1);>
11    IfNum(%%!3,3,'=', 'Stop', '');>
12    " );>
13    LetRegNum(2, %%!2+1);>
14    LetRegNum(3,0);>
15    IfNum(%%!2,3,"=", "Stop", "");>
16    ');
17 EndGroup;
18 End;

```

### 2.3 Storing integer expressions for later use

This macro produces the following output:

```

%!2 = 0, %!3 = 0, %!4 = 0
%!2 = 0, %!3 = 1, %!4 = 10
%!2 = 0, %!3 = 2, %!4 = 20
%!2 = 1, %!3 = 0, %!4 = 100
%!2 = 1, %!3 = 1, %!4 = 111
%!2 = 1, %!3 = 2, %!4 = 122
%!2 = 2, %!3 = 0, %!4 = 200
%!2 = 2, %!3 = 1, %!4 = 212
%!2 = 2, %!3 = 2, %!4 = 224

```

How? First, the integer expression  $((100*%!2)+(10*%!3)+(%!2*%!3))$  gets stored in %!1 (note the use of double parentheses to suppress expansion). Then %!2 and %!3 are initialized to 0. We then enter a doubly-nested loop. We take advantage of some expansion trickery in the first line of the double loop. Here, `LetRegNum` is at level 2 since it appears in a string inside a string; if we wanted to suppress expansion of %!1, we would have to use at least four percent signs %%%!1 to raise that occurrence of that register to level 3 (or higher). However, we *don't* want to suppress expansion since we want to use the contents of %!1 — the complicated integer expression — when assigning a value to %!4. Thus %!1 gets expanded *before* `WinEdt` begins processing the `LetRegNum` call. When `WinEdt` begins processing the `LetRegNum` call, it first expands the second argument of this macro — the complicated integer expression we want to use! Thus, %!4 gets assigned the value 0 the first time through the double loop, since both %!2 and %!3 equal 0.

String arguments can be explicitly specified — e.g., "This is a string" — or they can use registers (local or global) whose contents “make sense” when expanded. Make sense? Here's what I mean. Consider the macro `InsText`. It takes a single string argument and inserts it at the current cursor position. Clearly any string you specify makes sense here, for `WinEdt` will expand all registers present in the string and then insert the expanded string into the document. However, consider the macro `Loop`. Here, the only strings which make sense are those which expand into a sequence of macros `WinEdt` can execute. Giving "This is a string" as an argument to `InsText` won't result in an error, but giving it as an argument to `Loop` will cause `WinEdt` to complain.





Defining the notion of a string “making sense” more precisely would require that we introduce the notion of a *valid string*, where the validity of a string depends upon the macro we are speaking of. (Exercise: prove that no string is valid for every macro.) I leave this to the hands of those more capable at theoretical computer science than myself.

New users often find the way WinEdt handles string arguments one of the more puzzling aspects of WinEdt’s macro language. Here’s an example:

```
LetReg(1, "This is some text.");           This is some text
Ins("%!1");
```

LetReg takes two arguments, a numeric argument telling which one of the local registers (%!0, ..., %!9) to set and a string argument containing the soon-to-be value of the register.



Registers can also be used in places where WinEdt expects a numeric argument and, by using registers with the Do macro, one can use registers to access string registers. The following example shows how to exploit this primitive type of pointer.

```
1 BeginGroup;
2 LetRegNum(0,4);
3 LetRegNum(1,5);
4 LetRegNum(2,6);
5 LetRegNum(3,7);
6 LetReg(%!0,"Jellical cats come out tonight,")
7 LetReg(%!1,"Jellical cats come one, come all,");
8 LetReg(%!2,"The Jellical moon is shining bright,");
9 LetReg(%!3,"Jellicals come to the Jellical ball."); %!4: Jellical cats come out tonight,
10 LetRegNum(8,4); %!5: Jellical cats come one, come all,
11 Loop("> %!6: The Jellical moon is shining bright,
12   InsText('Reg. #%%!8: '); %!7: Jellicals come to the Jellical ball.
13   Do('>
14     InsText("%%!8");>
15     NewLine;>
16   ');>
17   LetRegNum(8,%%!8+1);>
18   IfNum(%!8, 7, '>', 'Stop', '');>
19 ");
20 EndGroup;
```

#### 2.4 Using registers as “pointers”

Registers 0, ..., 3 point to registers 4, ..., 7 and are used to assign values to registers 4, ..., 7 via LetReg. After assigning values to these registers, we loop four times, printing out the contents of one register (plus a NewLine) on each pass. Note how in the Loop macro we use register %!8 not only to control the number of iterations we perform, but also to determine the contents of which register we insert.

Omitting quotation marks in Ins("%!1") generates the WinEdt warning: String Parameter Expected. Why? Because Ins requires a string argument and %!1 isn’t a string — it’s the *name* of a local register *containing* a string. One might wonder why Ins("%!1") works, then. After all, why doesn’t WinEdt interpret "%!1" as a string containing the sequence of characters %, !, and 1, causing the text %!1 to appear in the document?

The answer lies in WinEdt’s treatment of strings parameters. Whenever WinEdt executes a macro requiring a string parameter, it first expands the string parameter *before* using it. When the parameter gets expanded, all registers at the bottom level (i.e., registers that aren’t suppressed) get replaced by their values, and every register of a higher level gets its level lowered by one. When WinEdt processes the macro Ins("%!1"), WinEdt replaces the register %!1 by its

value because it occurs at the bottom level. WinEdt then inserts the translated expression into the document.



Strictly speaking, WinEdt doesn't keep track of levels at all; as far as WinEdt is concerned, all it works with are strings (or nested strings) and registers. My reason for introducing the concept of a *level* of a register is to make it easier to keep track of how many %-signs are needed at any given point.

If you prefer not to think in terms of levels, here's how WinEdt actually handles strings and registers inside nested strings. %!1 is the name of a string register and gets replaced by its contents whenever WinEdt expands %!1. WinEdt also expands %% to a single percent sign: %. If you write a macro that has a register appearing inside a nested string, you can control when that register gets replaced by its value by the number of percent signs you stick in front of it.

Consider the following macro:

```
LetReg(1, "The best of all possible strings");
LetRegNum(2,1);
IfNum(%!2, 1, "=", >
    "Repeat(3, 'InsText("%%%!1");
      NewLine;')", >
    "Exit">
);
```

This macro is a convoluted way of inserting three copies of "The best of all possible strings" into the document (with line breaks appearing between each copy). However, it illustrates how WinEdt expands strings.

When WinEdt begins executing this macro, it begins on the first line with `LetReg`. WinEdt takes the string argument, expands it, and assigns the expanded string to register %!1. Since there were no register expressions occurring in the string argument, %!1 now contains The best of all possible strings. WinEdt then processes the second line and assigns %!2 the numeric value 1.

When WinEdt begins to process the third line, it first concatenates all the lines ending with > (this allows us to use line breaks to increase the readability of macros, since, technically, all WinEdt macros must lie on a single line). Once that has been done, WinEdt checks to see whether %!2 equals 1 or not. Since it does, WinEdt will execute the fourth argument of `IfNum`.

Before WinEdt executes this fourth argument, it expands the string. After expansion, WinEdt faces the sequence of macros:

```
Repeat(3, 'InsText("%%!1"); NewLine;')
```

Notice how the four %-signs was reduced to two. Why? As WinEdt expanded the fourth argument of `IfNum`, it encountered the expression %% which it replaced by %. WinEdt then found another occurrence of the expression %% which also was replaced by %. WinEdt left !1 alone since this was not part of any register. (WinEdt also adjusts the quotation marks so that the strings are nested correctly.)

WinEdt then executes the `Repeat` macro. In doing so, WinEdt first expands the second argument of `Repeat`. The sequence of commands after expansion is:

```
InsText("%%!1"); NewLine;
```

which gets repeated three times. As WinEdt executes the `InsText` macro, it first expands the string argument. In the process of expansion, WinEdt finally finds an occurrence of the register %!1 and replaces it by the text The best of all possible strings which WinEdt finally inserts into the document.



One might want to suppress WinEdt's automatic replacement of registers (this happens most frequently when writing complicated macros containing nested loops). WinEdt provides two ways to suppress register replacement: (1) doubling the number of %-signs at the register's name raises its level by one (thus delaying its replacement), or (2) adding a ! before the string parameter prevents the entire string from being translated, effectively raising the level of every register inside the string by one.

Suppressing translation of the entire string can be used to eliminate the need to include horribly long streams of parenthesis in deeply nested macros, but it doesn't provide the same level of control that method (1) has.

```
LetReg(1, "It was...");
LetReg(2, "the best of times.");           It was... the best of times
Ins("%!1 %!2");
```

```
LetReg(1, "It was...");
LetReg(2, "the best of times.");           It was... %!2
Ins("%!1 %%!2");
```

```
LetReg(1, "It was...");
LetReg(2, "the best of times.");           %!1 %!2
Ins(!"%!1 %!2");
```

## 2.2 Setting the values of registers

First of all, what registers have values one can modify from within a macro? Certainly the local string registers (%!0,...,%!9), since these registers are intended to be the macro writer's main temporary storage area. You can also modify the value of the global string registers (%0,...,%9) but, if you do so, you must keep in mind the following very important restriction:

*Do not attempt to place an expression spanning multiple lines inside any global string register. Since the values of these registers are stored permanently in your `winedt.ini` file, this file can become corrupted if one assigns multi-line values to the global string registers.*

This restriction does not usually present a problem since one rarely is tempted to use a global register. And, in the cases where one does need a global register, there is either no need to use multi-line values, or this limitation can be easily circumvented.

WinEdt provides several ways to modify the values of the local string registers. Modifications which do not require any input from the user can be done using `LetReg` (assigning string values to local registers) or `LetRegNum` (assigning numerical values to a local register). WinEdt also provides a way to write interactive macros that collect input from the user, but discussion of these macro methods won't appear until § 5.

```
LetRegNum(StrRegister: 0..9, Num_Expression: -999999999..999999999);
This function can be used to assign the result of a numeric expression to the string Register (eg. %!0).
```

```
LetReg(StrRegister: 0..9, "Value");
Assigns the specified value to the String Register (eg. %!0).
```

If you really want to fly in the face of danger, you can assign values to the global registers (but *do heed* the above warning) via `LetStr`—the global register equivalent of `LetReg`. `LetStr` assumes you know what you're doing when you make assignments and it will not check to see if you attempt to assign a multi-line value to a global register. As said before, unless you are careful this is a really easy way to irreversibly damage your `winedt.ini` file.

```
LetStr(StrRegister: 0..9, "Value");
Assigns the specified value to the Global String Register (eg. %0).
```



WinEdt provides another way to set the value of local string registers which often comes in handy. `GetSel` copies the currently selected block of text into the specified string register, if the active file has selected text. If no text is selected, the behavior of `GetSel` depends on the value of its first parameter: if 0, the specified string register receives the empty string; if 1, the specified string register receives the entire document. Obviously, one should exercise caution when using the latter case: you probably don't want to select the entire document if the next command is `CMD('Delete')` or `CMD('Sort Lines...')`!

Occasionally, the need arises to check the character to the immediate left or right of the cursor, performing some conditional action based on what you find there. `GetSel` provides a way to do this. Note, though, that this technique should only be used for isolated tests; using this technique inside a loop can cause the macro to run slowly because `CMD('Select Char Left')` and `CMD('Select Char Right')` do not execute quickly. The following macro demonstrates how to use `GetSel` in this way. Can you guess what this macro might be useful for?

```
1  CMD('Backspace');
2  CMD('Select Char Left');
3  GetSel(0,9);
4  CMD('Char Right');
5  IfStr('%!9',' ','=' , 'InsText(" ");End', '');
6  IfStr('%!9','.', '=' , 'InsText(" ");End', '');
7  IfStr('%!9',',', '=' , 'InsText(" ");End', '');
8  IfStr('%!9','!', '=' , 'InsText(" ");End', '');
9  IfStr('%!9','?', '=' , 'InsText(" ");End', 'InsText('' ''')');
```

### 2.5 Conditional insertion of characters based on surrounding text

Let's walk through this macro to see what, exactly, it does. First of all, line 1 shows I lied when I said the macro checks the character to the left of the cursor; the macro begins by *deleting* the character to the left of the cursor. Once that's done, lines 2–4 copy the character to the left of the cursor into register %!9 and then move right. (We need to move right in line 4 because the command in line 2 causes the cursor to move one character to the left. If we did not move right one character, the text inserted in lines 5–9 would appear in the wrong place.)

Lines 5–9 demonstrate one way of performing a multiple-way branch—that is, a conditional operation like C's `switch` statement or TeX's `\ifcase`. Since WinEdt doesn't have command providing a multiple-way branch outright, we need to manufacture one by using multiple `If...` statements. Simply use one `If...` test for each possible branch, where each `If...` test (they need not all be the same) has an empty `else`-clause. If you want to specify a default branch, that is, a sequence of commands to be executed when none of your explicit `If...` tests succeed (always a good idea), put this sequence of commands in the `else`-clause of the *last* `If...` test. The above macro uses this construction: notice the use of `End`; to terminate macro execution after a successful `IfStr` test and how the `IfStr`s of lines 5–8 all have empty `else`-clauses.

Lines 5–9 check the character in register %!9 to see whether it's a space, period, comma, exclamation point, question mark, or other (that's the default case). If it's a space, WinEdt inserts ` ` into the text at the current cursor position. If it's one of the four punctuation symbols, WinEdt inserts ' ' into the text. In all other cases, a single " gets inserted. Note how *four* ' -marks were needed in lines 6–9 to get *two* ' -marks in the text.

When would such a macro be needed? Consider what would happen if " were made an active string with this macro bound to it. Then whenever a " was typed, WinEdt would be replacing the " with one of three possibilities depending upon the character preceding the ". In other words, this macro would create "smart quote" behavior inside WinEdt, similar to that available in Emacs or word processors like Microsoft Word.



Macro 2.5 runs slowly, so slowly it might frustrate some users. If you need to write macros that test surrounding text in order to decide what type of action to take, a faster way needs to be found. The author has found that, when speed is of the essence, avoiding `CMD("Select Char Left")` entirely gives the best results.

Using `CMD("Select Char Left")` (or its sister command `CMD("Select Char Right")`) slows macro execution down because they use Windows messaging commands to perform the indicated actions—internally, essentially the same sorts of procedures as if you move the cursor into position, held the Shift key down, and then

moved either left or right. I.e., a Windows message is generated, sent through the message loop, dispatched, and processed.

All that involves unnecessary computational overhead. This overhead can be avoided by using `GotoCol`, `GotoLin`, or `GotoCL` to move the cursor into the correct position, `SetSel(0)` to first deselect everything, and then `SetSel(1)` (some other nonzero number may also be used) to begin the selection process. At this point, using `GotoCol`, etc. moves the cursor to the specified position, selecting all the text in between the cursor position when `SetSel(1)` was called at the new position. At this point, `GetSel` may be used to copy the selected text into the desired register.

Macro 2.5, rewritten using these techniques, will then look like:

```

1  CMD( 'Backspace' );
2  Mark(0);
3  IfNum( '%c', 1, '=', 'InsText( "" );End', '' );
4  SetSel(1);
5  GotoCol( '( %c-1 )' );
6  GetSel(0,9);
7  SetSel(0);
8  Goto(0);
9  IfStr( '%!9', ' ', '=', 'InsText( " " );End', '' );
10 IfStr( '%!9', '.', '=', 'InsText( "." );End', '' );
11 IfStr( '%!9', ',', '=', 'InsText( "," );End', '' );
12 IfStr( '%!9', '!', '=', 'InsText( "!" );End', '' );
13 IfStr( '%!9', '?', '=', 'InsText( "?" );End', '>
14     'InsText( " " );End', '>
15 End;
```

### 2.6 Faster conditional insertion of characters based on surrounding text

After deleting the undesired " character in line 1, the cursor is correctly positioned for inserting the desired character(s) (either ' ' or ' ' or "). However, selecting the text requires that we move the cursor to the right by one, once the text has been selected. The use of `Mark(0)` in line 2, following by `Goto(0)` in line 8, performs these needed movements. If you try both forms of the macro on a fast (but not blazingly so) computer, you may detect the speed difference between the two versions.

Aside from the local and global string registers, most of WinEdt's other registers have their values changed indirectly through the use of other macro commands. For example, the editor registers `%c` and `%l` contain, respectively, the current column and line number (the current column and line number identifies the position of the cursor in the active document). You cannot use `LetRegNum` to change the value of these registers. To change the value of `%c` or `%l`, you must either manually move the cursor (the changes are automatically reflected in the values of `%c` and `%l`) or use `GotoCol`, `GotoLin` or `GotoCL` to jump the cursor to a particular spot in the document.

```

GotoLin( LineNum: 1..1600000 );
    Go to the specified line and track the Caret's position.

GotoCol( ColumnNum: 1..9999999 );
    Go to the specified column and track the Caret's position.

GotoCL( ColumnNum: 1..9999999, LineNum: 1..1600000 );
    Go to the specified text position and track the Caret.
```

However, WinEdt does allow the user to change a few document registers directly through the commands `SetFileName` and `SetFileMode`. These commands will probably be used rather infrequently, but they are there in the event you need to use them.

```
SetFileName("File Name");
```

Changes the file name of the current document.

```
SetFileMode("File Mode");
```

Changes the "Mode" of the current document and resets certain properties. Check WinEdt's documentation on modes for more information.

## 3 Procedures and Functions

Macro languages can save time and reduce errors by automating repeated tasks. As people become more familiar with the flexibility of a macro language, they tend to write longer macros to perform more complicated tasks. As macros grow in length, it becomes important (from the programmer's point of view—the macro interpreter couldn't care less!) to impose some sort of structure upon the macros, making them easier to follow and to write. This structure usually involves the creation of subroutines and procedures, so that a single subroutine can be called every time it's needed, obviating the need to include that code at every point it's used.

As a trivial example, suppose we want a macro to automatically create a new  $\text{\LaTeX}$  document, configuring the document by adding text at various places. In particular, suppose we want the document to look like this when the macro completes operation:

```
\documentclass[12pt]{article}
\usepackage{amsmath}
\usepackage{verbatim}

\begin{document}

*

\bibliography{mybibliography}
\bibliographystyle{plain}
\end{document}
```

(Note: The bullet character which is now represented by a red asterisk, had in previous versions of WinEdt been represented by  $\square$ .)


One way to code the macro without using subroutines (or procedures) would be:

```
1  CMD("New");
2  Mark(0);
3  Ins("\documentclass[12pt]{article}"); NewLine;
4  Ins("\usepackage{amsmath}"); NewLine;
5  Ins("\usepackage{verbatim}"); NewLine; NewLine;
6  Ins("\begin{document}"); NewLine; NewLine;
7  Ins(" *"); NewLine; NewLine;
8  CMD("Go To Beginning Of Line");
9  Ins("\bibliography{mybibliography}"); NewLine;
10 Ins("\bibliographystyle{plain}"); NewLine;
11 Ins("\end{document}");
12 Goto(0);
13 CMD("Next Bullet");
```

### 3.7 Macro without subroutines

Without explicitly instructing WinEdt, in line 8, to go to the beginning of the line, the text of lines 9–11 would be inserted in the wrong column. Note how we used `Mark` in line 2 to “memorize” a cursor position and `Goto` in line 12 to return to that position. (The only reason for returning to that position is that it moved the cursor before the bullet, thus guaranteeing that the call to `CMD("Next Bullet")` would snap the cursor to the desired point.)

This macro works perfectly well and most people would stop here. One problem, though, is that  $\text{\LaTeX}$  documents have a certain structure to them: a *preamble* (everything appearing before the `\begin{document}`) and what I’ll call a “postamble” (everything appearing after the `\end{document}`). This macro assumes the postamble to be empty—a reasonable assumption since  $\text{\LaTeX}$  ignores anything in the postamble—but, conceivably, there may be instances where the postamble would not be empty: version control information might be placed there, information about the author, etc. Since the preamble and postamble constitute logically separate parts of a  $\text{\LaTeX}$  document, let’s rewrite the macro using subroutines to reflect this fact. While we’re at it, let’s include a subroutine for all the middle stuff not appearing in either the preamble or postamble. (Although further enhancement of this macro might seem an unnecessary exercise in obfuscation, it illustrates how to use subroutines. In the future we’ll see many cases where using subroutines clarify the operation of a macro.)

 Strictly speaking, WinEdt doesn’t provide for subroutines. WinEdt provides the `Exe` macro, which looks for a specified macro file; if the file exists, WinEdt executes the sequence of commands contained within that file. But this allows us to write code that behaves *as if* WinEdt allowed one to define subroutines: simply write your code for the function, but save it to a file with a descriptive name and use `Exe("...")` to call the subroutine.

Rewriting macro 3.7 so it uses subroutines as described means we must create two macro files on disk, one for the preamble and one for the postamble. You can save your macro files wherever you want to, just make sure you remember where they are. The author finds it convenient to prefix the name of macro files containing subroutines with two underscore characters, `__`, so he can tell just by looking at the file name that they aren’t stand-alone macro files (and that he might get into trouble if he deletes them!).

Since the postamble subroutine is the easiest to write, let’s start with it:

Relax;

#### 3.8 Postamble subroutine for macro 3.7

Assume the postamble subroutine is saved as %B\Macros\\_\_Postamble.edt, where the %B register stands for the WinEdt base directory.

The preamble subroutine isn't much longer:

```
Ins("\documentclass[12pt]{article}"); NewLine;
Ins("\usepackage{amsmath}"); NewLine;
Ins("\usepackage{verbatim}"); NewLine; NewLine;
```

#### 3.9 Preamble subroutine for macro 3.7

(Assume the preamble subroutine is saved as %B\Macros\\_\_Preamble.edt.)

And for the middle stuff subroutine we have:

```
Ins("\begin{document}"); NewLine; NewLine;
Ins(" *"); NewLine; NewLine;
CMD("Go To Beginning Of Line");
Ins("\bibliography{mybibliography}"); NewLine;
Ins("\bibliographystyle{plain}"); NewLine;
Ins("\end{document}");
```

#### 3.10 Middle stuff subroutine for macro 3.7

(Assume it was saved as %B\Macros\\_\_MiddleStuff.edt.)

When rewritten using subroutines, macro 3.7 looks like:

```
1  CMD("New");
2  Mark(0);
3  Exe("%B\Macros\__Preamble.edt");
4  Exe("%B\Macros\__MiddleStuff.edt");
5  Exe("%B\Macros\__Postamble.edt");
6  Goto(0);
7  CMD("Next Bullet");
```

#### 3.11 Macro with subroutines

Compare macro 3.11 to 3.7. Trivially, we see that 3.11 is shorter, but not so much so that this should convince us to use subroutines whenever possible. The author believes that the logical structure of 3.11 is clearer than 3.7 but this, too, may be disputed. However, three advantages of using subroutines are as follows: (1) future modifications of macro code become easier since the code has been broken down into bite-size chunks, and (2) debugging macros becomes easier since one can (usually) debug the subroutines in isolation from the main code, and (3) the need to use strings containing strings containing strings... disappears. Judicious use of subroutines should enable you to avoid working with nested strings of level greater than two. If you find WinEdt's conventions for nested strings and repeated %-signs confusing, this alone might convince you to use subroutines.





Using subroutines can make *writing* complicated macros easier, too. You can define `Exe` to be an active string so that when you double-click on it, WinEdt will automatically try to open the specified macro file. Simply go to Options | Settings... | Active Strings and bind the macro `[Do('Open(%?)')]` to a double-click of `Exe(?)`.

## 4 Recursion

Good macro languages allow for recursive programming, and WinEdt's language does not prove exception to the rule. Recursive programming in WinEdt requires that we use the method of subroutines sketched in § 3. Since writing recursive functions often requires several macro files to be saved to disk, we need to indicate in our examples what macro code belongs to which macro file. The convention that shall be adopted is to surround macro code by comments indicating the file name and where the file ends. This allows us to include the code for several files in the same display. E.g.,

```
// Filename: _MyMacro.edt
...
// Endfile

// Filename: _MyOtherMacro.edt
...
// Endfile
```

Consider the following simple example:

```
// Filename: %B\Macros\guide\_recursion.edt
InstText("a");
IfNum(%c,10,'<',\Exe("%B\Macros\guide\_recursion.edt'),'');
InstText("b");
// Endfile
```

### 4.12 A simple recursive macro

When started at the beginning of a line, WinEdt produces

```
aaaaaaaaabbbbbbbbb
```

If started five spaces in, we get

```
aaaabbbb
```

(Only four as appear when started five spaces in because column numbering starts at 1. When at the beginning of a line, the cursor is in column 1. Five spaces in, the cursor is at column 6, so WinEdt can only insert four as until the test fails.)

### 4.1 A lengthy example

Consider the following problem, posed by Heiner Richter: can WinEdt be customized so that it will automatically convert any string of digits (such as 1235678) to the form 1.235.678?

Before starting, let's generalize the problem to the following: customize WinEdt so that it will take a string of digits and insert the contents of a string register after every three digits (counting from the right). This generalization makes the macro more useful since conventions vary on whether to use a period, comma, raised dot, or space when separating digits.

First off, let's consider how this macro will be invoked. We don't want to invoke the macro until the digit string has been fully entered, but how do we postpone execution of the macro until the string has been completed? One natural way of doing this is to use active strings. But what active string? We can't make *every* digit active because WinEdt will then invoke the macro each time we type a digit. We could require that the user wrap the digit string with an identifier unlikely to occur in ordinary text, say `@digits{...}`, but this solution is clumsy: we want the macro to make it easier to type a string of digits, and typing `@digits{12345}` when you want `12,345` just makes more work for yourself.

One solution recognizes that a digit string has, as a natural delimiter, a digit followed by a space. Thus we can make the strings `'0 '` (zero followed by a space), ..., `'9 '` (nine followed by a space) active. This guarantees that the macro gets invoked only at the end of a digit string.

Second, we note that the macro will need to do some initialization: we need to place the desired delimiter into a string register and set a counter to 0 so that we can count off three digits at a time. Now, we can write the macro using either `Loop` or recursion. The problem using `Loop` is that it requires us to nest strings and—if the macro is long—this can be a headache. However, if we use recursion, we have to be careful of how we set the initial conditions. We cannot set the initial conditions inside the same subroutine used for the recursion because each time the recursion occurs the initial conditions will be re-established! Consequently, we will put the initialization code in a macro called `_APstartup.edt`, calling the recursive subroutine at the very end.

```

1 // Filename: %B\Macros\ap\_APstartup.edt
2 BeginGroup;
3 StartWorking("Adding separators...");
4 LetReg(5,','); // <--- Change this if you want commas instead of periods
5 LetRegNum(6,0); // Stop indicator
6 LetRegNum(7,%c); // Original column position
7 LetRegNum(8,0); // Found digit indicator
8 GotoCol('%c-1'); // Backup behind the space delimiter
9 Exe('%B\Macros\AddPeriods\_APchew.edt');
10 // Endfile

```

#### 4.13 Initialization routine for the “add periods” macro

Line 1 contains `BeginGroup` so that the separator addition can be undone with a single `Undo`. We call `StartWorking` in line 2 out of courtesy for the user since the macro might take some time to complete. Register `%!5` contains the delimiter string (here we use a comma). Register `%!6` contains a binary flag indicating whether the macro has completed (we'll see later why this is necessary). Register `%!7` contains the column that the cursor will return to when macro execution finishes. Finally, register `%!8` contains the number of digits we've found so far.

After initialization, WinEdt calls the macro `_APchew.edt`. This subroutine decides where to place the separators. Consider part of it:

```

1  SetSel(0); SetSel(1);
2  IfNum(%c,1,'=',>
3    'Exe("%B\Macros\AddPeriods\_APend.edt")',>
4    '');
5  GotoCol('(%c-1)'); GetSel(0,9);
6  IfStr('%!9','0','=',>
7    'LetRegNum(8,"%%!8+1");>
8    IfNum("%%!8","4","=",>
9      "SetSel(0);>
10     GotoCol('%%%c+1');>
11     InsText('%%%!5');>
12     GotoCol('%%%c-1');>
13     LetRegNum(8,0);>
14     LetRegNum(7,'%%%!7+1')",>
15     ">
16   );>
17   Exe("%B\Macros\AddPeriods\_APchew.edt")',>
18   '');
19 );
20 IfNum(%!6,1,'=', 'Exe("%B\Macros\AddPeriods\_APend.edt")', '');

```

#### 4.14 Part of the recursive subroutine called in 4.13

Lines 1–5 copy the character to the left of the cursor (if there is one) into register %!9. The test in line 2 checks to see if we are at the beginning of a line (calling `GotoCol` with a negative argument generates an error message). If the character to the left of the cursor *is* a digit, one of two possible actions can occur. If we have counted four digits, insert a separator, since three digits have passed since the end of the digit string or the last separator inserted, and then set the digit counter (register %!8) to zero again. If we have not yet encountered four digits, add one to the digit counter and move left one space.

Now, the trick is telling whether the character copied into register %!9 is a digit or not. Recall the discussion on page 11 about how to get the logical equivalent of a `switch` statement in WinEdt's language. We use that technique here: we have ten different `IfStr` tests to see whether the copied character is 0, . . . , 9. In 4.14, we have only listed the code for the first `IfStr` test (the complete code can be found in the Appendix).

Line 7 of 4.14 increments register %!8 by 1. Line 8 checks to see if this is the fourth digit found. If it is, all text is deselected by a call to `SetSel`. (At this point we have not yet deselected the character copied into register %!9. Without deselecting it, the `InsText` of line 11 would delete that digit!) Finally, we move the cursor to the left one column, reset the digit counter, and increment the counter tracking the number of separators inserted. Nothing is done if the character stored in %!9 was not the fourth digit found.

When the character in %!9 is not a digit, the ten `IfStr` tests in `_APchew` fail. Then %!6 gets assigned the value 1 (this occurs at line 155 in the full listing of the macro in the appendix). The `IfNum` test of macro 4.14, line 20, calls the shutdown routine. The complete listing of this macro is listed below:

```
// Filename: %B\Macros\ap\_APend.edt
GotoCol(%!7);
SetSel(0);
EndGroup;
StopWorking;
Exit;
// Endfile
```

The shutdown routine moves the cursor to the correct final position, deselects all text, ends the group started in `_APstart.edt`, and returns the cursor to its normal mode.



You might wonder why we hassle with the `%!6` flag indicating when the processing of the digit string has completed. The reason has to do out of concern for speed. `Exit` causes the macro to prematurely terminate, preventing it from backing out of the recursive `Exe` calls properly. Ordinarily, one wouldn't need to do this, but the multiple-way branching used in this macro results in a significant speed decrease.

## 5 Interactive Macros

WinEdt provides several macros that can be used to make your macros a little more user-friendly. The macros listed below fall into three types: getting data from the user (either numeric or string), prompting the user to make a choice at a certain key point in macro processing, or displaying the notorious Windows hourglass cursor (with an explanatory message in the WinEdt message area). The particular macros are:

`EnterLongReg(StrRegister: 0..9, "Prompt", "Caption")`

Prompts you to enter the value of the specified string register (`%!0...%!9`). Allows multi-line values.

`EnterReg(StrRegister: 0..9, "Prompt", "Caption")`

Prompts you to enter the value of the specified string register (`%!0...%!9`). The value is a one-line string.

`GetLongString("Prompt", "Caption")`

Same as `EnterLongReg` except that the value is stored in a special `%!?` string register (an extension of `%!0...%!9`)

`GetString("Prompt", "Caption")`

Same as `EnterReg` except that the value is stored in a special `%!?` string register (an extension of `%!0...%!9`)

`GetDim`

Displays the dialog allowing you to enter the values of `%!x` and `%!y` numeric registers. Their values are used in Macros that insert  $n \times m$  environments (such as `Array.edt`).

`GetCounter`

Displays the dialog that allows you to enter the values of `%!z` numeric register. Its value can be used for repeat loop and such...

`Prompt("Prompt", Type, Buttons: 0..3, "Macro 1", "Macro 2")`

Displays the dialog box with the specified prompt. Macro execution is resumed after the user closes the dialog box. 'Type' and 'Buttons' above can be any number in the range 0,1,2, or 3. These numbers alter the dialog boxes appearance and behavior in the following way: Types 0, 1, 2, or 3 tell the dialog box to appear as either a Information, Confirmation, Warning, or Error box, respectively. Button values 0, 1, 2, or 3 cause the dialog box to have OK/Cancel buttons (where clicking on Cancel terminates macro execution), an OK button, OK/Cancel buttons (where clicking on Cancel will invoke a special user macro), or Yes/No buttons.

`StartWorking("Message")`

Changes the cursor to the Hour Glass form and displays the specified message in the last field of the Status Line. Intended to indicate time-consuming macros,

`StopWorking`

Restores the cursor and removes the message from the status line.

With the exception of the last two macros (whose function should, hopefully, be more-or-less clear), all of these macros cause a dialog box to be displayed, allowing the user to enter a certain kind of data. If you've never done any Windows programming before, you might feel a certain rush of power at the ability to create a dialog box with such ease.

## 5.1 Getting data from the user

Close inspection will reveal some apparent redundancy between some of the above macros. For example, the macros `GetLongString` and `GetString` seem to be two ways of doing the same thing (sticking a string in the register `%!?`), and the difference between `EnterLongReg` and `EnterReg` also might not be immediately clear. The basic difference between them concerns the form of the dialog box that gets created. If you use either `GetLongString` or `EnterLongReg` the dialog box that appears has a large multi-line edit box allowing the user to type in whole paragraphs of text. On the other hand, `GetString` and `EnterReg` only use a single-line edit box, which (I suppose) implicitly tells the user to be concise. `WinEdt` doesn't perform any length checking on the argument for either of the short entry commands, so nothing prevents the user from typing an entire paragraph into the single-line edit box if they want.

The following macro illustrates one possible use of the `GetString` command:

```

1 BeginGroup;
2 GetString("What environment do you want to create?","Insert Environment");
3 LetRegNum(9,%c); // Store current column number
4 Mark(0);
5 InsText("\begin{?!?}");
6 NewLine;
7 GotoCol(!9);
8 InsText(" *"); NewLine;
9 GotoCol(!9);
10 InsText("\end{?!?}");
11 Goto(0);
12 CMD("Next Bullet");
13 EndGroup;

```

### 5.15 The Insert Environment macro

Line 1 begins a group so that the entire macro operation can be undone by a single undo operation. Line 2 contains the call to `GetString` that asks the user to type in the name of the environment that they want to insert. Line 3 stores the current column position of the cursor in the register `%!9` so that we can correctly position the rest of the lines.

For example, suppose the user wants to use this macro to insert a `cases` environment inside of an equation. Typically, this macro will be invoked when the user has typed only the following:

and so we define the incredibly important function as follows:

```

\[
  f(x) = *

```

where the cursor is at the point indicated by `*`. Now, *without* keeping track of the column that

the Insert Environment macro was invoked from (and consequently without including lines 3, 7, and 9 in the macro definition), the macro output would look like:

and so we define the incredibly important function as follows:

```
\[
  f(x) = \begin{cases}
    *
  \end{cases}
```

which is probably not what the user wanted. The `Newline` macro creates a new line and moves the cursor beneath the `f`, the `InsText(" *")` macro inserts the box directly beneath the `x`, and the second `Newline` macro creates a new line and moves the cursor directly beneath the `*` (assuming that WinEdt's default line wrapping conventions are on). Including lines 3, 7, and 9 causes the macro to insert the environment so that the `\begin` and `\end` have the correct vertical alignment like so:

and so we define the incredibly important function as follows:

```
\[
  f(x) = \begin{cases}
    *
  \end{cases}
```

## A An Example

```
1 SetSel(0); SetSel(1);
2 // Check to see if we are at the beginning of a line. If so,
3 // invoke __APend.
4 IfNum(%c,1,'=',>
5   'Exe("%B\Macros\AddPeriods\__APend.edt")',>
6   '');
7 GotoCol('(%c-1)'); GetSel(0,9);
8 IfStr('%!9','0','=',>
9   'LetRegNum(8,"%%!8+1");>
10  IfNum("%%!8","4","=",>
11    "SetSel(0);>
12    GotoCol('%%!c+1');>
13    InsText('%%!5');>
14    GotoCol('%%!c-1');>
15    LetRegNum(8,0);>
16    LetRegNum(7,'%%!7+1')",>
17    ">
18  );>
19  Exe("%B\Macros\AddPeriods\__APchew.edt")',>
20  ''>
21 );
22 IfNum(!6,1,'=', 'Exe("%B\Macros\AddPeriods\__APend.edt")', '');
23 IfStr('%!9','1','=',>
24   'LetRegNum(8,"%%!8+1");>
25   IfNum("%%!8","4","=",>
26     "SetSel(0);>
27     GotoCol('%%!c+1');>
28     InsText('%%!5');>
29     GotoCol('%%!c-1');>
30     LetRegNum(8,0);>
31     LetRegNum(7,'%%!7+1')",>
32     ">
33   );>
34   Exe("%B\Macros\AddPeriods\__APchew.edt")',>
35   ''>
36 );
37 IfNum(!6,1,'=', 'Exe("%B\Macros\AddPeriods\__APend.edt")', '');
38 IfStr('%!9','2','=',>
39   'LetRegNum(8,"%%!8+1");>
40   IfNum("%%!8","4","=",>
41     "SetSel(0);>
42     GotoCol('%%!c+1');>
43     InsText('%%!5');>
44     GotoCol('%%!c-1');>
45     LetRegNum(8,0);>
46     LetRegNum(7,'%%!7+1')",>
47     ">
48   );>
49   Exe("%B\Macros\AddPeriods\__APchew.edt")',>
50   ''>
51 );
52 IfNum(!6,1,'=', 'Exe("%B\Macros\AddPeriods\__APend.edt")', '');
53 IfStr('%!9','3','=',>
54   'LetRegNum(8,"%%!8+1");>
55   IfNum("%%!8","4","=",>
56     "SetSel(0);>
57     GotoCol('%%!c+1');>
```

## A AN EXAMPLE

---

```
58         InsText(''%%!!5'');>
59         GotoCol(''%%!!c-1'');>
60         LetRegNum(8,0);>
61         LetRegNum(7,''%%!!7+1''),>
62         "">
63     );>
64     Exe("%B\Macros\AddPeriods\__APchew.edt"),>
65     ''>
66 );
67 IfNum(!6,1,'=', 'Exe("%B\Macros\AddPeriods\__APend.edt"),');
68 IfStr('!9','4','=',>
69     'LetRegNum(8,"%%!8+1");>
70     IfNum("%%!8","4","=",>
71         "SetSel(0);>
72         GotoCol(''%%!!c+1'');>
73         InsText(''%%!!5'');>
74         GotoCol(''%%!!c-1'');>
75         LetRegNum(8,0);>
76         LetRegNum(7,''%%!!7+1''),>
77         "">
78     );>
79     Exe("%B\Macros\AddPeriods\__APchew.edt"),>
80     ''>
81 );
82 IfNum(!6,1,'=', 'Exe("%B\Macros\AddPeriods\__APend.edt"),');
83 IfStr('!9','5','=',>
84     'LetRegNum(8,"%%!8+1");>
85     IfNum("%%!8","4","=",>
86         "SetSel(0);>
87         GotoCol(''%%!!c+1'');>
88         InsText(''%%!!5'');>
89         GotoCol(''%%!!c-1'');>
90         LetRegNum(8,0);>
91         LetRegNum(7,''%%!!7+1''),>
92         "">
93     );>
94     Exe("%B\Macros\AddPeriods\__APchew.edt"),>
95     ''>
96 );
97 IfNum(!6,1,'=', 'Exe("%B\Macros\AddPeriods\__APend.edt"),');
98 IfStr('!9','6','=',>
99     'LetRegNum(8,"%%!8+1");>
100    IfNum("%%!8","4","=",>
101        "SetSel(0);>
102        GotoCol(''%%!!c+1'');>
103        InsText(''%%!!5'');>
104        GotoCol(''%%!!c-1'');>
105        LetRegNum(8,0);>
106        LetRegNum(7,''%%!!7+1''),>
107        "">
108    );>
109    Exe("%B\Macros\AddPeriods\__APchew.edt"),>
110    ''>
111 );
112 IfNum(!6,1,'=', 'Exe("%B\Macros\AddPeriods\__APend.edt"),');
113 IfStr('!9','7','=',>
114     'LetRegNum(8,"%%!8+1");>
115     IfNum("%%!8","4","=",>
116         "SetSel(0);>
117         GotoCol(''%%!!c+1'');>
118         InsText(''%%!!5'');>
119         GotoCol(''%%!!c-1'');>
```



## A AN EXAMPLE

---

```
120         LetRegNum(8,0);>
121         LetRegNum(7,'''%%!7+1'''),>
122         "">
123     );>
124     Exe("%B\Macros\AddPeriods\__APchew.edt"),>
125     ''>
126 );
127 IfNum(!6,1,'=', 'Exe("%B\Macros\AddPeriods\__APend.edt"),');
128 IfStr('!9','8','=',>
129     'LetRegNum(8,"%%!8+1");>
130     IfNum("%%!8","4","=",>
131         "SetSel(0);>
132         GotoCol(''%%!c+1'');>
133         InsText(''%%!5'');>
134         GotoCol(''%%!c-1'');>
135         LetRegNum(8,0);>
136         LetRegNum(7,'''%%!7+1'''),>
137         "">
138     );>
139     Exe("%B\Macros\AddPeriods\__APchew.edt"),>
140     ''>
141 );
142 IfNum(!6,1,'=', 'Exe("%B\Macros\AddPeriods\__APend.edt"),');
143 IfStr('!9','9','=',>
144     'LetRegNum(8,"%%!8+1");>
145     IfNum("%%!8","4","=",>
146         "SetSel(0);>
147         GotoCol(''%%!c+1'');>
148         InsText(''%%!5'');>
149         GotoCol(''%%!c-1'');>
150         LetRegNum(8,0);>
151         LetRegNum(7,'''%%!7+1'''),>
152         "">
153     );>
154     Exe("%B\Macros\AddPeriods\__APchew.edt"),>
155     'LetRegNum(6,1)'>
156 );
157 IfNum(!6,1,'=', 'Exe("%B\Macros\AddPeriods\__APend.edt"),');
```

## Contents

<b>1 Basics</b>	<b>2</b>
<b>2 Variables/Registers</b>	<b>4</b>
2.1 Strings and things . . . . .	5
2.2 Setting the values of registers . . . . .	10
<b>3 Procedures and Functions</b>	<b>13</b>
<b>4 Recursion</b>	<b>16</b>
4.1 A lengthy example . . . . .	16
<b>5 Interactive Macros</b>	<b>19</b>
5.1 Getting data from the user . . . . .	20
<b>A An Example</b>	<b>22</b>